Outline of Lectures

1. (Fri, 9/10) Administration, course overview. Informal discussion: what do we mean when we say two programs are "the same" or are "different"? Grammar of **IMP**.
2. (Mon, 9/13) Definition of kernel Scheme.
3. (Wed, 9/15) Extended Scheme vs. kernel Scheme: syntactic sugar.
4. (Fri, 9/17) The substitution model of kernel Scheme; rewrite rules.
5. (Mon, 9/20) The substitution model continued.
6. (Wed, 9/22) Definition of syntactic substitution. Declaratioun contexts and referential transparency.
7. (Fri, 9/24) Final lecture on the substitution model: evaluation contexts, deterministic evaluation.
8. (Mon, 9/27) Inductive definition of rules.
9. (Wed, 9/29) Proofs by induction.
10. (Fri, 10/1) What is a good inductive definition? What is a well-defined function?
11. (Mon, 10/4) Functional equations with no, many, unique, unique least solutions. Mention least fixed points.
12. (Wed, 10/6) Evaluation assertions for **IMP**. Functionality of arithmetic exp's by structural ind.
13. (Fri, 10/8) ADD DATE. Derivations in **IMP**. Example: Euclid's algorithm. Proofs by induction on derivations.

    (Mon, 10/11) COLUMBUS DAY.
14. (Wed, 10/13) What is the "meaning" of an **IMP** program?
15. (Fri, 10/15) QUIZ 1.
16. (Mon, 10/18) Compositionality of meaning, observational congruence, and full abstraction.
17. (Wed, 10/20) Proof of compositionality.
18. (Fri, 10/22) Introduction to Hoare logic. Partial correctness assertions. Rules of Hoare logic for **IMP**. Soundness.
19. (Mon, 10/25) Hoare logic rule for assignments. While-loop invariants.
20. (Wed, 10/27) Proof of correctness for Euclid's algorithm.
21. (Fri, 10/29) Language of assertions. Using assertions to encode primes and prime-powers.
22. (Mon, 11/1) Express "$i = k^n$" in **Assn** using base-p concatenation.
23. (Wed, 11/3) **IMP**-computable functions, predicates, and sets. Closure properties of **IMP**-decidable sets. "Computable" implies "expressible".
24. (Fri, 11/5) Expressibility of **IMP** input/output relations.
25. (Mon, 11/8) Checkable, decidable, and expressible sets.

    (Tue, 11/9, evening) QUIZ 2.

26. (Wed, 11/10) All **IMP**-computable functions are expressible. Dynamic assertions. Weakest preconditions. Expressing partial correctness assertions as dynamic assertions.
27. (Fri, 11/12) Relative completeness of Hoare logic for **IMP**. Closure properties of expressible sets.
28. (Mon, 11/15) More on closure properties of decidable and expressibile sets. Checkable sets. A set is decidable if and only if it is checkable and co-checkable.
29. (Wed, 11/17) Relative computability/expressibility. The halting problem. $\overline{H}$ is not checkable, and H is not decidable.
30. (Fri, 11/19) DROP DATE. $\bar{H}_0$ is not checkable. Many-one reducibility ($\leq_m$). Gödel encoding of commands.
31. (Mon, 11/22) H is checkable. $\overline{\text{True}}$ is not checkable. True is not checkable. Definition of provable; Peano arithmetic.
32. (Wed, 11/24) General definition of proof system as a decidable relation, $\vdash$, between a countable set, D, of derivations and an **Assn**. Theorem: for any proof system, $\vdash$, let **Provable** = $\{ A \mid d \vdash A \}$. Then **Provable** is checkable. Corollary (Incompleteness): for sound proof systems, **Provable** $\subsetneq$ **Valid**. Statement of Rice's Theorem.

    (Fri, 11/26) THANKSGIVING HOLIDAY.
33. (Mon, 11/29) Proof of Rice's Theorem. Examples of checkable, not decidable sets; $2H \cup 2H + 1$ is expressible but not checkable, not co-checkable. Statement of Theorem: **Validity** is not expressible.
34. (Wed, 12/1) Proof of Theorem: **Validity** is not expressible. Turing-reducibility ($\leq_T$). The Arithmetic Hierarchy and Hilbert's 10th Problem.
35. (Fri, 12/3) More on Hilbert's 10th Problem. The Diophantine sets = the checkable sets. Hilbert's 10th is undecidable, in fact uncheckable. Corollary: $\{ b \in \textbf{Bexp} \mid \ \models b \}$ is uncheckable. Corollary: it is uncheckable whether two **while**-free commands are equivalent.
36. (Mon, 12/6) Theorem: $\{ (a_1, a_2) \in \textbf{Aexp} \times \textbf{Aexp} \mid \ \models a_1 = a_2 \}$ is decidable and completely axiomatizable (by transformation to canonical form).
37. (Wed, 12/8) Further courses: 6.821 Programming Languages; 6.830 Program Semantics and Verification; 6.826 Principles of Systems; 18.505 Mathematical Logic. Further Topics: Undecidable problems in grammars, geometry, algebra, program schemes; logic programming; semantics of concurrency; denotational semantics of first-class functions; types in programming and constructive logic; formal logic, predicate calculus and completeness.

    (Wed, 12/15, 1:30–4:30 PM, room 24–115) FINAL EXAM.

Course Information

Staff.

Lecturer:	Prof. Albert R. Meyer `meyer@theory.lcs.mit.edu`	NE43-315	x3-6024
Teaching Assistant:	Trevor Jim `trevor@theory.lcs.mit.edu`	NE43-338	x3-7583
Secretary:	David Jones `6044-secretary@theory.lcs.mit.edu`	NE43-316	x3-5936

Lectures and Tutorials. Class meets MWF from 1:00–2:00 PM in 24-115. There will be no recitation sections, but tutorial/review sessions may be organized in response to requests. The TA will have one regularly scheduled office hour to be announced. Further meetings with the TA or instructor can be scheduled by appointment.

Prerequisites. The official prerequisite for the course is either 18.063 *Introduction to Algebraic Systems*, or 18.310 *Principles of Applied Mathematics.* Students who have taken 6.045J/18.400J or 6.840J/18.404J have more than met the prerequisites. (There will be less than 25% overlap between this course and 6.045J/18.400J or 6.840J/18.404J, so students who have taken either of these other courses are welcome to take this course.)

The courses 6.045J/18.400J or 6.840J/18.404J contain much more material than is necesssary for this course, because the actual prerequisite is knowledge of the basic vocabulary of mathematics and how to do elementary proofs. With such mathematical experience you should be able to handle this course; in this case *ask the instructor for permission* to take the course.

Textbook. The required text for the course is *Introduction to the Formal Semantics of Programming Languages* by Glynn Winskel, published by MIT Press in 1993. The book is available at the Coop.

Grading. There will be homework problems, two one hour quizzes, and a final exam. The problems and exams *count about equally* toward the final grade. The grading is nonlinear: ace the homework *or* the quizzes and you get an A, but counting on exam grades to outweigh neglected homework is a high-risk strategy.

Problem Sets. There will be problem sets most weeks. Homework will usually be assigned on a Friday and due the following one.

Handouts and Notebook. You may find it useful to get a loose-leaf notebook for use with the course, since all handouts and homework will be on standard three-hole punched paper. If you fail to obtain a handout in lecture, you can get a copy from the file cabinet to the right of the door to room NE43-311. If you take the last copy of a handout, please inform the course secretary, and get instructions on making more copies.

Handouts will also be available on-line in the 6.044 directory. To access this directory from Athena, type

```
attach -m /theory/6.044 -e theory.lcs.mit.edu:/pub/ftp/pub/6.044
source /theory/6.044/.athena_startup
```

(We recommend adding these lines to your `.environment` file, causing them to be executed every time you log in.) You will get a warning that "theory.lcs.mit.edu isn't registered with kerberos," which is entirely accurate but irrelevant. This will make the 6.044 directory available to you as `/theory/6.044` and tell LaTeX where to find the additional files it needs. All handouts are written in LaTeX.

If all else fails, the handouts can be retrieved via anonymous ftp or by mail from Theory. To retrieve these files by ftp, run `ftp theory.lcs.mit.edu`, supplying "anonymous" as the name (account) and "guest" as the password. Files may then be fetched by first typing "`cd pub/6.044`" to change directories and then typing "`get` *filename*". If you get the files in this way, you will also need to get the files `6.044.sty`, `handout.sty` and `handouts-6044-fall-92.tex` from `pub/6.044/input` in order to run LaTeX on the handout files. To find out about retrieving files by mail, send mail to `archive-server@theory.lcs.mit.edu` containing the single word "help" in the body of the message.

Electronic mail. All students are encouraged to subscribe to the course mail list by sending email to `6044-secretary@theory.lcs.mit.edu`; other administrative requests should also be directed to this address.

To facilitate communication in the class, there are three electronic mail addresses:

```
6044-secretary@theory.lcs.mit.edu
6044-forum@theory.lcs.mit.edu
6044-staff@theory.lcs.mit.edu
```

The `6044-forum` mailing list is for general communication by students, the instructor, and the TA to the class; a message sent here will automatically be distributed to those on the mailing list. Students are encouraged to use `6044-forum` to arrange study sessions, discuss ambiguities and problems with homework, and send comments to the whole class. The TA and instructor may also post bugs and corrections to homeworks and handouts to `6044-forum`.

Messages to the instructor or TA should be sent to `6044-staff`.

Pictures. You can help us learn who you are by giving us your photograph with your name on it. This is especially helpful if you later need a recommendation.

Diagnostic Quiz

You will not be graded on this quiz. Take it sometime after class, and return it in class on Monday, September 13. Be sure to indicate your name, the date and "6.044 Diagnostic Quiz" on your answer sheet.

Problem 1. Let *succ* be the successor function on integers:

$$succ(x) = x + 1$$

Describe the function $(succ \circ succ)$, that is, the composition of *succ* with itself.

Problem 2. How many strings of length 4 are there over the alphabet $\{a, b, c\}$?

Problem 3. Define $\preceq$ to be the binary relation between sets such that $A \preceq B$ if and only if the cardinality of A is less than or equal to the cardinality of B.

(a) What is the definition of "uncountable set"? Now express the definition in terms of $\preceq$. Give an example of an uncountable set.

(b) For each of the following properties, state whether the relation $\preceq$ has the property, and if not, give a simple counterexample.

1. reflexivity
2. symmetricity
3. transitivity

Problem 4. Two Boolean formulas $F_1(x_1, \ldots, x_n)$ and $F_2(x_1, \ldots, x_n)$ are *equivalent* iff they yield the same truth value for all truth assignments to the variables $x_1, \ldots, x_n$.

(a) The Boolean binary operation *conjunction* (and), which our text writes as "&", is commutative, namely x_1 & x_2 is equivalent to x_2 & x_1. Describe a Boolean binary operation which is not commutative.

(b) Describe an infinite set of equivalent Boolean formulas.

(c) Explain why "equivalent" is actually an equivalence relation on formulas.

(d) Explain why there are only a finite number of equivalence classes of formulas with (at most) variables $x_1, \ldots, x_n$.

(e) How many are there?

Problem 5. A binary relation $\leq$ is *anti-symmetric* if $x = y$ whenever $x \leq y$ and $y \leq x$. A *partial order* of a set D is a binary relation $\leq$ that is reflexive, transitive, and anti-symmetric on D. A partial order is *total* if for every $d_1, d_2 \in D$, either $d_1 \leq d_2$ or $d_2 \leq d_1$. If $X \subseteq D$ and $y \leq x$ for every $x \in X$, then y is a *lower bound* of X. Similarly, if $x \leq y$ for every $x \in X$, then y is an *upper bound* of X. If y is a lower bound of X, and $z \leq y$ for every lower bound z of X, then y is the *greatest lower bound* ("glb" or "meet") of X. If y is an upper bound of X, and $y \leq z$ for every upper bound z of X, then y is the *least upper bound* ("lub") of X.

Let A be the set $\{1, 2, 3, 4, 5\}$.

(a) Give a total order of A. That is, describe a binary relation $\leq$ on A that is a total order.

(b) Give a partial order of A such that: (1) every pair of elements has a glb; and (2) there is no lub of $\{4, 5\}$.

(c) Give a partial order of A such that every pair of elements has both an lub and a glb, but the order is not total.

Problem 6. For any set, A, let $\mathcal{P}\text{ow}(A)$ be the powerset of A, namely, the set of all subsets of A. Exhibit the members of $\mathcal{P}\text{ow}(\mathcal{P}\text{ow}(\mathcal{P}\text{ow}(\emptyset)))$.

Problem 7. Have you taken 18.063 or 18.310? If not, do you have any alternate qualifications that have prepared you for taking this course?

Problem 8. About how long did it take you to complete this quiz?

Outline of lectures from last year

This is an outline of the lectures given LAST YEAR (fall 1992) in 6.044. This may give you an idea of the pace of the course and the topics it covers. We do plan to make some changes in the course, but the outline should be about 75% accurate.

Lecture outline, 1992

1. (Fri, 9/11) Administrivia. Sample **IMP** while-program, Euclid, p.34; brief sketch of partial correctness and termination.

2. (Mon, 9/14) Syntax of **IMP**, and the "natural" evaluation semantics of **Aexp**. The derivation tree for $\langle (M+N)\times N, \sigma[10/N][6/M]\rangle \rightarrow 160$.

3. (Wed, 9/16) Natural eval rules for **Com**. Derivation tree for $\langle Euclid, \sigma[10/N][6/M]\rangle \rightarrow \sigma[2/M][2/N]$. Uniqueness of derivation tree for each configuration; exists for **Aexp**, **Bexp**, and *while-free* **Com**, but $\langle \textit{while true do}\, c, \sigma\rangle \not\rightarrow$ for all c, σ. No proofs.

4. (Fri, 9/18) One-step rules. Example $\langle Euclid, \sigma[10/N][6/M]\rangle \rightarrow_1^* \sigma[2/M][2/N]$. Remark: $\rightarrow_1$ is total, functional, computable relation. Inductive def of transitive closure. Statement of"equivalence" of one-step and natural rules: $\gamma \rightarrow_1^* \delta$ iff $\gamma \rightarrow \delta$ for all configurations γ and values $\delta \in N \cup \mathbf{T} \cup \Sigma$.

5. (Mon, 9/21) Proof of equiv of natural and one-step semantics.

6. (Wed, 9/23) Proof by induction on deriv. of functionality of command evaluation (Winsk, 3.11). Proof by minimum principle that $\langle \textit{while true do}\, c, \sigma\rangle \not\rightarrow$ (Winsk. 3.12).

7. (Fri, 9/25) Formal def of derivations, and induction on them (§3.4). Set R of rule instances determines a monotone, continuous, operator $\hat{R}$ on sets (§4.4) with derivable elements $= \mathit{fix}(\hat{R})$.

8. (Mon, 9/28) (Winskel §5.4) Def and examples of cpo's, monotone and continuous functions. Contrast with usual (epsilon-delta) continuity.

9. (Wed, 9/30) Examples of $\hat{R_0}(A)$ for $R_0 = \{\emptyset/3, \emptyset/4, \{n, n+1\}/n+2\}$. Proof that $\hat{R}$ is continuous. Proof of fixed points of continuous functions on cpo's.

10. (Fri, 10/2) QUIZ 1, IN CLASS, on lectures 1–8

11. (Mon, 10/5) Comments on Quiz 1. Discussion of wellformed and non-wellformed recursive function def's, eg, $e(x) = e(x+1)$, $f(x) = f(x+1)+1$, for g, h functions on ω^+: $g(1) = 1$; $g(x+y) = g(x)+g(y)$, $h(1) = 1$; $h(x+y) = h(x) + 2h(y)$. Function def by structural induction, eg, length and depth of a derivation, def of loc_L (§3.5) and statement w/o proof: c only effects $loc_L(c)$ (Winskel 4.7). Brief mention of capturing computational behavior of recursive def's by choosing *least* partial functions satisfying constraints.

12. (Wed, 10/7) Motivation for fixed points as explanation of recursion: While-loops as fixed points of mappings on command meanings. Command meanings, C, will be partial functions $\in \Sigma \rightharpoonup \Sigma$ (meanings of expressions will be total functions from states to **Num** or **T**). Statement of equivalence of denotational and natural semantics: Eval$(c) = \mathcal{C}[\![c]\!]$. Then define denotational semantics by structural induction assuming Γ_{while} (Winskel p.62) has a least fixed point.

13. (Fri, 10/9) Motivate Γ_{while} by considering $G : Com \rightarrow C$ where $G(while) =$ unwind-once-*while*. Outline proof that Eval(*while*) is fixed point of Γ; observe that there may be other fixed points: every comand is fixed point of $\Gamma_{\textit{while true do skip}}$. State that Eval(*while*) is *least* fixed point and Γ_{while} has least fixed point because it is continuous on cpo C.

    (Mon, 10/12) COLUMBUS DAY

14. (Wed, 10/14) Properties like $\text{Eval}((c_1; c_2)) = \text{Eval}(c_2) \circ \text{Eval}(c_1)$. Comments on proof of equivalence of natural and denotational semantics (Winskel Thm. 5.7): by structural induction, with subinduction for *while* case.

15. (Fri, 10/16) First-order arithmetic: **Assn**'s and their meaning. Assn's for "is prime," "divides," "lcm." Inductive def of free variables.

16. (Mon, 10/19) Formal def of $\sigma \models^I A$ (Winskel§6.3). Validity, satisfiability, invalidity.

17. (Wed, 10/21) Semantic of partial correctness; $\sigma \models \{true\}c\{false\}$ iff c diverges in state σ; A equiv $\{true\}\, skip\, \{A\}$. Sample axioms and rules of Hoare logic, mention soundness, hint about completeness and incompleteness.

18. (Fri, 10/23) Def of equivalent **Assn**'s. Lemma: A equiv B iff $\models (A \Rightarrow B)$ & $(B \Rightarrow A)$. Lemma: $\neg\forall j.A$ equiv $\exists j.\neg A$. Substitution Lemma: for expressions (WinskelLemma 6.8).

19. (Mon, 10/26) Substitution Lemma: for **Assn**'s (WinskelLemma 6.9). Prove validity of Hoare Assignment axiom. Lemma: A only depends on $FV(A)$ and $loc(A)$. Lemma: If $j \notin FV(A)$, then $\forall j.(A \vee B)$ equiv $(\forall j.A) \vee (\forall j.B)$ equiv $A \vee (\forall j.B)$. Proofs omitted.

    EVENING QUIZ 2, Mon, 10/26, on lectures 9, 11–18

20. (Wed, 10/28) Soundness of inference versus antecedents implying consequent. Mention optional exercise: which Hoare rules are valid as implications. Informal soundness of Hoare rules and proof example: Euclid.

21. (Fri, 10/30) Soundness of Hoare loop invariant rule. Weakest preconditions and Dynamic Assertions. Translate dynamic assertions into assertions, assuming expressiveness.

22. (Mon, 11/2) Prove expressiveness of **Assn** for **IMP**. Corollary: Relative completeness of Hoare logic.

23. (Wed, 11/4) Thm: Every **Assn** equiv to prenex.[polynomial = 0]. Intro to rules for **Aexp** equations and sum-of-products polynomial representation of **Aexp**'s.

24. (Fri, 11/6) Notion of canonical form. Deriving enough equational axioms to put **Aexp**'s into sum-of-monomials form.

25. (Mon, 11/9) Canonical forms for **Aexp**'s as polynomials-with-multivariate-polynomial-coefficients. Proof that distinct canonical forms have distinct meanings by induction on number of variables. Completeness and decidability for polynomial equations.

    (Wed, 11/11) VETERAN'S DAY

26. (Fri, 11/13) Gödel numbers of **Assn**'s. $A_{p(m,n)} \equiv A_n[m/i_0]$ for expressible p. Nonexpressibility of **Truth** for **Assn**'s.

27. (Mon, 11/16) QUIZ 3, IN CLASS, on lectures 19–25

28. (Wed, 11/18) Complete proof of nonexpressibility of **Truth**. Define **IMP** checkable and state Lemma: Checkable implies expressible. Mention incompleteness.

29. (Fri, 11/20) DROP DATE & Underground Guide Survey. Prove Checkable implies Expressible using expressiveness. Define **IMP**-decidable proof system as having **IMP**-decidable proof relation. State Lemma: **IMP**-decidable proof system has **IMP**-checkable set of provable Assn's, so *Provable* $\neq$ *Truth*.

30. (Mon, 11/23) Def of computable, decidable. Remark: **IMP**-computable same as IMP_r-computable by Handout—but not obvious. Decidable implies checkable. Thm: D decidable and f total computable implies f(D) checkable. Cor: Provable assertions are checkable.

31. (Wed, 11/25) Vocabulary: Checkable = r.e., decidable = recursive, computable = (partial) recursive. Decidable closed under intersection, complement. Mention dovetailing, f(r.e.) is r.e. Discussion of thesis that Effectively decidable = **IMP**-decidable. The set of (Gödel numbers of) sentences is a decidable set; likewise, the set of commands. Incompleteness Theorem: In a sound proof system, there is a true sentence which is not provable.

    (Fri, 11/27) THANKSGIVING HOLIDAY

32. (Mon, 11/30) Uncheckability of $\bar{H}$ where H is the self-halting problem, so undecidability of H. Venn diagram of decidable, checkable, co-checkable, expressible. Decidable iff checkable and co-checkable. Universal **IMP** command, u. Checkability of halting problem.

33. (Wed, 12/2) Uncheckability of zero-halting problem (by reduction). $\leq_m$ and Rice's Theorem.

34. (Fri, 12/4) Incompleteness of substitution instances of the single **Assn** $W(false, c_H)[n/X_1]$. Hilbert's 10th. Mention other undecidable problem: semigroup word problem; tiling problem (no time to mention zero matrix product problem; CFG equivalence and ambiguity problem, CSG emptiness).

35. (Mon, 12/7) IMP_{par}ala Brookes. Noncompositionality of state transition semantics. Def of observational congruence. X:=X not cong skip; some other identities do hold.

36. (Wed, 12/9) Compositional "interrupt sequence" semantics "interrupt sequences" fully abstract when n-ary-test-and-set is added to IMP_{par}. Comments on what was not covered: higher-order-**IMP**. Follow-up courses: 6.821 (programming linguistics and semantics), 6.830 (research in logic and semantics of programs), 6.840 (computability and complexity), 6.826 (Systems modelling and specification), Math and Philosophy courses in Logic.

    (Thu, 12/17) (Exam Period) QUIZ 4, 1:30–3:30 in du Pont, on lectures 26, 28–36

Diagnostic Quiz Solutions

Problem 1. Let *succ* be the successor function on integers:

$$succ(x) = x + 1$$

Describe the function $(succ \circ succ)$, that is, the composition of *succ* with itself.

The composition of *succ* with itself is the "add two" function, *add*2:

$$add2(x) = x + 2$$

because

$$\begin{aligned}(succ \circ succ)(x) &= succ(succ(x)) \\ &= succ(x+1) \\ &= (x+1)+1 \\ &= x+2 \\ &= add2(x)\end{aligned}$$

Problem 2. How many strings of length 4 are there over the alphabet $\{a, b, c\}$?

With three possibilities in each of four positions, there are $3^4 = 81$ possible strings.

Problem 3. Define $\preceq$ to be the binary relation between sets such that $A \preceq B$ if and only if the cardinality of A is less than or equal to the cardinality of B.

(a) What is the definition of "uncountable set"? Now express the definition in terms of $\preceq$. Give an example of an uncountable set.

A set A is uncountable iff there is no one-to-one and onto (bijective) correspondence between A and a subset of the integers. Thus, if N is the set of integers, then an uncountable set is any set A such that $A \not\preceq N$. The real numbers, $\mathbb{R}$, are a well-known example of an uncountable set, as is the powerset of any infinite set.

(b) For each of the following properties, state whether the relation $\preceq$ has the property, and if not, give a simple counterexample.

1. reflexivity
2. symmetricity
3. transitivity

The relation $\preceq$ is reflexive ($A \preceq A$ for all sets A) and transitive (if $A \preceq B$ and $B \preceq C$ then $A \preceq C$), but not symmetric (there are sets A and B with $A \preceq B$ but $B \not\preceq A$, for example N and $\mathbb{R}$).

Problem 4. Two Boolean formulas $F_1(x_1, \ldots, x_n)$ and $F_2(x_1, \ldots, x_n)$ are *equivalent* iff they yield the same truth value for all truth assignments to the variables $x_1, \ldots, x_n$.

(a) The Boolean binary operation *conjunction* (and), which our text writes as "&", is commutative, namely x_1 & x_2 is equivalent to x_2 & x_1. Describe a Boolean binary operation which is not commutative.

The "implies" operation, often written $\Rightarrow$, is non-commutative: $false \Rightarrow true$, but $true \not\Rightarrow false$, so $x_1 \Rightarrow x_2$ and $x_2 \Rightarrow x_1$ are not equivalent.

(b) Describe an infinite set of equivalent Boolean formulas.

Define the formulas F_i as $F_0 \stackrel{def}{=} x_0$ and $F_{i+1} \stackrel{def}{=} (F_i \mathbin{\&} x_0)$ for all $i \geq 0$. These formulas are all equivalent, since for any truth assignment they all have the same truth value as x_0.

(c) Explain why "equivalent" is actually an equivalence relation on formulas.

An equivalence relation is a binary relation on a set A, that is, $R \subseteq A \times A$, which is reflexive ($a \mathrel{R} a$ for all $a \in A$), transitive (if $a \mathrel{R} a'$ and $a' \mathrel{R} a''$ then $a \mathrel{R} a''$), and symmetric ($a \mathrel{R} a'$ iff $a' \mathrel{R} a$).

Take any three formulas F_0, F_1 and F_2. Given any truth assignment to $x_1, \ldots, x_n$:

- F_0 yields the same truth value as itself, so "equivalent" is reflexive.
- If F_0 yields the same truth value as F_1, then F_1 yields the same truth value as F_0, so "equivalent" is symmetric.
- If F_0 yields the same truth value as F_1, and F_1 yields the same truth value as F_2, then clearly F_0 yields the same truth value as F_2, so "equivalent" is transitive.

Thus, "equivalent" is an equivalence relation on formulas.

(d) Explain why there are only a finite number of equivalence classes of formulas with (at most) variables $x_1, \ldots, x_n$.

(e) How many?

Formulas are equivalent iff their truth tables agree, so there are as many equivalence classes as there are truth tables. Given n variables, there are 2^{2^n} possible truth tables. To see this, think of a truth table as having a row for each possible truth assignment. A truth assignment consists of a *true* or *false* value for each variable, so there are (size of $\{true, false\})^n = 2^n$ possible truth assignments. Then, a truth table consists of an assignment of *true* or *false* to each truth assignment, so with 2^n truth assignments there are 2^{2^n} possible truth tables, giving us 2^{2^n} equivalence classes of formulas.

Problem 5. A binary relation $\leq$ is *anti-symmetric* if $x = y$ whenever $x \leq y$ and $y \leq x$. A *partial order* of a set D is a binary relation $\leq$ that is reflexive, transitive, and anti-symmetric on D. A partial order is *total* if for every $d_1, d_2 \in D$, either $d_1 \leq d_2$ or $d_2 \leq d_1$. If $X \subseteq D$ and $y \leq x$ for every $x \in X$, then y is a *lower bound* of X. Similarly, if $x \leq y$ for every $x \in X$, then y is an *upper bound* of X. If y is a lower bound of X, and $z \leq y$ for every lower bound z of X, then y is the *greatest lower bound* ("glb" or "meet") of X. If y is an upper bound of X, and $y \leq z$ for every upper bound z of X, then y is the *least upper bound* ("lub") of X.

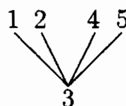
Let A be the set $\{1, 2, 3, 4, 5\}$.

(a) Give a total order of A. That is, describe a binary relation $\leq$ on A that is a total order.

The usual numerical "less than or equal to" relation is a total order.

(b) Give a partial order of A such that: (1) every pair of elements has a glb; and (2) there is no lub of $\{4, 5\}$.

The ordering in which 3 is less than or equal to any other number, but all other numbers are incomparable, is such an order. In pictures, we can draw the order as follows:

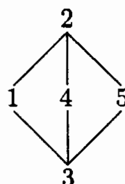


Here we are using a common graphical shorthand for partial orders, in which an element is "less than" another element if we can trace an upwards path from the first element to the second. Since there is an upwards path from 3 to every other number, we have $3 \leq x$ for all $x \in A$. (There is an upwards path from 3 to 3, namely the upwards path of length 0). However, $1 \not\leq 4$ since there is no upwards path from 1 to 4, and $4 \not\leq 1$ since there is no upwards path from 4 to 1; that is, 1 and 4 are incomparable.

From the diagram, it is clear that every pair of distinct numbers has a unique lower bound in the ordering, namely 3. Since there is only one lower bound possible, it is clearly the greatest lower bound. However, 4 and 5 have no upper bound at all, let alone a least upper bound.

(c) Give a partial order of A such that every pair of elements has both an lub and a glb, but the order is not total.

A slight variant, in which we leave 3 as the least element but make 2 the greatest, gives us an ordering in which every pair of elements has both a least upper bound and a greatest lower bound, but the ordering is still not total.



Problem 6. For any set, A, let $\mathcal{P}\text{ow}(A)$ be the powerset of A, namely, the set of all subsets of A. Exhibit the members of $\mathcal{P}\text{ow}(\mathcal{P}\text{ow}(\mathcal{P}\text{ow}(\emptyset)))$.

Given the empty set, $\emptyset$, with no members, the only subset is the (non-proper) subset $\emptyset$. Thus, if $\mathcal{P}\text{ow}(\emptyset)$ is the set of all subsets of $\emptyset$, then

$$\mathcal{P}\text{ow}(\emptyset) = \{\emptyset\}.$$

Moving along, the subsets of $\mathcal{P}\text{ow}(\emptyset)$ are $\emptyset$ and $\{\emptyset\}$, so

$$\mathcal{P}\text{ow}(\mathcal{P}\text{ow}(\emptyset)) = \{\emptyset, \{\emptyset\}\}.$$

Then, $\mathcal{P}\text{ow}(\mathcal{P}\text{ow}(\emptyset))$ has subsets $\emptyset$, $\{\emptyset\}$, $\{\{\emptyset\}\}$, and $\{\emptyset, \{\emptyset\}\}$, so

$$\mathcal{P}\text{ow}(\mathcal{P}\text{ow}(\mathcal{P}\text{ow}(\emptyset))) = \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\}$$

```
SUB-EVAL==> (define (factorial n)
  (if (<= n 0)
      1
      (* n (factorial (- n 1)))))


(define (factorial n) (if (<= n 0) 1 (* n (factorial (- n 1)))))
--------
factorial has been defined

*value-not-specified*
SUB-EVAL==> (factorial 0)
(factorial 0)
1 --------

((lambda (n) (if (<= n 0) 1 (* n (factorial (- n 1))))) 0)
2 --------

(letrec ((n 0)) (if (<= n 0) 1 (* n (factorial (- n 1)))))
3 --------

(letrec ((n 0)) (if (<<<=>> 0 0) 1 (* n (factorial (- n 1)))))
5 --------

1
SUB-EVAL==> (factorial 1)
(factorial 1)
1 --------

((lambda (n) (if (<= n 0) 1 (* n (factorial (- n 1))))) 1)
2 --------

(letrec ((n 1)) (if (<= n 0) 1 (* n (factorial (- n 1)))))
3 --------

(letrec ((n 1)) (if (<<<=>> 1 0) 1 (* n (factorial (- n 1)))))
5 --------

(letrec ((n 1)) (* n (factorial (- n 1))))
8 --------

(letrec ((n 1)) (<<*>> 1 (factorial (- n 1))))
10 --------

(letrec
 ((n 1))
 (<<*>> 1 ((lambda (n) (if (<= n 0) 1 (* n (factorial (- n 1))))) (- n 1))))
11 --------

(<<*>> 1 ((lambda (n) (if (<= n 0) 1 (* n (factorial (- n 1))))) (<<->> 1 1)))
13 --------

(<<*>> 1 (letrec ((n 0)) (if (<= n 0) 1 (* n (factorial (- n 1))))))
15 --------

(<<*>> 1 (letrec ((n 0)) (if (<<<=>> 0 0) 1 (* n (factorial (- n 1))))))
```

```
17 --------

(<<*>> 1 1)
19 --------

1
SUB-EVAL==> (factorial 2)
(factorial 2)
1 --------

((lambda (n) (if (<= n 0) 1 (* n (factorial (- n 1))))) 2)
2 --------

(letrec ((n 2)) (if (<= n 0) 1 (* n (factorial (- n 1)))))
3 --------

(letrec ((n 2)) (if (<<<=>> 2 0) 1 (* n (factorial (- n 1)))))
5 --------

(letrec ((n 2)) (* n (factorial (- n 1))))
8 --------

(letrec ((n 2)) (<<*>> 2 (factorial (- n 1))))
10 --------

(letrec
 ((n 2))
 (<<*>> 2 ((lambda (n) (if (<= n 0) 1 (* n (factorial (- n 1))))) (- n 1))))
11 --------

(<<*>> 2 ((lambda (n) (if (<= n 0) 1 (* n (factorial (- n 1))))) (<<->> 2 1)))
13 --------

(<<*>> 2 (letrec ((n 1)) (if (<= n 0) 1 (* n (factorial (- n 1))))))
15 --------

(<<*>> 2 (letrec ((n 1)) (if (<<<=>> 1 0) 1 (* n (factorial (- n 1))))))
17 --------

(<<*>> 2 (letrec ((n 1)) (* n (factorial (- n 1)))))
20 --------

(<<*>> 2 (letrec ((n 1)) (<<*>> 1 (factorial (- n 1)))))
22 --------

(<<*>>
 2
 (letrec
  ((n 1))
  (<<*>> 1 ((lambda (n) (if (<= n 0) 1 (* n (factorial (- n 1))))) (- n 1)))))
23 --------

(<<*>>
 2
 (<<*>>
  1
```

```
  ((lambda (n) (if (<= n 0) 1 (* n (factorial (- n 1))))) (<<->> 1 1))))
25 --------

(<<*>> 2 (<<*>> 1 (letrec ((n 0)) (if (<= n 0) 1 (* n (factorial (- n 1)))))))
27 --------

(<<*>>
 2
 (<<*>> 1 (letrec ((n 0)) (if (<<<=>> 0 0) 1 (* n (factorial (- n 1)))))))
29 --------

(<<*>> 2 (<<*>> 1 1))
31 --------

2
```

```
SUB-EVAL==>  (define (zero f) (lambda (x) x))

(define (zero f) (lambda (x) x))
--------
zero has been defined

*value-not-specified*
SUB-EVAL==>  (define (one f) (lambda (x) (f x)))

(define (one f) (lambda (x) (f x)))
--------
one has been defined

*value-not-specified*
SUB-EVAL==>  (define (suc n) (lambda (f) (lambda (x) (f ((n f) x)))))

(define (suc n) (lambda (f) (lambda (x) (f ((n f) x)))))
--------
suc has been defined

*value-not-specified*
SUB-EVAL==>  (define (to-num n) ((n 1+) 0))

(define (to-num n) ((n 1+) 0))
--------
to-num has been defined

*value-not-specified*
SUB-EVAL==>  (define (x^x n) (n n))
(define (x^x n) (n n))
--------
x^x has been defined

*value-not-specified*
SUB-EVAL==>

(x^x one)
(x^x one)
1 --------

((lambda (n) (n n)) one)
2 --------

((lambda (n) (n n)) (lambda (f) (lambda (x) (f x))))
3 --------

(letrec ((n (lambda (f) (lambda (x) (f x))))) (n n))
4 --------

(letrec
 ((n (lambda (f) (lambda (x) (f x)))))
 ((lambda (f) (lambda (x) (f x))) n))
5 --------

((lambda (f) (lambda (x) (f x))) (lambda (f) (lambda (x) (f x))))
6 --------
```

```
(letrec ((f (lambda (f) (lambda (x) (f x))))) (lambda (x) (f x)))
SUB-EVAL==> (suc zero)
(suc zero)
1 --------

((lambda (n) (lambda (f) (lambda (x) (f ((n f) x))))) zero)
2 --------

((lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))))
 (lambda (f) (lambda (x) x)))
3 --------

(letrec
 ((n (lambda (f) (lambda (x) x))))
 (lambda (f) (lambda (x) (f ((n f) x)))))
SUB-EVAL==> (((suc one) 1+) 3)
(((suc one) 1+) 3)
1 --------

((((lambda (n) (lambda (f) (lambda (x) (f ((n f) x))))) one) 1+) 3)
2 --------

((((lambda (n) (lambda (f) (lambda (x) (f ((n f) x)))))
   (lambda (f) (lambda (x) (f x))))
  1+)
 3)
3 --------

(((letrec
   ((n (lambda (f) (lambda (x) (f x)))))
   (lambda (f) (lambda (x) (f ((n f) x)))))
  1+)
 3)
4 --------

((letrec
  ((n (lambda (f) (lambda (x) (f x)))))
  ((lambda (f) (lambda (x) (f ((n f) x)))) <<1+>>))
 3)
6 --------

((letrec
  ((n (lambda (f) (lambda (x) (f x)))))
  (letrec ((f <<1+>>)) (lambda (x) (f ((n f) x)))))
 3)
7 --------

(letrec
 ((n (lambda (f) (lambda (x) (f x)))))
 ((letrec ((f <<1+>>)) (lambda (x) (f ((n f) x)))) 3))
8 --------

(letrec
 ((n (lambda (f) (lambda (x) (f x)))))
 (letrec ((f <<1+>>)) ((lambda (x) (f ((n f) x))) 3)))
```

```
9 --------

(letrec
 ((n (lambda (f) (lambda (x) (f x)))))
 (letrec ((f <<1+>>)) (letrec ((x 3)) (f ((n f) x)))))
10 --------

(letrec
 ((n (lambda (f) (lambda (x) (f x)))))
 (letrec ((f <<1+>>)) (letrec ((x 3)) (<<1+>> ((n f) x)))))
11 --------

(letrec
 ((f <<1+>>))
 (letrec ((x 3)) (<<1+>> (((lambda (f) (lambda (x) (f x))) f) x))))
12 --------

(letrec ((x 3)) (<<1+>> (((lambda (f) (lambda (x) (f x))) <<1+>>) x)))
13 --------

(letrec ((x 3)) (<<1+>> ((letrec ((f <<1+>>)) (lambda (x) (f x))) x)))
14 --------

(<<1+>> ((letrec ((f <<1+>>)) (lambda (x) (f x))) 3))
15 --------

(<<1+>> (letrec ((f <<1+>>)) ((lambda (x) (f x)) 3)))
16 --------

(<<1+>> (letrec ((f <<1+>>)) (letrec ((x 3)) (f x))))
17 --------

(<<1+>> (letrec ((x 3)) (<<1+>> x)))
18 --------

(<<1+>> (<<1+>> 3))
19 --------

5
```

```
SUB-EVAL==> (define (update new-passkey passkey-checker)
  (define (updated-checker passkey)
    (if (= passkey new-passkey) #t (passkey-checker passkey)))
  updated-checker)

(define
 (update new-passkey passkey-checker)
 (define
  (updated-checker passkey)
  (if (= passkey new-passkey) #T (passkey-checker passkey)))
 updated-checker)
--------
update has been defined

*value-not-specified*
SUB-EVAL==> (define keychecker (update 3 (update 2 (update 1 (lambda (key) #f)))))
(define keychecker (update 3 (update 2 (update 1 (lambda (key) ())))))
--------

(define
 keychecker
 ((lambda
   (new-passkey passkey-checker)
   (letrec
    ((updated-checker
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))
  3
  (update 2 (update 1 (lambda (key) ())))))
--------

(define
 keychecker
 ((lambda
   (new-passkey passkey-checker)
   (letrec
    ((updated-checker
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))
  3
  ((lambda
    (new-passkey passkey-checker)
    (letrec
     ((updated-checker
       (lambda
        (passkey)
        (if (= passkey new-passkey) #T (passkey-checker passkey)))))
     updated-checker))
   2
   (update 1 (lambda (key) ())))))
--------
```

```
(define
 keychecker
 ((lambda
   (new-passkey passkey-checker)
   (letrec
    ((updated-checker
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))
  3
  ((lambda
    (new-passkey passkey-checker)
    (letrec
     ((updated-checker
       (lambda
        (passkey)
        (if (= passkey new-passkey) #T (passkey-checker passkey)))))
     updated-checker))
   2
   ((lambda
     (new-passkey passkey-checker)
     (letrec
      ((updated-checker
        (lambda
         (passkey)
         (if (= passkey new-passkey) #T (passkey-checker passkey)))))
      updated-checker))
    1
    (lambda (key) ())))))
--------

(define
 keychecker
 ((lambda
   (new-passkey passkey-checker)
   (letrec
    ((updated-checker
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))
  3
  ((lambda
    (new-passkey passkey-checker)
    (letrec
     ((updated-checker
       (lambda
        (passkey)
        (if (= passkey new-passkey) #T (passkey-checker passkey)))))
     updated-checker))
   2
   (letrec
    ((new-passkey 1) (passkey-checker (lambda (key) ())))
    (letrec
     ((updated-checker
```

```
       (lambda
        (passkey)
        (if (= passkey new-passkey) #T (passkey-checker passkey)))))
     updated-checker)))))
--------

(define
 keychecker
 ((lambda
   (new-passkey passkey-checker)
   (letrec
    ((updated-checker
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))
  3
  ((lambda
    (new-passkey passkey-checker)
    (letrec
     ((updated-checker
       (lambda
        (passkey)
        (if (= passkey new-passkey) #T (passkey-checker passkey)))))
     updated-checker))
   2
   (letrec
    ((new-passkey 1) (passkey-checker (lambda (key) ())))
    (lambda
     (passkey)
     (if (= passkey new-passkey) #T (passkey-checker passkey)))))))
--------

(define
 keychecker
 ((lambda
   (new-passkey passkey-checker)
   (letrec
    ((updated-checker
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))
  3
  (letrec
   ((new-passkey 2)
    (passkey-checker
     (letrec
      ((new-passkey 1) (passkey-checker (lambda (key) ())))
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey))))))
   (letrec
    ((updated-checker
      (lambda
       (passkey)
```

```
        (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))))
--------

(define
 keychecker
 ((lambda
   (new-passkey passkey-checker)
   (letrec
    ((updated-checker
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))
  3
  (letrec
   ((new-passkey 2)
    (new-passkey#1 1)
    (passkey-checker#1 (lambda (key) ()))
    (passkey-checker
     (lambda
      (passkey)
      (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))))
   (letrec
    ((updated-checker
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))))
--------

(define
 keychecker
 ((lambda
   (new-passkey passkey-checker)
   (letrec
    ((updated-checker
      (lambda
       (passkey)
       (if (= passkey new-passkey) #T (passkey-checker passkey)))))
    updated-checker))
  3
  (letrec
   ((new-passkey 2)
    (new-passkey#1 1)
    (passkey-checker#1 (lambda (key) ()))
    (passkey-checker
     (lambda
      (passkey)
      (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))))
   (lambda
    (passkey)
    (if (= passkey new-passkey) #T (passkey-checker passkey))))))
--------

(define
```

```
 keychecker
 (letrec
  ((new-passkey 3)
   (passkey-checker
    (letrec
     ((new-passkey 2)
      (new-passkey#1 1)
      (passkey-checker#1 (lambda (key) ()))
      (passkey-checker
       (lambda
        (passkey)
        (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))))
     (lambda
      (passkey)
      (if (= passkey new-passkey) #T (passkey-checker passkey))))))
  (letrec
   ((updated-checker
     (lambda
      (passkey)
      (if (= passkey new-passkey) #T (passkey-checker passkey)))))
   updated-checker)))
--------

(define
 keychecker
 (letrec
  ((new-passkey 3)
   (new-passkey#2 2)
   (new-passkey#1 1)
   (passkey-checker#1 (lambda (key) ()))
   (passkey-checker#2
    (lambda
     (passkey)
     (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
   (passkey-checker
    (lambda
     (passkey)
     (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
  (letrec
   ((updated-checker
     (lambda
      (passkey)
      (if (= passkey new-passkey) #T (passkey-checker passkey)))))
   updated-checker)))
--------

(define
 keychecker
 (letrec
  ((new-passkey 3)
   (new-passkey#2 2)
   (new-passkey#1 1)
   (passkey-checker#1 (lambda (key) ()))
   (passkey-checker#2
    (lambda
     (passkey)
```

```
      (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
    (passkey-checker
     (lambda
      (passkey)
      (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
   (lambda
    (passkey)
    (if (= passkey new-passkey) #T (passkey-checker passkey)))))
--------
keychecker has been defined

*value-not-specified*
SUB-EVAL==> (keychecker 3)
(keychecker 3)
1 --------

((letrec
  ((new-passkey 3)
   (new-passkey#2 2)
   (new-passkey#1 1)
   (passkey-checker#1 (lambda (key) ()))
   (passkey-checker#2
    (lambda
     (passkey)
     (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
   (passkey-checker
    (lambda
     (passkey)
     (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
  (lambda (passkey) (if (= passkey new-passkey) #T (passkey-checker passkey))))
 3)
2 --------

(letrec
 ((new-passkey 3)
  (new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
  (passkey-checker
   (lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
 ((lambda (passkey) (if (= passkey new-passkey) #T (passkey-checker passkey)))
  3))
3 --------

(letrec
 ((new-passkey 3)
  (new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
```

```
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
  (passkey-checker
   (lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
 (letrec
  ((passkey 3))
  (if (= passkey new-passkey) #T (passkey-checker passkey))))
4 --------

(letrec
 ((new-passkey 3)
  (new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
  (passkey-checker
   (lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
 (letrec
  ((passkey 3))
  (if (<<=>> 3 new-passkey) #T (passkey-checker passkey))))
6 --------

(letrec
 ((new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
  (passkey-checker
   (lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
 (letrec ((passkey 3)) (if (<<=>> 3 3) #T (passkey-checker passkey))))
7 --------

#T
SUB-EVAL==> (keychecker 0)
(keychecker 0)
1 --------

((letrec
  ((new-passkey 3)
   (new-passkey#2 2)
   (new-passkey#1 1)
   (passkey-checker#1 (lambda (key) ()))
   (passkey-checker#2
```

```
    (lambda
     (passkey)
     (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
   (passkey-checker
    (lambda
     (passkey)
     (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
  (lambda (passkey) (if (= passkey new-passkey) #T (passkey-checker passkey))))
 0)
2 --------

(letrec
 ((new-passkey 3)
  (new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
  (passkey-checker
   (lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
 ((lambda (passkey) (if (= passkey new-passkey) #T (passkey-checker passkey)))
  0))
3 --------

(letrec
 ((new-passkey 3)
  (new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
  (passkey-checker
   (lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
 (letrec
  ((passkey 0))
  (if (= passkey new-passkey) #T (passkey-checker passkey))))
4 --------

(letrec
 ((new-passkey 3)
  (new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
  (passkey-checker
```

```
   (lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
 (letrec
  ((passkey 0))
  (if (<<=>> 0 new-passkey) #T (passkey-checker passkey))))
6 --------

(letrec
 ((new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
  (passkey-checker
   (lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
 (letrec ((passkey 0)) (if (<<=>> 0 3) #T (passkey-checker passkey))))
7 --------

(letrec
 ((new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
  (passkey-checker
   (lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))))
 (letrec ((passkey 0)) (passkey-checker passkey)))
10 --------

(letrec
 ((new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))))
 (letrec
  ((passkey 0))
  ((lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))
   passkey)))
11 --------

(letrec
 ((new-passkey#2 2)
```

```
   (new-passkey#1 1)
   (passkey-checker#1 (lambda (key) ()))
   (passkey-checker#2
    (lambda
     (passkey)
     (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))))
 ((lambda
    (passkey)
    (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey)))
  0))
12 --------

(letrec
 ((new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))))
 (letrec
  ((passkey 0))
  (if (= passkey new-passkey#2) #T (passkey-checker#2 passkey))))
13 --------

(letrec
 ((new-passkey#2 2)
  (new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))))
 (letrec
  ((passkey 0))
  (if (<<=>> 0 new-passkey#2) #T (passkey-checker#2 passkey))))
15 --------

(letrec
 ((new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))))
 (letrec ((passkey 0)) (if (<<=>> 0 2) #T (passkey-checker#2 passkey))))
16 --------

(letrec
 ((new-passkey#1 1)
  (passkey-checker#1 (lambda (key) ()))
  (passkey-checker#2
   (lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))))
 (letrec ((passkey 0)) (passkey-checker#2 passkey)))
```

```
19 --------

(letrec
 ((new-passkey#1 1) (passkey-checker#1 (lambda (key) ())))
 (letrec
  ((passkey 0))
  ((lambda
    (passkey)
    (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))
   passkey)))
20 --------

(letrec
 ((new-passkey#1 1) (passkey-checker#1 (lambda (key) ())))
 ((lambda
   (passkey)
   (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey)))
  0))
21 --------

(letrec
 ((new-passkey#1 1) (passkey-checker#1 (lambda (key) ())))
 (letrec
  ((passkey 0))
  (if (= passkey new-passkey#1) #T (passkey-checker#1 passkey))))
22 --------

(letrec
 ((new-passkey#1 1) (passkey-checker#1 (lambda (key) ())))
 (letrec
  ((passkey 0))
  (if (<<=>> 0 new-passkey#1) #T (passkey-checker#1 passkey))))
24 --------

(letrec
 ((passkey-checker#1 (lambda (key) ())))
 (letrec ((passkey 0)) (if (<<=>> 0 1) #T (passkey-checker#1 passkey))))
25 --------

(letrec
 ((passkey-checker#1 (lambda (key) ())))
 (letrec ((passkey 0)) (passkey-checker#1 passkey)))
28 --------

(letrec ((passkey 0)) ((lambda (key) ()) passkey))
29 --------

((lambda (key) ()) 0)
30 --------

()
SUB-EVAL==> quit
*value-not-specified*
Exiting from SUB-EVAL...done
;Value: done
```

Substitution Model Examples

We begin with the usual recursive definition of factorial:

```
SUB-EVAL==>
(define
 rec-factorial
 (lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))))

--------
rec-factorial has been defined
```

Now we let it run:

```
SUB-EVAL==>
(rec-factorial 5)
==[1]==>

((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))) 5)
==[2]==>

(letrec ((n 5)) (if (<= n 0) 1 (* n (rec-factorial (- n 1)))))
==[3]==>

(letrec ((n 5)) (if (<<<=>> 5 0) 1 (* n (rec-factorial (- n 1)))))
==[5]==>

(letrec ((n 5)) (* n (rec-factorial (- n 1))))
==[7]==>

(letrec ((n 5)) (<<*>> 5 (rec-factorial (- n 1))))
==[9]==>

(letrec
 ((n 5))
 (<<*>>
  5
  ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))) (- n 1))))
==[10]==>

(<<*>>
 5
 ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))) (<<->> 5 1)))
==[12]==>

(<<*>> 5 (letrec ((n 4)) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))))
==[14]==>

(<<*>> 5 (letrec ((n 4)) (if (<<<=>> 4 0) 1 (* n (rec-factorial (- n 1))))))
```

```
==[16]==>

(<<*>> 5 (letrec ((n 4)) (* n (rec-factorial (- n 1)))))
==[18]==>

(<<*>> 5 (letrec ((n 4)) (<<*>> 4 (rec-factorial (- n 1)))))
==[20]==>

(<<*>>
 5
 (letrec
  ((n 4))
  (<<*>>
   4
   ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))) (- n 1)))))
==[21]==>

(<<*>>
 5
 (<<*>>
  4
  ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))) (<<->> 4 1))))
==[23]==>

(<<*>>
 5
 (<<*>> 4 (letrec ((n 3)) (if (<= n 0) 1 (* n (rec-factorial (- n 1)))))))
==[25]==>

(<<*>>
 5
 (<<*>> 4 (letrec ((n 3)) (if (<<<=>> 3 0) 1 (* n (rec-factorial (- n 1)))))))
==[27]==>

(<<*>> 5 (<<*>> 4 (letrec ((n 3)) (* n (rec-factorial (- n 1))))))
==[29]==>

(<<*>> 5 (<<*>> 4 (letrec ((n 3)) (<<*>> 3 (rec-factorial (- n 1))))))
==[31]==>

(<<*>>
 5
 (<<*>>
  4
  (letrec
   ((n 3))
   (<<*>>
    3
    ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))) (- n 1))))))
==[32]==>

(<<*>>
 5
```



```
 (<<*>>
  4
  (<<*>>
   3
   ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))) (<<->> 3 1)))))
==[34]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>> 3 (letrec ((n 2)) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))))))
==[36]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>>
   3
   (letrec ((n 2)) (if (<<<=>> 2 0) 1 (* n (rec-factorial (- n 1))))))))
==[38]==>

(<<*>> 5 (<<*>> 4 (<<*>> 3 (letrec ((n 2)) (* n (rec-factorial (- n 1)))))))
==[40]==>

(<<*>>
 5
 (<<*>> 4 (<<*>> 3 (letrec ((n 2)) (<<*>> 2 (rec-factorial (- n 1)))))))
==[42]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>>
   3
   (letrec
    ((n 2))
    (<<*>>
     2
     ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))) (- n 1)))))))
==[43]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>>
   3
   (<<*>>
    2
    ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1)))))
```

```
    (<<->> 2 1))))))
==[45]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>>
   3
   (<<*>> 2 (letrec ((n 1)) (if (<= n 0) 1 (* n (rec-factorial (- n 1)))))))))
==[47]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>>
   3
   (<<*>>
    2
    (letrec ((n 1)) (if (<<<=>> 1 0) 1 (* n (rec-factorial (- n 1)))))))))
==[49]==>

(<<*>>
 5
 (<<*>> 4 (<<*>> 3 (<<*>> 2 (letrec ((n 1)) (* n (rec-factorial (- n 1))))))))
==[51]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>> 3 (<<*>> 2 (letrec ((n 1)) (<<*>> 1 (rec-factorial (- n 1))))))))
==[53]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>>
   3
   (<<*>>
    2
    (letrec
     ((n 1))
     (<<*>>
      1
      ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1)))))
       (- n 1))))))))
==[54]==>

(<<*>>
 5
```



```
 (<<*>>
  4
  (<<*>>
   3
   (<<*>>
    2
    (<<*>>
     1
     ((lambda (n) (if (<= n 0) 1 (* n (rec-factorial (- n 1)))))
      (<<->> 1 1)))))))
==[56]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>>
   3
   (<<*>>
    2
    (<<*>>
     1
     (letrec ((n 0)) (if (<= n 0) 1 (* n (rec-factorial (- n 1))))))))))
==[58]==>

(<<*>>
 5
 (<<*>>
  4
  (<<*>>
   3
   (<<*>>
    2
    (<<*>>
     1
     (letrec ((n 0)) (if (<<<=>> 0 0) 1 (* n (rec-factorial (- n 1))))))))))
==[60]==>

(<<*>> 5 (<<*>> 4 (<<*>> 3 (<<*>> 2 (<<*>> 1 1)))))
==[62]==>

120
```

Next, we try the usual iterative version:

```
SUB-EVAL==>
(define
 iter-factorial
 (lambda
  (n)
  (letrec
```

```
   ((iter
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
   (iter n 1))))

--------
iter-factorial has been defined

SUB-EVAL==>
(iter-factorial 5)
==[1]==>

((lambda
  (n)
  (letrec
   ((iter
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   (iter n 1)))
 5)
==[2]==>

(letrec
 ((n 5))
 (letrec
  ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
  (iter n 1)))
==[3]==>

(letrec
 ((n 5))
 (letrec
  ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))) n 1)))
==[4]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))) 5 1))
==[5]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec ((n 5) (result 1)) (if (<= n 0) result (iter (- n 1) (* n result)))))
==[6]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 5) (result 1))
  (if (<<<=>> 5 0) result (iter (- n 1) (* n result)))))
==[8]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
```



```
 (letrec ((n 5) (result 1)) (iter (- n 1) (* n result))))
==[10]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 5) (result 1))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (- n 1)
   (* n result))))
==[11]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 5) (result 1))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (<<->> 5 1)
   (* n result))))
==[13]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((result 1))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   4
   (<<*>> 5 result))))
==[16]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
  4
  (<<*>> 5 1)))
==[17]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec ((n 4) (result 5)) (if (<= n 0) result (iter (- n 1) (* n result)))))
==[19]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 4) (result 5))
  (if (<<<=>> 4 0) result (iter (- n 1) (* n result)))))
==[21]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec ((n 4) (result 5)) (iter (- n 1) (* n result))))
==[23]==>
```

```
(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 4) (result 5))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (- n 1)
   (* n result))))
==[24]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 4) (result 5))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (<<->> 4 1)
   (* n result))))
==[26]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((result 5))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   3
   (<<*>> 4 result))))
==[29]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
  3
  (<<*>> 4 5)))
==[30]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec ((n 3) (result 20)) (if (<= n 0) result (iter (- n 1) (* n result)))))
==[32]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 3) (result 20))
  (if (<<<=>> 3 0) result (iter (- n 1) (* n result)))))
==[34]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec ((n 3) (result 20)) (iter (- n 1) (* n result))))
==[36]==>

(letrec
```



```
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 3) (result 20))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (- n 1)
   (* n result))))
==[37]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 3) (result 20))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (<<->> 3 1)
   (* n result))))
==[39]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((result 20))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   2
   (<<*>> 3 result))))
==[42]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
  2
  (<<*>> 3 20)))
==[43]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec ((n 2) (result 60)) (if (<= n 0) result (iter (- n 1) (* n result)))))
==[45]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 2) (result 60))
  (if (<<<=>> 2 0) result (iter (- n 1) (* n result)))))
==[47]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec ((n 2) (result 60)) (iter (- n 1) (* n result))))
==[49]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
```

```
  ((n 2) (result 60))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (- n 1)
   (* n result))))
==[50]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 2) (result 60))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (<<->> 2 1)
   (* n result))))
==[52]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((result 60))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   1
   (<<*>> 2 result))))
==[55]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
  1
  (<<*>> 2 60)))
==[56]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 1) (result 120))
  (if (<= n 0) result (iter (- n 1) (* n result)))))
==[58]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 1) (result 120))
  (if (<<<=>> 1 0) result (iter (- n 1) (* n result)))))
==[60]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec ((n 1) (result 120)) (iter (- n 1) (* n result))))
==[62]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
```



```
  ((n 1) (result 120))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (- n 1)
   (* n result))))
==[63]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 1) (result 120))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   (<<->> 1 1)
   (* n result))))
==[65]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((result 120))
  ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
   0
   (<<*>> 1 result))))
==[68]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
  0
  (<<*>> 1 120)))
==[69]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 0) (result 120))
  (if (<= n 0) result (iter (- n 1) (* n result)))))
==[71]==>

(letrec
 ((iter (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((n 0) (result 120))
  (if (<<<=>> 0 0) result (iter (- n 1) (* n result)))))
==[73]==>

(letrec ((result 120)) result)
==[75]==>

120
```

One way to understand recursion is by "repeated unwinding" of a recursive definition. We begin by considering the body of the recursive definition as a function of the name being defined:

```
SUB-EVAL==>
(define
 factdef
 (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))

--------
factdef has been defined
```

Applying **factdef** to anything yields a procedure, call it **fact0**, which acts like recursive factorial on arguments less than or equal to 0. Applying **factdef** to **fact0** yields a procedure which acts like factorial on arguments less than or equal to 1. Here's an example of **fact3** in action:

```
SUB-EVAL==>
(define
 fact3
 (lambda (n) ((factdef (factdef (factdef (factdef (lambda (n) 'error))))) n)))

--------
fact3 has been defined

SUB-EVAL==>
(fact3 3)
==[1]==>

((lambda (n) ((factdef (factdef (factdef (factdef (lambda (n) 'error))))) n))
 3)
==[2]==>

(letrec
 ((n 3))
 ((factdef (factdef (factdef (factdef (lambda (n) 'error))))) n))
==[3]==>

(letrec
 ((n 3))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   (factdef (factdef (factdef (lambda (n) 'error)))))
  n))
==[4]==>

(letrec
 ((n 3))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    (factdef (factdef (lambda (n) 'error)))))
  n))
==[5]==>

(letrec
 ((n 3))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
```



```
    ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
     (factdef (lambda (n) 'error)))))
  n))
==[6]==>

(letrec
 ((n 3))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
     ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
      (lambda (n) 'error)))))
  n))
==[7]==>

(letrec
 ((n 3))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
     (letrec
      ((fact (lambda (n) 'error)))
      (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))))
  n))
==[8]==>

(letrec
 ((n 3))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    (letrec
     ((fact
       (letrec
        ((fact (lambda (n) 'error)))
        (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
     (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  n))
==[9]==>

(letrec
 ((n 3))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    (letrec
     ((fact#1 (lambda (n) 'error))
      (fact (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
     (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  n))
==[10]==>

(letrec
 ((n 3))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
```

```
(letrec
 ((fact
   (letrec
    ((fact#1 (lambda (n) 'error))
     (fact (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
    (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
 n))
==[11]==>

(letrec
 ((n 3))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
  (letrec
   ((fact#1 (lambda (n) 'error))
    (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
    (fact (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
   (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
 n))
==[12]==>

(letrec
 ((n 3))
 ((letrec
   ((fact
     (letrec
      ((fact#1 (lambda (n) 'error))
       (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
       (fact (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
      (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
   (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
  n))
==[13]==>

(letrec
 ((n 3))
 ((letrec
   ((fact#1 (lambda (n) 'error))
    (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
    (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
    (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
   (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
  n))
==[14]==>

((letrec
  ((fact#1 (lambda (n) 'error))
   (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
   (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
   (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
  (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
 3)
==[15]==>
```



```
(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 ((lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))) 3))
==[16]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 (letrec ((n 3)) (if (<= n 0) 1 (* n (fact (- n 1))))))
==[17]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 (letrec ((n 3)) (if (<<<=>> 3 0) 1 (* n (fact (- n 1))))))
==[19]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 (letrec ((n 3)) (* n (fact (- n 1)))))
==[21]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 (letrec ((n 3)) (<<*>> 3 (fact (- n 1)))))
==[23]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (letrec
  ((n 3))
  (<<*>> 3 ((lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1))))) (- n 1)))))
==[24]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
```

```
   (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
  (<<*>> 3 ((lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1))))) (<<->> 3 1))))
==[26]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (<<*>> 3 (letrec ((n 2)) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
==[28]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (<<*>> 3 (letrec ((n 2)) (if (<<<=>> 2 0) 1 (* n (fact#3 (- n 1)))))))
==[30]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (<<*>> 3 (letrec ((n 2)) (* n (fact#3 (- n 1))))))
==[32]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (<<*>> 3 (letrec ((n 2)) (<<*>> 2 (fact#3 (- n 1))))))
==[34]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>>
  3
  (letrec
   ((n 2))
   (<<*>> 2 ((lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))) (- n 1))))))
==[35]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>>
  3
  (<<*>> 2 ((lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))) (<<->> 2 1)))))
==[37]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
```



```
  (<<*>> 3 (<<*>> 2 (letrec ((n 1)) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))))
==[39]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>>
  3
  (<<*>> 2 (letrec ((n 1)) (if (<<<=>> 1 0) 1 (* n (fact#2 (- n 1))))))))
==[41]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>> 3 (<<*>> 2 (letrec ((n 1)) (* n (fact#2 (- n 1)))))))
==[43]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>> 3 (<<*>> 2 (letrec ((n 1)) (<<*>> 1 (fact#2 (- n 1)))))))
==[45]==>

(letrec
 ((fact#1 (lambda (n) 'error)))
 (<<*>>
  3
  (<<*>>
   2
   (letrec
    ((n 1))
    (<<*>> 1 ((lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))) (- n 1)))))))
==[46]==>

(letrec
 ((fact#1 (lambda (n) 'error)))
 (<<*>>
  3
  (<<*>>
   2
   (<<*>>
    1
    ((lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))) (<<->> 1 1))))))
==[48]==>

(letrec
 ((fact#1 (lambda (n) 'error)))
 (<<*>>
  3
  (<<*>> 2 (<<*>> 1 (letrec ((n 0)) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))))
==[50]==>

(letrec
```

```
((fact#1 (lambda (n) 'error)))
(<<*>>
 3
 (<<*>>
  2
  (<<*>> 1 (letrec ((n 0)) (if (<<<=>> 0 0) 1 (* n (fact#1 (- n 1)))))))))
==[52]==>

(<<*>> 3 (<<*>> 2 (<<*>> 1 1)))
==[54]==>

6
```

Here's what happens when fact3 is applied to too large an argument:

```
SUB-EVAL==>
(fact3 8)
==[1]==>

((lambda (n) ((factdef (factdef (factdef (factdef (lambda (n) 'error))))) n))
 8)
==[2]==>

(letrec
 ((n 8))
 ((factdef (factdef (factdef (factdef (lambda (n) 'error))))) n))
==[3]==>

(letrec
 ((n 8))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   (factdef (factdef (factdef (lambda (n) 'error)))))
  n))
==[4]==>

(letrec
 ((n 8))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    (factdef (factdef (lambda (n) 'error)))))
  n))
==[5]==>

(letrec
 ((n 8))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
     (factdef (lambda (n) 'error)))))
  n))
==[6]==>
```



```
(letrec
 ((n 8))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
     ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
      (lambda (n) 'error)))))
  n))
==[7]==>

(letrec
 ((n 8))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
     (letrec
      ((fact (lambda (n) 'error)))
      (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))))
  n))
==[8]==>

(letrec
 ((n 8))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    (letrec
     ((fact
       (letrec
        ((fact (lambda (n) 'error)))
        (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
     (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  n))
==[9]==>

(letrec
 ((n 8))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    (letrec
     ((fact#1 (lambda (n) 'error))
      (fact (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
     (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  n))
==[10]==>

(letrec
 ((n 8))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   (letrec
    ((fact
      (letrec
       ((fact#1 (lambda (n) 'error))
        (fact (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
```

```
      (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
     (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
   n))
==[11]==>

(letrec
 ((n 8))
 (((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   (letrec
    ((fact#1 (lambda (n) 'error))
     (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
     (fact (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
    (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
  n))
==[12]==>

(letrec
 ((n 8))
 ((letrec
   ((fact.
     (letrec
      ((fact#1 (lambda (n) 'error))
       (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
       (fact (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
      (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
   (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
  n))
==[13]==>

(letrec
 ((n 8))
 ((letrec
   ((fact#1 (lambda (n) 'error))
    (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
    (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
    (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
   (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
  n))
==[14]==>

((letrec
  ((fact#1 (lambda (n) 'error))
   (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
   (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
   (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
  (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
 8)
==[15]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
```



```
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 ((lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))) 8))
==[16]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 (letrec ((n 8)) (if (<= n 0) 1 (* n (fact (- n 1))))))
==[17]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 (letrec ((n 8)) (if (<<<=>> 8 0) 1 (* n (fact (- n 1))))))
==[19]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 (letrec ((n 8)) (* n (fact (- n 1)))))
==[21]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))
  (fact (lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
 (letrec ((n 8)) (<<*>> 8 (fact (- n 1)))))
==[23]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (letrec
  ((n 8))
  (<<*>> 8 ((lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1))))) (- n 1)))))
==[24]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (<<*>> 8 ((lambda (n) (if (<= n 0) 1 (* n (fact#3 (- n 1))))) (<<->> 8 1))))
==[26]==>

(letrec
```

```
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (<<*>> 8 (letrec ((n 7)) (if (<= n 0) 1 (* n (fact#3 (- n 1)))))))
==[28]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (<<*>> 8 (letrec ((n 7)) (if (<<<=>> 7 0) 1 (* n (fact#3 (- n 1)))))))
==[30]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (<<*>> 8 (letrec ((n 7)) (* n (fact#3 (- n 1))))))
==[32]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))))
  (fact#3 (lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1)))))))
 (<<*>> 8 (letrec ((n 7)) (<<*>> 7 (fact#3 (- n 1))))))
==[34]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>>
  8
  (letrec
   ((n 7))
   (<<*>> 7 ((lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))) (- n 1))))))
==[35]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>>
  8
  (<<*>> 7 ((lambda (n) (if (<= n 0) 1 (* n (fact#2 (- n 1))))) (<<->> 7 1)))))
==[37]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>> 8 (<<*>> 7 (letrec ((n 6)) (if (<= n 0) 1 (* n (fact#2 (- n 1))))))))
==[39]==>

(letrec
 ((fact#1 (lambda (n) 'error))
```



```
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>>
  8
  (<<*>> 7 (letrec ((n 6)) (if (<<<=>> 6 0) 1 (* n (fact#2 (- n 1))))))))
==[41]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>> 8 (<<*>> 7 (letrec ((n 6)) (* n (fact#2 (- n 1)))))))
==[43]==>

(letrec
 ((fact#1 (lambda (n) 'error))
  (fact#2 (lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))
 (<<*>> 8 (<<*>> 7 (letrec ((n 6)) (<<*>> 6 (fact#2 (- n 1)))))))
==[45]==>

(letrec
 ((fact#1 (lambda (n) 'error)))
 (<<*>>
  8
  (<<*>>
   7
   (letrec
    ((n 6))
    (<<*>> 6 ((lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))) (- n 1)))))))
==[46]==>

(letrec
 ((fact#1 (lambda (n) 'error)))
 (<<*>>
  8
  (<<*>>
   7
   (<<*>>
    6
    ((lambda (n) (if (<= n 0) 1 (* n (fact#1 (- n 1))))) (<<->> 6 1))))))
==[48]==>

(letrec
 ((fact#1 (lambda (n) 'error)))
 (<<*>>
  8
  (<<*>> 7 (<<*>> 6 (letrec ((n 5)) (if (<= n 0) 1 (* n (fact#1 (- n 1)))))))))
==[50]==>

(letrec
 ((fact#1 (lambda (n) 'error)))
 (<<*>>
  8
  (<<*>>
   7
```

```
   (<<*>> 6 (letrec ((n 5)) (if (<<<=>> 5 0) 1 (* n (fact#1 (- n 1)))))))))
==[52]==>

(letrec
 ((fact#1 (lambda (n) 'error)))
 (<<*>> 8 (<<*>> 7 (<<*>> 6 (letrec ((n 5)) (* n (fact#1 (- n 1))))))))
==[54]==>

(letrec
 ((fact#1 (lambda (n) 'error)))
 (<<*>> 8 (<<*>> 7 (<<*>> 6 (letrec ((n 5)) (<<*>> 5 (fact#1 (- n 1))))))))
==[56]==>

(<<*>>
 8
 (<<*>> 7 (<<*>> 6 (letrec ((n 5)) (<<*>> 5 ((lambda (n) 'error) (- n 1)))))))
==[57]==>

(<<*>> 8 (<<*>> 7 (<<*>> 6 (<<*>> 5 ((lambda (n) 'error) (<<->> 5 1))))))
==[59]==>

(<<*>> 8 (<<*>> 7 (<<*>> 6 (<<*>> 5 'error))))

;The object error, passed as the second argument to integer-multiply,
;is not the correct type.
```

We can define an procedure which carries out the unwinding as many times as necessary. This procedure is called a *fixed point* operator; we shall discuss why later in lecture. For historical reasons the fixed point operator is also called the Y operator:

```
SUB-EVAL==>
(define
 y
 (lambda
  (f)
  ((lambda (x) (f (lambda (z) ((x x) z))))
   (lambda (x) (f (lambda (z) ((x x) z)))))))

--------
y has been defined
```

Applying the fixed point operator to the body of the recursive definition of factorial will yield a procedure `y-fact` which computes fatorials:

```
SUB-EVAL==>
(define y-fact (y factdef))
==[1]==>

(define
 y-fact
 ((lambda
   (f)
```



```
   ((lambda (x) (f (lambda (z) ((x x) z))))
    (lambda (x) (f (lambda (z) ((x x) z))))))
  factdef))
==[2]==>

(define
 y-fact
 ((lambda
   (f)
   ((lambda (x) (f (lambda (z) ((x x) z))))
    (lambda (x) (f (lambda (z) ((x x) z))))))
  (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
==[3]==>

(define
 y-fact
 (letrec
  ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  ((lambda (x) (f (lambda (z) ((x x) z))))
   (lambda (x) (f (lambda (z) ((x x) z)))))))
==[4]==>

(define
 y-fact
 (letrec
  ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (f (lambda (z) ((x x) z))))))
==[5]==>

(define
 y-fact
 (letrec
  ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    (lambda (z) ((x x) z))))))
==[6]==>

(define
 y-fact
 (letrec
  ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (letrec
    ((fact (lambda (z) ((x x) z))))
    (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))))

--------
y-fact has been defined
```

Now we can test it:

```
SUB-EVAL==>
(y-fact 5)
==[1]==>

((letrec
  ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (letrec
    ((fact (lambda (z) ((x x) z))))
    (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 5)
==[2]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 ((letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (letrec
    ((fact (lambda (z) ((x x) z))))
    (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
  5))
==[3]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (letrec
  ((x (lambda (x) (f (lambda (z) ((x x) z))))))
  ((letrec
    ((fact (lambda (z) ((x x) z))))
    (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
   5)))
==[4]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (letrec
  ((x (lambda (x) (f (lambda (z) ((x x) z))))))
  (letrec
   ((fact (lambda (z) ((x x) z))))
   ((lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))) 5))))
==[5]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (letrec
  ((x (lambda (x) (f (lambda (z) ((x x) z))))))
  (letrec
   ((fact (lambda (z) ((x x) z))))
```



```
   (letrec ((n 5)) (if (<= n 0) 1 (* n (fact (- n 1))))))))
==[6]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (letrec
  ((x (lambda (x) (f (lambda (z) ((x x) z))))))
  (letrec
   ((fact (lambda (z) ((x x) z))))
   (letrec ((n 5)) (if (<<<=>> 5 0) 1 (* n (fact (- n 1))))))))
==[8]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (letrec
  ((x (lambda (x) (f (lambda (z) ((x x) z))))))
  (letrec
   ((fact (lambda (z) ((x x) z))))
   (letrec ((n 5)) (* n (fact (- n 1)))))))
==[10]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (letrec
  ((x (lambda (x) (f (lambda (z) ((x x) z))))))
  (letrec
   ((fact (lambda (z) ((x x) z))))
   (letrec ((n 5)) (<<*>> 5 (fact (- n 1)))))))
==[12]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (letrec
  ((x (lambda (x) (f (lambda (z) ((x x) z))))))
  (letrec ((n 5)) (<<*>> 5 ((lambda (z) ((x x) z)) (- n 1))))))
==[13]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (letrec
  ((x (lambda (x) (f (lambda (z) ((x x) z))))))
  (<<*>> 5 ((lambda (z) ((x x) z)) (<<->> 5 1)))))
==[15]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (letrec
  ((x (lambda (x) (f (lambda (z) ((x x) z))))))
  (<<*>> 5 (letrec ((z 4)) ((x x) z)))))
==[17]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
```

```
(letrec
 ((x (lambda (x) (f (lambda (z) ((x x) z))))))
 (<<*>> 5 (letrec ((z 4)) (((lambda (x) (f (lambda (z) ((x x) z)))) x) z)))))
==[18]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((z 4))
   (((lambda (x) (f (lambda (z) ((x x) z))))
     (lambda (x) (f (lambda (z) ((x x) z)))))
    z))))
==[19]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((z 4))
   ((letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (f (lambda (z) ((x x) z))))
    z))))
==[20]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((z 4))
   ((letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
      (lambda (z) ((x x) z))))
    z))))
==[21]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((z 4))
   ((letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (letrec
      ((fact (lambda (z) ((x x) z))))
      (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
    z))))
```



```
==[22]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  ((letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (letrec
     ((fact (lambda (z) ((x x) z))))
     (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
   4)))
==[23]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   ((letrec
     ((fact (lambda (z) ((x x) z))))
     (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
    4))))
==[24]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (letrec
    ((fact (lambda (z) ((x x) z))))
    ((lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))) 4)))))
==[25]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (letrec
    ((fact (lambda (z) ((x x) z))))
    (letrec ((n 4)) (if (<= n 0) 1 (* n (fact (- n 1))))))))))
==[26]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
```

```
   ((x (lambda (x) (f (lambda (z) ((x x) z)))))
   (letrec
    ((fact (lambda (z) ((x x) z))))
    (letrec ((n 4)) (if (<<<=>> 4 0) 1 (* n (fact (- n 1)))))))))
==[28]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (letrec
    ((fact (lambda (z) ((x x) z))))
    (letrec ((n 4)) (* n (fact (- n 1))))))))
==[30]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (letrec
    ((fact (lambda (z) ((x x) z))))
    (letrec ((n 4)) (<<*>> 4 (fact (- n 1))))))))
==[32]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (letrec ((n 4)) (<<*>> 4 ((lambda (z) ((x x) z)) (- n 1)))))))
==[33]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (<<*>> 4 ((lambda (z) ((x x) z)) (<<->> 4 1))))))
==[35]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (<<*>> 4 (letrec ((z 3)) ((x x) z))))))
```



```
==[37]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (letrec
   ((x (lambda (x) (f (lambda (z) ((x x) z))))))
   (<<*>>
    4
    (letrec ((z 3)) (((lambda (x) (f (lambda (z) ((x x) z)))) x) z))))))
==[38]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((z 3))
    (((lambda (x) (f (lambda (z) ((x x) z))))
      (lambda (x) (f (lambda (z) ((x x) z)))))
     z)))))
==[39]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((z 3))
    ((letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (f (lambda (z) ((x x) z))))
     z)))))
==[40]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((z 3))
    ((letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
       (lambda (z) ((x x) z))))
     z)))))
```

```
==[41]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((z 3))
    ((letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (letrec
       ((fact (lambda (z) ((x x) z))))
       (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
     z)))))
==[42]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   ((letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (letrec
      ((fact (lambda (z) ((x x) z))))
      (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
    3))))
==[43]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    ((letrec
      ((fact (lambda (z) ((x x) z))))
      (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
     3)))))
==[44]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
```



```
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (letrec
     ((fact (lambda (z) ((x x) z))))
     ((lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))) 3))))))
==[45]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (letrec
     ((fact (lambda (z) ((x x) z))))
     (letrec ((n 3)) (if (<= n 0) 1 (* n (fact (- n 1))))))))))
==[46]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (letrec
     ((fact (lambda (z) ((x x) z))))
     (letrec ((n 3)) (if (<<<=>> 3 0) 1 (* n (fact (- n 1))))))))))
==[48]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (letrec
     ((fact (lambda (z) ((x x) z))))
     (letrec ((n 3)) (* n (fact (- n 1)))))))))
==[50]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
```

```
   (letrec
    ((fact (lambda (z) ((x x) z))))
    (letrec ((n 3)) (<<*>> 3 (fact (- n 1)))))))))
==[52]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (letrec ((n 3)) (<<*>> 3 ((lambda (z) ((x x) z)) (- n 1))))))))
==[53]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (<<*>> 3 ((lambda (z) ((x x) z)) (<<->> 3 1)))))))
==[55]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (<<*>> 3 (letrec ((z 2)) ((x x) z)))))))
==[57]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (<<*>>
     3
     (letrec ((z 2)) (((lambda (x) (f (lambda (z) ((x x) z)))) x) z)))))))
==[58]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
```



```
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((z 2))
     (((lambda (x) (f (lambda (z) ((x x) z))))
       (lambda (x) (f (lambda (z) ((x x) z)))))
      z))))))
==[59]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((z 2))
     ((letrec
       ((x (lambda (x) (f (lambda (z) ((x x) z))))))
       (f (lambda (z) ((x x) z))))
      z))))))
==[60]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((z 2))
     ((letrec
       ((x (lambda (x) (f (lambda (z) ((x x) z))))))
       ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
        (lambda (z) ((x x) z))))
      z))))))
==[61]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
```

```
    3
    (letrec
     ((z 2))
     ((letrec
       ((x (lambda (x) (f (lambda (z) ((x x) z))))))
       (letrec
        ((fact (lambda (z) ((x x) z))))
        (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
      z))))))
==[62]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    ((letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (letrec
       ((fact (lambda (z) ((x x) z))))
       (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
     2)))))
==[63]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     ((letrec
       ((fact (lambda (z) ((x x) z))))
       (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
      2))))))
==[64]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
```



```
     (letrec
      ((fact (lambda (z) ((x x) z))))
      ((lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))) 2)))))))
==[65]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (letrec
      ((fact (lambda (z) ((x x) z))))
      (letrec ((n 2)) (if (<= n 0) 1 (* n (fact (- n 1))))))))))))
==[66]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (letrec
      ((fact (lambda (z) ((x x) z))))
      (letrec ((n 2)) (if (<<<=>> 2 0) 1 (* n (fact (- n 1))))))))))))
==[68]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (letrec
      ((fact (lambda (z) ((x x) z))))
      (letrec ((n 2)) (* n (fact (- n 1)))))))))))
==[70]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
```

```
 5
 (<<*>>
  4
  (<<*>>
   3
   (letrec
    ((x (lambda (x) (f (lambda (z) ((x x) z))))))
    (letrec
     ((fact (lambda (z) ((x x) z))))
     (letrec ((n 2)) (<<*>> 2 (fact (- n 1))))))))))
==[72]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (letrec ((n 2)) (<<*>> 2 ((lambda (z) ((x x) z)) (- n 1)))))))))
==[73]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (<<*>> 2 ((lambda (z) ((x x) z)) (<<->> 2 1))))))))
==[75]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (<<*>> 2 (letrec ((z 1)) ((x x) z))))))))
==[77]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
```



```
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (letrec
     ((x (lambda (x) (f (lambda (z) ((x x) z))))))
     (<<*>>
      2
      (letrec ((z 1)) (((lambda (x) (f (lambda (z) ((x x) z)))) x) z))))))))
==[78]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((z 1))
      (((lambda (x) (f (lambda (z) ((x x) z))))
        (lambda (x) (f (lambda (z) ((x x) z)))))
       z)))))))
==[79]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((z 1))
      ((letrec
        ((x (lambda (x) (f (lambda (z) ((x x) z))))))
        (f (lambda (z) ((x x) z))))
       z)))))))
==[80]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
```

```
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((z 1))
      ((letrec
        ((x (lambda (x) (f (lambda (z) ((x x) z))))))
        ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
         (lambda (z) ((x x) z))))
       z)))))))
==[81]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((z 1))
      ((letrec
        ((x (lambda (x) (f (lambda (z) ((x x) z))))))
        (letrec
         ((fact (lambda (z) ((x x) z))))
         (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
       z)))))))
==[82]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     ((letrec
       ((x (lambda (x) (f (lambda (z) ((x x) z))))))
       (letrec
        ((fact (lambda (z) ((x x) z))))
        (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
      1))))))
==[83]==>

(letrec
```



```
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      ((letrec
        ((fact (lambda (z) ((x x) z))))
        (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
       1)))))))
==[84]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (letrec
       ((fact (lambda (z) ((x x) z))))
       ((lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))) 1))))))))
==[85]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (letrec
       ((fact (lambda (z) ((x x) z))))
       (letrec ((n 1)) (if (<= n 0) 1 (* n (fact (- n 1))))))))))))
==[86]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
```

```
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (letrec
       ((fact (lambda (z) ((x x) z))))
       (letrec ((n 1)) (if (<<<=>> 1 0) 1 (* n (fact (- n 1))))))))))))
==[88]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (letrec
       ((fact (lambda (z) ((x x) z))))
       (letrec ((n 1)) (* n (fact (- n 1))))))))))))
==[90]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (letrec
       ((fact (lambda (z) ((x x) z))))
       (letrec ((n 1)) (<<*>> 1 (fact (- n 1))))))))))))
==[92]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
```



```
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (letrec ((n 1)) (<<*>> 1 ((lambda (z) ((x x) z)) (- n 1))))))))))
==[93]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (<<*>> 1 ((lambda (z) ((x x) z)) (<<->> 1 1)))))))))
==[95]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
      (<<*>> 1 (letrec ((z 0)) ((x x) z)))))))))
==[97]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (letrec
      ((x (lambda (x) (f (lambda (z) ((x x) z))))))
```

```
      (<<*>>
       1
       (letrec ((z 0)) (((lambda (x) (f (lambda (z) ((x x) z)))) x) z)))))))))
==[98]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (<<*>>
      1
      (letrec
       ((z 0))
       (((lambda (x) (f (lambda (z) ((x x) z))))
         (lambda (x) (f (lambda (z) ((x x) z)))))
        z))))))))
==[99]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (<<*>>
      1
      (letrec
       ((z 0))
       ((letrec
         ((x (lambda (x) (f (lambda (z) ((x x) z))))))
         (f (lambda (z) ((x x) z))))
        z))))))))
==[100]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
```



```
     2
     (<<*>>
      1
      (letrec
       ((z 0))
       ((letrec
         ((x (lambda (x) (f (lambda (z) ((x x) z))))))
         ((lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
          (lambda (z) ((x x) z))))
        z))))))))
==[101]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (<<*>>
      1
      (letrec
       ((z 0))
       ((letrec
         ((x (lambda (x) (f (lambda (z) ((x x) z))))))
         (letrec
          ((fact (lambda (z) ((x x) z))))
          (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
        z))))))))
==[102]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (<<*>>
      1
      ((letrec
        ((x (lambda (x) (f (lambda (z) ((x x) z))))))
        (letrec
         ((fact (lambda (z) ((x x) z))))
         (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1)))))))
       0)))))))
==[103]==>
```

```
(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (<<*>>
      1
      (letrec
       ((x (lambda (x) (f (lambda (z) ((x x) z))))))
       ((letrec
         ((fact (lambda (z) ((x x) z))))
         (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))
        0))))))))
==[104]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (<<*>>
      1
      (letrec
       ((x (lambda (x) (f (lambda (z) ((x x) z))))))
       (letrec
        ((fact (lambda (z) ((x x) z))))
        ((lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))) 0)))))))))
==[105]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (<<*>>
      1
      (letrec
```



```
       ((x (lambda (x) (f (lambda (z) ((x x) z))))))
       (letrec
        ((fact (lambda (z) ((x x) z))))
        (letrec ((n 0)) (if (<= n 0) 1 (* n (fact (- n 1))))))))))))))
==[106]==>

(letrec
 ((f (lambda (fact) (lambda (n) (if (<= n 0) 1 (* n (fact (- n 1))))))))
 (<<*>>
  5
  (<<*>>
   4
   (<<*>>
    3
    (<<*>>
     2
     (<<*>>
      1
      (letrec
       ((x (lambda (x) (f (lambda (z) ((x x) z))))))
       (letrec
        ((fact (lambda (z) ((x x) z))))
        (letrec ((n 0)) (if (<<<=>> 0 0) 1 (* n (fact (- n 1))))))))))))))
==[108]==>

(<<*>> 5 (<<*>> 4 (<<*>> 3 (<<*>> 2 (<<*>> 1 1)))))
==[110]==>

120
```

A slightly different fixed point operator is needed for recursive definitions with two formals:

```
SUB-EVAL==>
(define
 y2
 (lambda
  (f)
  ((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))
   (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))))))

--------
y2 has been defined
```

With $Y2$ we can get the iterative factorial by unwinding:

```
SUB-EVAL==>
(define
 y-iter-fact
 (lambda
  (n)
  ((y2
```

```
    (lambda
     (iter)
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
   n
   1)))

--------
y-iter-fact has been defined
```

And we can test it:

```
SUB-EVAL==>
(y-iter-fact 5)
==[1]==>

((lambda
  (n)
  ((y2
    (lambda
     (iter)
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
   n
   1))
 5)
==[2]==>

(letrec
 ((n 5))
 ((y2
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
  n
  1))
==[3]==>

(letrec
 ((n 5))
 (((lambda
    (f)
    ((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
  n
  1))
==[4]==>

(letrec
 ((n 5))
 ((letrec
```



```
   ((f
     (lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   ((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  n
  1))
==[5]==>

(letrec
 ((n 5))
 ((letrec
   ((f
     (lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   (letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (f (lambda (z1 z2) ((x x) z1 z2)))))
  n
  1))
==[6]==>

(letrec
 ((n 5))
 ((letrec
   ((f
     (lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   (letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    ((lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
     (lambda (z1 z2) ((x x) z1 z2)))))
  n
  1))
==[7]==>

(letrec
 ((n 5))
 ((letrec
   ((f
     (lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   (letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
```

```
  n
  1))
==[8]==>

((letrec
  ((f
    (lambda
     (iter)
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
  (letrec
   ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
   (letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 5
 1)
==[9]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 ((letrec
   ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
   (letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
  5
  1))
==[10]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
   5
   1)))
==[11]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
```



```
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
    5
    1))))
==[12]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 5) (result 1))
    (if (<= n 0) result (iter (- n 1) (* n result)))))))
==[13]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 5) (result 1))
    (if (<<<=>> 5 0) result (iter (- n 1) (* n result)))))))
==[15]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec ((n 5) (result 1)) (iter (- n 1) (* n result))))))
==[17]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
```

```
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 5) (result 1))
   ((lambda (z1 z2) ((x x) z1 z2)) (- n 1) (* n result)))))
==[18]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 5) (result 1))
   ((lambda (z1 z2) ((x x) z1 z2)) (<<->> 5 1) (* n result)))))
==[20]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((result 1)) ((lambda (z1 z2) ((x x) z1 z2)) 4 (<<*>> 5 result)))))
==[23]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((lambda (z1 z2) ((x x) z1 z2)) 4 (<<*>> 5 1))))
==[24]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((z1 4) (z2 5)) ((x x) z1 z2))))
==[26]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
```



```
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((z1 4) (z2 5))
   (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))) x) z1 z2))))
==[27]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 4) (z2 5))
  (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))))
   z1
   z2)))
==[28]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 4) (z2 5))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (f (lambda (z1 z2) ((x x) z1 z2))))
   z1
   z2)))
==[29]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 4) (z2 5))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    ((lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
     (lambda (z1 z2) ((x x) z1 z2))))
   z1
   z2)))
==[30]==>

(letrec
 ((f
```

```
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 4) (z2 5))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   z1
   z2)))
==[31]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z2 5))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   4
   z2)))
==[32]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 ((letrec
   ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
   (letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
  4
  5))
==[33]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
```



```
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
   4
   5)))
==[34]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
    4
    5))))
==[35]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 4) (result 5))
    (if (<= n 0) result (iter (- n 1) (* n result)))))))
==[36]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 4) (result 5))
    (if (<<<=>> 4 0) result (iter (- n 1) (* n result)))))))
==[38]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
```

```
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec ((n 4) (result 5)) (iter (- n 1) (* n result))))))
==[40]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 4) (result 5))
   ((lambda (z1 z2) ((x x) z1 z2)) (- n 1) (* n result)))))
==[41]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 4) (result 5))
   ((lambda (z1 z2) ((x x) z1 z2)) (<<->> 4 1) (* n result)))))
==[43]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((result 5)) ((lambda (z1 z2) ((x x) z1 z2)) 3 (<<*>> 4 result)))))
==[46]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((lambda (z1 z2) ((x x) z1 z2)) 3 (<<*>> 4 5))))
==[47]==>

(letrec
 ((f
   (lambda
```



```
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((z1 3) (z2 20)) ((x x) z1 z2))))
==[49]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((z1 3) (z2 20))
   (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))) x) z1 z2))))
==[50]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 3) (z2 20))
  (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))))
   z1
   z2)))
==[51]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 3) (z2 20))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (f (lambda (z1 z2) ((x x) z1 z2))))
   z1
   z2)))
==[52]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 3) (z2 20))
```

```
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    ((lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
     (lambda (z1 z2) ((x x) z1 z2))))
   z1
   z2)))
==[53]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 3) (z2 20))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   z1
   z2)))
==[54]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z2 20))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   3
   z2)))
==[55]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 ((letrec
   ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
   (letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
  3
```



```
  20))
==[56]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
   3
   20)))
==[57]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
    3
    20))))
==[58]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 3) (result 20))
    (if (<= n 0) result (iter (- n 1) (* n result)))))))
==[59]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
```

```
  ((iter (lambda (z1 z2) ((x x) z1 z2)))
  (letrec
   ((n 3) (result 20))
   (if (<<<=>> 3 0) result (iter (- n 1) (* n result))))))
==[61]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec ((n 3) (result 20)) (iter (- n 1) (* n result))))))
==[63]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 3) (result 20))
   ((lambda (z1 z2) ((x x) z1 z2)) (- n 1) (* n result)))))
==[64]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 3) (result 20))
   ((lambda (z1 z2) ((x x) z1 z2)) (<<->> 3 1) (* n result)))))
==[66]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((result 20)) ((lambda (z1 z2) ((x x) z1 z2)) 2 (<<*>> 3 result)))))
==[69]==>

(letrec
```



```
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((lambda (z1 z2) ((x x) z1 z2)) 2 (<<*>> 3 20))))
==[70]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((z1 2) (z2 60)) ((x x) z1 z2))))
==[72]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((z1 2) (z2 60))
   (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))) x) z1 z2))))
==[73]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((z1 2) (z2 60))
  (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))))
   z1
   z2)))
==[74]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
 (letrec
  ((z1 2) (z2 60))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
```

```
   (f (lambda (z1 z2) ((x x) z1 z2))))
  z1
  z2)))
==[75]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 2) (z2 60))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    ((lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
     (lambda (z1 z2) ((x x) z1 z2))))
   z1
   z2)))
==[76]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 2) (z2 60))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   z1
   z2)))
==[77]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z2 60))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   2
   z2)))
==[78]==>
```



```
(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 ((letrec
   ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
   (letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
  2
  60))
==[79]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
   2
   60)))
==[80]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
    2
    60))))
==[81]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
```

```
    ((n 2) (result 60))
    (if (<= n 0) result (iter (- n 1) (* n result)))))))
==[82]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 2) (result 60))
    (if (<<<=>> 2 0) result (iter (- n 1) (* n result)))))))
==[84]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec ((n 2) (result 60)) (iter (- n 1) (* n result))))))
==[86]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 2) (result 60))
   ((lambda (z1 z2) ((x x) z1 z2)) (- n 1) (* n result)))))
==[87]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 2) (result 60))
   ((lambda (z1 z2) ((x x) z1 z2)) (<<->> 2 1) (* n result)))))
==[89]==>
```



```
(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((result 60)) ((lambda (z1 z2) ((x x) z1 z2)) 1 (<<*>> 2 result)))))
==[92]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((lambda (z1 z2) ((x x) z1 z2)) 1 (<<*>> 2 60))))
==[93]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((z1 1) (z2 120)) ((x x) z1 z2))))
==[95]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((z1 1) (z2 120))
   (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))) x) z1 z2))))
==[96]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 1) (z2 120))
  (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))))
   z1
```

```
   z2)))
==[97]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 1) (z2 120))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (f (lambda (z1 z2) ((x x) z1 z2))))
   z1
   z2)))
==[98]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 1) (z2 120))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    ((lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
     (lambda (z1 z2) ((x x) z1 z2))))
   z1
   z2)))
==[99]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 1) (z2 120))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   z1
   z2)))
==[100]==>

(letrec
 ((f
   (lambda
```



```
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z2 120))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   1
   z2)))
==[101]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 ((letrec
   ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
   (letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
  1
  120))
==[102]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
   1
   120)))
==[103]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
    1
    120))))
```

```
==[104]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 1) (result 120))
    (if (<= n 0) result (iter (- n 1) (* n result)))))))
==[105]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 1) (result 120))
    (if (<<<=>> 1 0) result (iter (- n 1) (* n result)))))))
==[107]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec ((n 1) (result 120)) (iter (- n 1) (* n result))))))
==[109]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 1) (result 120))
   ((lambda (z1 z2) ((x x) z1 z2)) (- n 1) (* n result)))))
==[110]==>
```



```
(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((n 1) (result 120))
   ((lambda (z1 z2) ((x x) z1 z2)) (<<->> 1 1) (* n result)))))
==[112]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((result 120)) ((lambda (z1 z2) ((x x) z1 z2)) 0 (<<*>> 1 result)))))
==[115]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((lambda (z1 z2) ((x x) z1 z2)) 0 (<<*>> 1 120))))
==[116]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec ((z1 0) (z2 120)) ((x x) z1 z2))))
==[118]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((z1 0) (z2 120))
   (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))) x) z1 z2))))
```

```
==[119]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 0) (z2 120))
  (((lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2)))))
   z1
   z2)))
==[120]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 0) (z2 120))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (f (lambda (z1 z2) ((x x) z1 z2))))
   z1
   z2)))
==[121]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 0) (z2 120))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    ((lambda
      (iter)
      (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
     (lambda (z1 z2) ((x x) z1 z2))))
   z1
   z2)))
==[122]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z1 0) (z2 120))
```



```
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   z1
   z2)))
==[123]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((z2 120))
  ((letrec
    ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
    (letrec
     ((iter (lambda (z1 z2) ((x x) z1 z2))))
     (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
   0
   z2)))
==[124]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 ((letrec
   ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
   (letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))))
  0
  120))
==[125]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  ((letrec
    ((iter (lambda (z1 z2) ((x x) z1 z2))))
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))
   0
   120)))
==[126]==>
```

```
(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   ((lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result))))
    0
    120))))
==[127]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 0) (result 120))
    (if (<= n 0) result (iter (- n 1) (* n result)))))))
==[128]==>

(letrec
 ((f
   (lambda
    (iter)
    (lambda (n result) (if (<= n 0) result (iter (- n 1) (* n result)))))))
 (letrec
  ((x (lambda (x) (f (lambda (z1 z2) ((x x) z1 z2))))))
  (letrec
   ((iter (lambda (z1 z2) ((x x) z1 z2))))
   (letrec
    ((n 0) (result 120))
    (if (<<<=>> 0 0) result (iter (- n 1) (* n result)))))))
==[130]==>

(letrec ((result 120)) result)
==[132]==>

120

SUB-EVAL==>
```

Substitution Model: Formal Definitions

by Albert R. Meyer, Justin Liu, and Brian So

1 The Functional Kernel of Scheme

1.1 Syntax

We follow the grammatical conventions of the Revised^4 Scheme Report, using x, y, z to denote variables, and M, N, B to denote expressions. We use * to indicate zero or more occurrences of a phrase type and + for one or more occurrences.

⟨keyword⟩	::=	⟨binding-keyword⟩ | ⟨nonbinding-keyword⟩	
⟨nonbinding-keyword⟩	::=	`if`	
⟨binding-keyword⟩	::=	`lambda` | `letrec`	
⟨exp⟩	::=	`(if` ⟨exp⟩ ⟨exp⟩ ⟨exp⟩ `)`	
		| `(`⟨exp⟩$^+$`)`	
		| `(lambda (`⟨formals⟩`)` ⟨exp⟩`)`	
		| `(letrec (`⟨bindings⟩`)` ⟨exp⟩`)`	
		| ⟨constant⟩	
		| ⟨var⟩	
⟨bindings⟩	::=	`(`⟨var⟩ ⟨exp⟩`)`*	(all ⟨var⟩'s must be distinct)
⟨formals⟩	::=	⟨var⟩*	(all ⟨var⟩'s must be distinct)
⟨constant⟩	::=	⟨numeral⟩ | ⟨boolean⟩ | ⟨scheme-constant⟩ | ⟨system-constant⟩	
⟨numeral⟩	::=	`0` | `-1` | `3.14159` | ...	
⟨boolean⟩	::=	`#t` | `#f`	
⟨scheme-constant⟩	::=	⟨built-in⟩ | ⟨rule-defined⟩	
⟨system-constant⟩	::=	identifiers of the form <<···>> that are not ⟨scheme-constant⟩'s	
⟨var⟩	::=	identifiers other than ⟨constant⟩'s or ⟨keyword⟩'s	

The ⟨scheme-constant⟩'s correspond to familiar Scheme procedures and are listed in Table 1. The constants in the first group of ⟨built-in⟩'s in Table 1 correspond to basic operations on numerals as specified in the Revised^4 Scheme Report. To eliminate some uninteresting steps in evaluations, we also introduce a second group of ⟨built-in⟩'s. These could have been omitted, since their behavior is definable by simple ⟨exp⟩'s using the other constants. The ⟨system-constant⟩'s correspond to other procedures which may be added to the system by loading external code.

1.2 Extended Syntax

To shorten the description of the Substitution Model, we have left out of the kernel syntax some convenient and familiar Scheme constructs. For example, we want the Substitution Model to handle additional ⟨exp⟩'s specified by the following extension of the grammar.

⟨built-in⟩ ::= <<+>> | <<->> | <<*>> | <</>> | << = >> | << < >> | << > >> | <<<=>>
| <<>=>> | <<**expt**>> | <<**round**>> | <<**gcd**>> | <<**max**>> | <<**min**>>
| <<**exp**>> | <<**log**>> | <<**sin**>> | <<**log**>> | <<**tan**>> | <<**asin**>>
| <<**acos**>> | <<**atan**>> | <<**quotient**>>
⟨rule-defined⟩ ::= <<**boolean?**>> | <<**number?**>> | <<**procedure?**>>
⟨built-in⟩ ::= ... | <<**zero?**>> | <<**positive?**>> | <<**negative?**>> | <<**odd?**>> | <<**even?**>>
| <<**abs**>> | <<**1+**>> | <<**-1+**>> | <<**sqrt**>> | <<**not**>>

Table 1: Scheme Procedure Constants

⟨nonbinding-keyword⟩ ::= ... | **and** | **or** | **begin** | **cond**
⟨binding-keyword⟩ ::= ... | **let** | **define**
⟨exp⟩ ::= ... | (⟨nonbinding-keyword⟩ ⟨exp⟩*) (except for cond)
| (**cond** (⟨exp⟩ ⟨exp⟩)* (**else** ⟨exp⟩))
| (**let** (⟨bindings⟩) ⟨exp⟩)
| ⟨define⟩* ⟨exp⟩ (all defined ⟨var⟩'s must be distinct)
⟨define⟩ ::= (**define** ⟨var⟩ ⟨exp⟩) | (**define** (⟨var⟩ ⟨formals⟩) ⟨exp⟩)
⟨built-in⟩ ::= ... | <<**display**>> | <<**newline**>> | <<**error**>>

(For simplicity, this grammar does not show the constraints on numbers of subforms a nonbinding keyword may have, i.e., `if` may have only three or two subforms and `begin` must have at least one subform.)

It's easy to extend the Substitution Model to handle these extensions directly, and indeed our implementation does so (see the Appendix). However, another easy way to understand these extensions is by thinking of them as abbreviations for kernel expressions. For example,

$$\texttt{(let ((}x_1\ N_1\texttt{)}\cdots\texttt{)}\ B\texttt{)}$$

can be understood as an abbreviation for

$$\texttt{((lambda (}x_1\cdots\texttt{)}\ B\texttt{)}\ N_1\cdots\texttt{)}.$$

Similarly,

$$\texttt{(define}\ x_1\ N_1\texttt{)}\ldots\texttt{(define}\ x_n\ N_n\texttt{)}\ M$$

can be understood as an abbreviation for

$$\texttt{(letrec ((}x_1\ N_1\texttt{)}\ldots\texttt{(}x_n\ N_n\texttt{))}\ M\texttt{)},$$

and

$$\texttt{(cond (}M_1\ N_1\texttt{) (}M_2\ N_2\texttt{)}\ \ldots\texttt{(else}\ N_n\texttt{))}$$

can be understood as an abbreviation for

$$\texttt{(if}\ M_1\ N_1\ \texttt{(if}\ M_2\ N_2\ \ldots\texttt{(if}\ M_{n-1}\ N_{n-1}\ N_n\texttt{)}\ldots\texttt{))}.$$

So we can define the evaluation of extended-⟨exp⟩'s by translating them into kernel-⟨exp⟩'s prior to evaluation by the (kernel) Substitution Model. This translation is linear-time, one-pass, and yields a kernel-⟨exp⟩ of

size proportional to the original extended-⟨exp⟩. Real Scheme interpreters and compilers typically carry out such translations. The Revised[4] Scheme Report describes the above translations and others for and, or, **begin**, **(define** (⟨var⟩ ⟨formals⟩) ⟨exp⟩) and several further forms.

Actually, a **begin** expression is pointless in a truly functional language since (**begin** $N_1 \dots N_n$) has the same value as N_n. Nevertheless, we include it to control side-effects by calls to operations like `display`, etc., which do not affect the value returned by an evaluation.

1.3 Functional Values

The purpose of evaluating an expression is to obtain its "value."

⟨lambda-val⟩ ::= (**lambda** (⟨formals⟩) ⟨exp⟩)
⟨letrec-free-val⟩ ::= ⟨lambda-val⟩ | ⟨constant⟩
⟨value⟩ ::= ⟨letrec-free-val⟩ | (**letrec** ((⟨var⟩ ⟨letrec-free-val⟩)*) ⟨value⟩)

The ⟨letrec-free-val⟩'s will play a particularly important role in specifying Scheme's evaluation rules. We let $V, V_1, \dots$ denote ⟨letrec-free-val⟩'s.

2 Free and Bound Variable Occurrences

We define the *free* and *bound occurrences* of variables in an expression M.

(1) M is a ⟨constant⟩.

$$\text{FreeO}(M) = \emptyset$$
$$\text{BoundO}(M) = \emptyset$$

(2) M is x.

$$\text{FreeO}(M) = \{\text{the occurrence of } x \text{ in } M\}$$
$$\text{BoundO}(M) = \emptyset$$

(3) M is (*key* $N_1 \cdots$) or ($N_1 \cdots$).

$$\text{FreeO}(M) = \text{FreeO}(N_1, \cdots)$$
$$\text{BoundO}(M) = \text{BoundO}(N_1, \cdots)$$

where *key* is a ⟨nonbinding-keyword⟩.

(4) M is (**lambda** ($x_1 \dots$) N).

$$\text{FreeO}(M) = \{\, o \in \text{FreeO}(N) \mid o \text{ is } \textit{not} \text{ an occurrence of one of } x_1, \dots \}$$
$$\text{BoundO}(M) = \{\, o \in \text{FreeO}(N) \mid o \textit{ is} \text{ an occurrence of one of } x_1, \dots \} \cup \text{BoundO}(N)$$

(5) M is (**letrec** ((x_1 N_1)$\cdots$)B).

$$\text{FreeO}(M) = \{\, o \in \text{FreeO}(B) \cup \text{FreeO}(N_1) \cup \cdots \mid o \text{ is } \textit{not} \text{ an occurrence of one of } x_1, \dots \}$$
$$\text{BoundO}(M) = \{\, o \in \text{FreeO}(B) \cup \text{FreeO}(N_1) \cup \cdots \mid o \textit{ is} \text{ an occurrence of one of } x_1, \dots \} \cup \text{BoundO}(B) \cup \text{BoundO}(N_1) \cup \cdots$$

A variable is *free* in an expression if it has a free occurrence in the expression. It is free in a set of expressions if it is free in any of them. We write FreeV(M) for the set of variables free in M; similarly for FreeV($\{M_1, \dots\}$).

3 Substitutions

A *substitution*, σ, is formally defined to be a total function whose domain is a finite set of $\langle$var$\rangle$'s and whose range is a set of $\langle$exp$\rangle$'s.

Any substitution σ determines an inductively defined function from $\langle$exp$\rangle$'s to $\langle$exp$\rangle$'s. We write $M\sigma$ for the result of applying this function to M. More precisely, $M\sigma$ is defined by induction simultaneously on the structure of M and the size of domain(σ). The base Case (1) of the definition ensures that the function on expressions determined by σ acts the same on variables in domain(σ) as σ itself.

(1)

$$x\sigma = \begin{cases} \sigma(x) & \text{if } x \in \text{domain}(\sigma), \\ x & \text{otherwise.} \end{cases}$$

(2) If domain$(\sigma) = \emptyset$, then

$$M\sigma = M$$

For any set, $\mathcal{F}$, of $\langle$var$\rangle$'s, we write $\sigma \restriction \mathcal{F}$ for the restriction of σ to $\mathcal{F} \cap \text{domain}(\sigma)$.

(3)

$$M\sigma = M\,(\sigma \restriction \text{FreeV}(M))$$

Thus, if FreeV$(M) = \emptyset$, then $M\sigma = M$.

(4)

$$(N_1 \cdots)\sigma = (N_1\sigma \cdots)$$
$$(\mathit{key}\ N_1 \cdots)\sigma = (\mathit{key}\ N_1\sigma \cdots)$$

where *key* is a $\langle$nonbinding-keyword$\rangle$.

Scheme is specified to obey the "static" scoping conventions of familiar mathematical notation. This means that when M has binding constructs, free variables in expressions being substituted into M should not get bound by binding constructs in M. Such unintended binding is prevented by selectively renaming bound variables in M to be "fresh" variables. (Substituting without renaming would model the "dynamic" scoping of pre-Scheme Lisp dialects, leading to notorious "false capture" or "funarg" problems.)

A *renaming* is defined to be a substitution which maps the variables in its domain to *distinct* $\langle$var$\rangle$'s. A renaming, τ, is *fresh* with respect to a set of expressions and substitutions if none of the variables in its range have free or binding occurrences in any of the expressions or domains and ranges of the substitutions in the set. When the relevant set is clear, we simply say that τ is a *fresh renaming.*

For substitutions σ, τ, the *extension* of τ by σ, written $\tau + \sigma$, is the substitution combining σ and τ, with τ given priority where the domains overlap. That is,

$$(\tau + \sigma)\,(y) = \begin{cases} \tau(y) & \text{if } y \in \text{domain}(\tau), \\ \sigma(y) & \text{if } y \in \text{domain}(\sigma) - \text{domain}(\tau). \end{cases}$$

Note that in general $\tau + \sigma \neq \sigma + \tau$.

The next cases in the definition of $M\sigma$ are simplified by assuming that domain$(\sigma) \subseteq$ FreeV(M). By Case (3), there is no loss of generality in this assumption.

(5)

$$\texttt{(lambda (}x_1\, x_2 \cdots\texttt{)}\ N\texttt{)}\sigma = \texttt{(lambda (}x_1\tau\ \ x_2\tau \cdots\texttt{)}\ N(\tau + \sigma)\texttt{)}$$

where τ is a fresh renaming whose domain is $\{x_1, \ldots\} \cap \text{FreeV}(\text{range}(\sigma))$.

(6)

$$\texttt{(letrec ((}x_1\ \ N_1\texttt{)}\cdots\texttt{)}\ B\texttt{)}\sigma = \texttt{(letrec ((}x_1\tau\ \ N_1(\tau + \sigma)\texttt{)}\ \ \cdots\texttt{)}\ B(\tau + \sigma)\texttt{)}$$

where τ is a fresh renaming whose domain is $\{x_1, \ldots\} \cap \text{FreeV}(\text{range}(\sigma))$.

4 Rewrite Rules

In Revised[4] Scheme, only **#f** counts as false in conditional expressions. Other values such as numerals and procedures count as true.

$$\langle\text{false-value}\rangle ::= \texttt{\#f}$$

- IF

$$\texttt{(if}\ V\, N_1\, N_2\texttt{)} \longrightarrow \begin{cases} N_2 & \text{if } V \text{ is a } \langle\text{false-value}\rangle, \\ N_1 & \text{otherwise.} \end{cases}$$

To capture the Revised[4] Scheme behavior of $\langle$built-in$\rangle$'s, we introduce a partial function, System-Eval from $\langle$exp$\rangle$'s to $\langle$exp$\rangle$'s.

- CONST: *built-in and system constants*

$$\texttt{(}\mathit{cnst}\ \ \langle\text{letrec-free-val}\rangle^*\texttt{)} \longrightarrow \text{System-Eval}\left(\texttt{(}\mathit{cnst}\ \ \langle\text{letrec-free-val}\rangle^*\texttt{)}\right)$$

where *cnst* is a $\langle$built-in$\rangle$ or $\langle$system-constant$\rangle$, and System-Eval $\left(\texttt{(}\mathit{cnst}\ \ \langle\text{letrec-free-val}\rangle^*\texttt{)}\right)$ is defined.

For example,

$$\text{System-Eval}(\texttt{(<<+>> 3 -7.1 9)}) = \texttt{4.9}$$

Errors are somewhat messy to incorporate into the Substitution Model, so we take System-Eval to be undefined in cases where Revised[4] Scheme specifies errors. For example, an expression such as `(<<+>> #t)` which generates an immediate Revised[4] Scheme error will not match the lefthand side of any rewrite rule.

System-Eval also handles $\langle$system-constant$\rangle$'s generated by calls to external code which return procedures. This allows the Substitution Model implementation to interface with real Scheme code.

- LAM-APP: *lambda-application*

$$\texttt{((lambda (}x_1 \ldots\texttt{)}\ B\texttt{)}\ \ N_1 \ldots\texttt{)} \longrightarrow \texttt{(letrec ((}x_1\tau\ \ N_1\texttt{)}\ldots\texttt{)}\ \ B\tau\texttt{)}$$

where τ is a fresh renaming whose domain is $\{x_1, \ldots\} \cap \text{FreeV}(\{N_1, N_2, \ldots\})$.

We write $\{x \leftarrow M\}$ for the substitution whose domain is $\{x\}$ and which maps x to M.

- letrec

  INST: *letrec-instantiation*

$$\texttt{(letrec (}\langle\text{bindings}\rangle\ \texttt{(}x\ V\texttt{)}\cdots\texttt{)}\ B\{z \leftarrow x\}\texttt{)} \longrightarrow \texttt{(letrec (}\langle\text{bindings}\rangle\ \texttt{(}x\ V\texttt{)}\cdots\texttt{)}\ B\{z \leftarrow V\}\texttt{)}$$

  where B has exactly one free occurrence of z.

  OUT: *letrec-out*:

$$\texttt{(}M_1 \ldots M_k\ \texttt{(letrec (}\langle\text{bindings}\rangle\texttt{)}\ B\texttt{)}\ M_{k+1} \ldots M_n\texttt{)} \longrightarrow \texttt{(letrec (}\langle\text{bindings}\rangle\texttt{)}\ \texttt{(}z_1 \ldots z_k\, B\, z_{k+1} \ldots z_n\texttt{))}\sigma$$

$$\texttt{(}key\ M_1 \ldots M_k\ \texttt{(letrec (}\langle\text{bindings}\rangle\texttt{)}\ B\texttt{)}\ M_{k+1} \ldots M_n\texttt{)} \longrightarrow \texttt{(letrec (}\langle\text{bindings}\rangle\texttt{)}\ \texttt{(}key\ z_1 \ldots z_k\, B\, z_{k+1} \ldots z_n\texttt{))}\sigma$$

  where σ is the substitution whose domain is a set $\{z_1, \ldots, z_n\}$ of fresh variables, $\sigma(z_i) = M_i$, and *key* is a ⟨nonbinding-keyword⟩.

  FLAT: *letrec-flatten*:

$$\texttt{(letrec (}\langle\text{bindings}\rangle_1\ \texttt{(}x\ \texttt{(letrec (}\langle\text{bindings}\rangle_2\texttt{)}\ B_0\texttt{))}\ \langle\text{bindings}\rangle_3\texttt{)}\ B\texttt{)} \longrightarrow \texttt{(letrec (}\langle\text{bindings}\rangle_1\ \texttt{(}\langle\text{bindings}\rangle_2\,\tau\texttt{)}\ \texttt{(}x\ B_0\tau\texttt{)}\ \langle\text{bindings}\rangle_3\texttt{)}\ B\texttt{)}$$

  where τ is a fresh renaming whose domain is $\mathcal{F}_2 \cap (\mathcal{F}_1 \cup \{x\} \cup \mathcal{F}_3 \cup \text{FreeV}(B))$ and $\mathcal{F}_i$ is the set of binding variables in $\langle\text{binding}\rangle_i$.

- BOOL?

$$\texttt{(<<boolean?>>}\ V\texttt{)} \longrightarrow \begin{cases} \texttt{\#t} & \text{if } V \text{ is } \texttt{\#t} \text{ or } \texttt{\#f}, \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

- NUM?

$$\texttt{(<<number?>>}\ V\texttt{)} \longrightarrow \begin{cases} \texttt{\#t} & \text{if } V \text{ is a } \langle\text{numeral}\rangle, \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

- PROC?

$$\texttt{(<<procedure?>>}\ V\texttt{)} \longrightarrow \begin{cases} \texttt{\#t} & \text{if } V \text{ is a } \langle\text{lambda-val}\rangle, \langle\text{scheme-constant}\rangle, \\ & \quad \text{or } \langle\text{system-constant}\rangle, \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

5 Garbage Collection Rules

$$\texttt{(letrec ()}\ B\texttt{)} \longrightarrow B$$

$$\texttt{(letrec ((}x_1\ N_1\texttt{)}\cdots\texttt{)}\ B\texttt{)} \longrightarrow \texttt{(letrec ((}x_{i_1}\ N_{i_1}\texttt{)}\cdots\texttt{)}\ B\texttt{)}$$

where $\{i_1, i_2, \ldots\} = \{\, i \mid x_i \text{ is not a } \textit{live}\text{-variable or } N_i \text{ is not a value}\}$. These *live* variables are defined inductively by the conditions:

(1) all variables in $\text{FreeV}(B)$ are live, and

(2) if x_i is live, then all variables in $\text{FreeV}(N_i)$ are live.

6 Contexts and Rewriting

A *context* is an ⟨exp⟩ with exactly one occurrence of a designated variable referred to as *the hole.* If C is a context, we write $C[M]$ for the expression that results from replacing the hole in C by M *without any renaming*; the hole itself is written [ ].

If $M_1 = C[N_1]$ and $M_2 = C[N_2]$ for some context C, and "$N_1 \longrightarrow N_2$" matches any rewrite rule above, then we say M_1 *rewrites in one step to* M_2, written

$$M_1 \underset{\mathbf{rew}}{\longrightarrow} M_2\,.$$

In particular, when "$N_1 \longrightarrow N_2$" matches one of the garbage collecting rules, we write

$$M_1 \underset{\mathbf{grbg}}{\longrightarrow} M_2\,.$$

We say M *rewrites to* N, written $M \underset{\mathbf{rew}}{\longrightarrow}^* N$ if either $M = N$, or else

$$M \underset{\mathbf{rew}}{\longrightarrow} M_1 \underset{\mathbf{rew}}{\longrightarrow} \cdots \underset{\mathbf{rew}}{\longrightarrow} N$$

for some expressions $M_1, \ldots$; similarly for $\underset{\mathbf{grbg}}{\longrightarrow}^*$. An expression, M, is a *normal form* when there is no N such that $M \underset{\mathbf{rew}}{\longrightarrow} N$. For example, according to the above rules of our Substitution Model, all ⟨constant⟩'s and ⟨var⟩'s are normal forms.

In general, an expression may have many distinct normal forms reflecting the different ways rewrite rules might be applied to it. Despite this, *numerical* normal forms are unique:

> **Theorem** (Determinacy). If an expression M has a normal form which is a numeral, then that numeral is the only normal form of M.

Determinacy implies that in calculating a *numerical* value of an expression, there is no need to consider where and which rewrite rules to apply. This is a fundamental property distinguishing the functional kernel of Scheme from the fuller language with side-effects.

The Determinacy Theorem is far from obvious, and considerable ingenuity is needed to prove it.

7 Correctness

The *initial global context*, C_{init}, is defined to be the context in which variables are bound to all the corresponding constants, viz.,

> C_{init} ::= `(letrec ((+ <<+>>)... (sqrt <<sqrt>>) ... (apply <<apply>>) ...) [ ])`.

Let Numval(M) denote the unique numeral, if any, which is the normal form of $C_{\text{init}}[M]$. Numval(M) is undefined if M has a normal form which is not a numeral, or if M has no normal form.

> **Correctness Claim.** Let M be an ⟨exp⟩ without scheme- or system-constants whose value in the initial environment is specified in Revised^4 Scheme. If the specified value is a numeral, then Numval(M) is defined and equals the numeral. Conversely, if Numval(M) is defined, then it equals the value specified in Revised^4 Scheme when M is evaluated in the initial environment.

Note that the Claim leaves open the possibility that Numval(M) may be defined in cases when M does not have a specified value in Revised^4 Scheme.

We hesitate to call this Claim a "Theorem", because the Revised^4 Scheme specifications remain somewhat imprecise, especially about the specification of `equal?`. We believe the Claim is a Theorem for ⟨exp⟩'s which do not contain `equal?`. Proving such a theorem connecting the (denotational semantics) specification in the Revised^4 Scheme Report with the rewriting rules of our Substitution Model requires sophisticated proof methods from programming language theory.

8 Order of Evaluation

Evaluation contexts, are used to define where and which rewrite rules Scheme would apply to subexpressions of an expression being evaluated.

$$
\begin{aligned}
\langle\text{eval-context}\rangle ::= \;& [\,] \\
\mid\;& (\langle\text{letrec-free-val}\rangle^* \; [\,] \; \langle\text{exp}\rangle^*) \\
\mid\;& (\texttt{letrec} \; ((\langle\text{var}\rangle \; \langle\text{letrec-free-val}\rangle)^* (\langle\text{var}\rangle \; [\,]) \langle\text{bindings}\rangle) \langle\text{exp}\rangle) \\
\mid\;& (\texttt{letrec} \; ((\langle\text{var}\rangle \langle\text{letrec-free-val}\rangle)^*) \; [\,]) \\
\mid\;& (\texttt{if} \; [\,] \; \langle\text{exp}\rangle \; \langle\text{exp}\rangle)
\end{aligned}
$$

If $M_1 = E[N_1]$ and $M_2 = E[N_2]$ for some *evaluation* context E, and "$N_1 \longrightarrow N_2$" matches a rewrite rule, then we say M_1 *evaluates in one step* to M_2, written

$$M_1 \underset{\text{eval}}{\longrightarrow} M_2 .$$

Also $\underset{\text{eval}}{\longrightarrow}^*$ is defined similarly to $\underset{\text{rew}}{\longrightarrow}^*$.

Actually, there is one further technicality in the definition. The *letrec*-rules are written in a general form that leaves many ways that a lefthand side might match an entire expression. To pin down how the *letrec*-rules should match an expression in the hole of an evaluation context, restricted forms of the rules are required to be used in the above definition of one-step evaluation. Namely,

Restricted letrec rules:

- *letrec-reduce*: The ⟨bindings⟩ in the rule must bind variables to ⟨letrec-free-val⟩'s.
- *letrec-out*: The expressions $M_1 \ldots M_k$ preceding the ⟨bindings⟩ must be ⟨letrec-free-val⟩'s.
- *letrec-flatten*: The $\langle\text{bindings}\rangle_1$ in the rule must bind variables to ⟨letrec-free-val⟩'s.

We remark that rules for call-with-current-continuation can also be defined nicely using evaluation contexts. For example, if M has no free variables, then we define $E[(\texttt{call/cc}\ M)]$ to evaluate in one step to $(M\ (\texttt{lambda}\ (z)\ E[z]))$ where z is fresh.

An expression M is *stopped* when there is no N such that $M \underset{\text{eval}}{\longrightarrow} N$. For example, ⟨lambda-val⟩'s are stopped.

In contrast to rewriting, there is only *one* way to do a *non*-garbage-collecting step in the *evaluation* of an expression. This can be verified straightforwardly from the definition of one-step evaluation, though there are a lot of cases to check. It is also straightforward to check that where a sequence of evaluation steps stops doesn't depend on when and where garbage-collecting rules are applied. In particular, the garbage collection rules *commute* with the other rewriting rules: if $M_1 \underset{\text{rew}}{\longrightarrow} M_2$ and $M_1 \underset{\text{grbg}}{\longrightarrow} M_3$ where $M_2 \not\equiv M_3$, then either $M_2 \underset{\text{grbg}}{\longrightarrow} M_3$, or there is an M_4 such that $M_2 \underset{\text{grbg}}{\longrightarrow} M_4$ and $M_3 \underset{\text{rew}}{\longrightarrow} M_4$. See Figure 1. This implies that

> **Fact.** If $M \underset{\text{eval}}{\longrightarrow}^* N$ and N is stopped, then N is the only such stopped expression. We call N *the result of evaluating M*.

Let Eval(M) denote the ⟨value⟩, if any, which is the result of evaluating $C_{\text{init}}[M]$.

> **Theorem** (Completeness of evaluation rules). Let M be an expression without system-constants. If Numval(M) is defined, then Eval(M) = Numval(M).

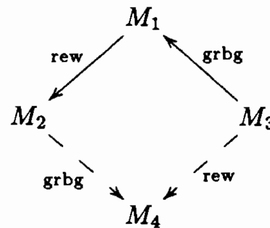


Figure 1: Garbage-collection commutes

An expression M *has garbage* if it is of the form $E[N]$ for some evaluation context E and subexpression N which matches the lefthand side of one of the garbage collection rules of Section 5. It is easy to check that

Fact. Let M be a value. Then M is stopped iff it does not have garbage.

If there is a result of evaluating $C_{\text{init}}[M]$ and it is *not* a value, then evaluation of M in Revised[4] Scheme causes an error. The converse may not be true however. For example, in Revised[4] Scheme `(letrec ((foo foo)) 0)` will cause an "unassigned-variable" error, but

$$\texttt{(letrec ((foo foo)) 0)} \underset{\texttt{rew}}{\longrightarrow} \texttt{(letrec ((foo foo)) 0)}$$

so in our Substitution Model it will rewrite forever without getting stopped. (The implementation will in fact detect this runaway behavior and report the error.)

9 Lists

We extend the functional kernel grammar with:

$$\begin{aligned} \langle\text{keyword}\rangle &::= \ldots \mid \texttt{quote} \\ \langle\text{exp}\rangle &::= \ldots \mid \langle\text{symbol}\rangle \mid \texttt{()} \\ \langle\text{symbol}\rangle &::= \texttt{(quote}\ \langle\text{nonconstant-identifier}\rangle\,\texttt{)} \end{aligned}$$

If M is a $\langle$symbol$\rangle$ or `()`, then

$$\begin{aligned} \text{FreeO}(M) &= \emptyset \\ \text{BoundO}(M) &= \emptyset \end{aligned}$$

In particular, by Case (3) of the definition of substitution,

$$\texttt{(quote } x\texttt{)}\sigma = \texttt{(quote } x\texttt{)}.$$

It is *not* equal to `(quote` $x\sigma$`)`—Case (4) does not apply because `quote` is a keyword which is neither nonbinding nor binding.

In MIT Scheme, the boolean `#f` is identified with the empty list, `()`, and `#f` always prints as `()`. To accomodate this regrettable situation in our model, we define the behaviour of conditionals so that the empty list is treated as a false value.

$$\langle\text{false-value}\rangle ::= \ldots \mid \texttt{()}$$

$$\begin{aligned}
\langle\text{rule-defined}\rangle ::=\ & \ldots \mid \texttt{<<cons>>} \mid \texttt{<<list>>} \mid \texttt{<<car>>} \mid \texttt{<<cdr>>} \mid \texttt{<<apply>>} \mid \texttt{<<symbol?>>} \\
& \mid \texttt{<<pair?>>} \mid \texttt{<<list?>>} \mid \texttt{<<equal?>>} \\
\langle\text{built-in}\rangle ::=\ & \ldots \mid \texttt{<<null?>>} \mid \texttt{<<append>>} \mid \texttt{<<reverse>>} \mid \texttt{<<map>>} \mid \texttt{<<list-ref>>} \\
& \mid \texttt{<<list-tail>>} \mid \texttt{<<delete>>}
\end{aligned}$$

Table 2: Scheme Procedure Constants for Lists

On the other hand, following Revised[4] Scheme, our Substitution Model maintains the distinction between lists and booleans, so `(equal? () #f)`, for example, evaluates to `#f`.

The initial global context is extended so the variable `nil` is bound to `()`. We also include the additional constants given in Table 2.

9.1 List Values

$$\begin{aligned}
\langle\text{letrec-free-val}\rangle &::= \ldots \mid () \mid \langle\text{pair-val}\rangle \\
\langle\text{pair-val}\rangle &::= (\texttt{<<list>>}\ \langle\text{letrec-free-val}\rangle^{+}) \mid \langle\text{cons-val}\rangle \\
\langle\text{cons-val}\rangle &::= (\texttt{<<cons>>}\ \langle\text{letrec-free-val}\rangle\ \langle\text{nonlist-letrec-free-val}\rangle) \\
\langle\text{nonlist-letrec-free-val}\rangle &::= \langle\text{constant}\rangle \mid \langle\text{lambda-val}\rangle \mid \langle\text{symbol}\rangle \mid \langle\text{cons-val}\rangle
\end{aligned}$$

9.2 List Rules

- CONS

$$\begin{aligned}
(\texttt{<<cons>>}\ N_1\ ()\ ) &\longrightarrow (\texttt{<<list>>}\ N_1) \\
(\texttt{<<cons>>}\ N_1\ (\texttt{<<list>>}\ N_2 \cdots)) &\longrightarrow (\texttt{<<list>>}\ N_1\ N_2 \cdots)
\end{aligned}$$

- list

$$(\texttt{<<list>>}) \longrightarrow ()$$

- CAR

$$\begin{aligned}
(\texttt{<<car>>}\ (\texttt{<<cons>>}\ V\ \langle\text{value}\rangle)) &\longrightarrow V \\
(\texttt{<<car>>}\ (\texttt{<<list>>}\ V\ \langle\text{value}\rangle^{*})) &\longrightarrow V
\end{aligned}$$

- CDR

$$\begin{aligned}
(\texttt{<<cdr>>}\ (\texttt{<<cons>>}\ \langle\text{value}\rangle\ V)) &\longrightarrow V \\
(\texttt{<<cdr>>}\ (\texttt{<<list>>}\ V_1\ V_2 \ldots)) &\longrightarrow (\texttt{<<list>>}\ V_2 \ldots) \\
(\texttt{<<cdr>>}\ (\texttt{<<list>>}\ V)) &\longrightarrow ()
\end{aligned}$$

- APPLY

$$\begin{aligned}
(\texttt{<<apply>>}\ V\ (\texttt{<<list>>}\ V_1 \cdots)) &\longrightarrow (V\ V_1 \cdots) \\
(\texttt{<<apply>>}\ V\ ()) &\longrightarrow (V)
\end{aligned}$$

- SYMB?

$$(\texttt{<<symbol?>>}\ V) \longrightarrow \begin{cases} \texttt{\#t} & \text{if } V \text{ is a } \langle\text{symbol}\rangle, \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

- PAIR?

$$(\texttt{<<pair?>>}\ V) \longrightarrow \begin{cases} \texttt{\#t} & \text{if } V \text{ is a } \langle\text{pair-val}\rangle, \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

- LIST?

$$(\texttt{<<list?>>}\ V) \longrightarrow \begin{cases} \texttt{\#t} & \text{if } V \text{ is } \texttt{()} \text{ or of the form } (\texttt{<<list>>}\ \langle\text{letrec-free-val}\rangle^{+}), \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

- EQUAL?

$$(\texttt{<<equal?>>}\ V\ \ V) \longrightarrow \texttt{\#t}$$
$$(\texttt{<<equal?>>}\ V_1\ \ V_2) \longrightarrow \texttt{\#f}$$

where V_1 and V_2 are *distinguishable.* Other cases of `equal?` are unspecified.

The distinguishable pairs of values are defined inductively by the conditions:

(1) Each ⟨boolean⟩, ⟨numeral⟩, ⟨symbol⟩, and `()` is distinguishable from all other ⟨letrec-free-val⟩'s,

(2) two ⟨letrec-free-val⟩'s with different phrase types among ⟨boolean⟩, ⟨numeral⟩, ⟨symbol⟩, ⟨lambda-val⟩, ⟨pair-val⟩, ⟨cons-val⟩ are distinguishable,

(3) ⟨pair-val⟩'s with different numbers of subforms are distinguishable,

(4) ⟨pair-val⟩'s are distinguishable if their corresponding subforms are distinguishable.

By the definition above, for example, the result of `(equal? null? not)` and `(equal? + -)` is unspecified in the Substitution Model, while

$$\text{Eval}(\texttt{(equal? (lambda(x)x) (lambda(x)x))}) = \texttt{\#t}$$

In MIT Scheme, on the other hand, these expressions respectively return `#t`, `#f`, and `#f`.

Exercise. Define

$$\langle\text{simple-val}\rangle ::= \langle\text{numeral}\rangle \mid \langle\text{boolean}\rangle \mid \langle\text{symbol}\rangle \mid \texttt{()}$$

Explain why the Correctness Claim of Section 7 implies the slightly stronger Claim obtained by replacing Numval by Simpval. Then reformulate the Claim so it also holds for

$$\langle\text{printable-val}\rangle ::= \langle\text{simple-val}\rangle \mid (\texttt{<<list>>}\ \langle\text{print-val}\rangle^{*}\ ) \mid (\texttt{<<cons>>}\ \langle\text{print-val}\rangle\ \langle\text{print-val}\rangle\ )$$

Finally, discuss the possibilities for extending the Theorem to ⟨letrec-free-val⟩'s and ⟨value⟩'s.

10 Remarks on the Implementation

Evaluation of top-level `define`'s results in extensions of the global context, as in underlying Scheme. Top-level expressions are automatically evaluated in the global context. This context is usually not displayed to avoid cluttering the printout. System-Eval generates errors following the underlying Scheme.

A somewhat fuller syntax of Scheme expressions is supported in the implementation: conditionals and `if`'s with no alternative branch, named `let`'s, strings, etc. At present, vectors and quoted lists are *not* supported, nor are sequences of expressions in `lambda` and `cond` bodies—sequences can only appear within explicit `begin`'s. Internal ⟨defines⟩ are buggy, to it is prudent to use ⟨letrec⟩ directly instead.

Beware, the MIT Scheme printer prints `(quote foo)` as `'foo`.

Acknowledgements

The authors are grateful to Derek Lindner, who proofread several drafts of the formal Substitution Model, especially checking for consistency with the implementation; to Daniel Otth who made numerous valuable suggestions for defining this formal model to maintain rewriting theory properties such as confluence; and to David M. Jones who provided his masterful LaTeX skills.

Appendix: Direct Rules for the Extended Syntax

These rules are included as a guide for implementations supporting extended-⟨exp⟩'s rather than translating them in to kernel-exp's.

- let

  If M is `(let ((`x_1 N_1`)` $\cdots$`)` B`)`, then

$$\begin{aligned}\mathrm{FreeO}(M) &= \{\, o \in \mathrm{FreeO}(B) \mid o \text{ is } \textit{not} \text{ an occurrence of } x_1, \ldots \} \cup \\ &\quad \mathrm{FreeO}(N_1) \cup \cdots \\ \mathrm{BoundO}(M) &= \{\, o \in \mathrm{FreeO}(B) \mid o \textit{ is} \text{ an occurrence of one of } x_1, \ldots \} \cup \\ &\quad \mathrm{BoundO}(B) \cup \mathrm{BoundO}(N_1) \cup \cdots\end{aligned}$$

  substitution:

$$\texttt{(let ((}x_1\ N_1\texttt{)}\cdots\texttt{)}\ B\texttt{)}\sigma = \texttt{(let ((}x_1\tau\ N_1\sigma\texttt{)}\cdots\texttt{)}\ B(\tau+\sigma)\texttt{)}$$

  where τ is a fresh renaming whose domain is $\{x_1, \ldots\} \cap \mathrm{FreeV}(\mathrm{range}(\sigma \restriction \mathrm{FreeV}(B)))$.

  LET: *rewrite rule*

$$\texttt{(let ((}x_1\ N_1\texttt{)}\cdots\texttt{)}\ B\texttt{)} \longrightarrow \texttt{((lambda (}x_1\cdots\texttt{)}\ B\texttt{)}\ N_1\cdots\texttt{)}$$

- cond

  substition

$$\texttt{(cond (}N_1\ N_2\texttt{)}\ldots\texttt{(else }N\texttt{))}\sigma = \texttt{(cond (}N_1\sigma\ N_2\sigma\texttt{)}\ldots\texttt{(else }N\sigma\texttt{))}$$

  COND: rewrite rule

$$\begin{aligned}\texttt{(cond (}V\,N\texttt{) ...)} &\longrightarrow \begin{cases}\texttt{(cond ...)} & \text{if } V \text{ is a ⟨false-value⟩,}\\ N & \text{otherwise.}\end{cases}\\ \texttt{(cond (else }N\texttt{))} &\longrightarrow N\end{aligned}$$

  AND

$$\begin{aligned}\texttt{(and }V\,N_1\cdots\texttt{)} &\longrightarrow \begin{cases}V & \text{if } V \text{ is a ⟨false-value⟩,}\\ \texttt{(and }N_1\cdots\texttt{)} & \text{otherwise.}\end{cases}\\ \texttt{(and }N\texttt{)} &\longrightarrow N\\ \texttt{(and)} &\longrightarrow \texttt{\#t}\end{aligned}$$

  OR

$$\begin{aligned}\texttt{(or }V\,N_1\cdots\texttt{)} &\longrightarrow \begin{cases}\texttt{(or }N_1\cdots\texttt{)} & \text{if } V \text{ is a ⟨false-value⟩,}\\ V & \text{otherwise.}\end{cases}\\ \texttt{(or }N\texttt{)} &\longrightarrow N\\ \texttt{(or)} &\longrightarrow \texttt{\#f}\end{aligned}$$

Problem Set 1

Reading assignment. Winskel Chapters 1–3.

Due: 24 September 1993.

Problems 1–3 refer to the substitution model Scheme (defined in Handout 6) and the example evaluations of the substitution model (from Handout 5).

Problem 1. List the rules that are applied in the substitution model evaluation of `(rec-factorial 5)`. You do not need to list the rules in order, or any rule twice.

Problem 2. For each of the following, give the first evaluation step at which garbage collection occurs, and the binding that is garbage collected.

1. `(iter-fact 5)`
2. `(y-iter-fact 5)`

Problem 3. As a function of n, how many steps will it take the each of the following to evaluate the factorial of n in the substitution model?

1. `rec-factorial`
2. `iter-factorial`
3. `y-fact`
4. `y-iter-fact`

Problem 4. Winskel, Exercise 3.4.

Problem 5. Winskel, Exercise 3.5.

Problem 6. Winskel, Exercise 3.6.

Clarifications to Problem Set 1

This handout should clarify some ambiguities in the first three problems of Problem Set 1 (Handout 7).

Please note that HANDOUTS 5 AND 6 HAVE BEEN REVISED; make sure you are using the versions labelled "Revised 22 September 1993".

In problem 2, "`(iter-fact 5)`" should be "`(iter-factorial 5)`".

We would like make the notions of "rule", "step", "how many steps", etc., as clear as possible. We use the following annotated example to do so. (You may recognize it as the "passkey" example.)

Example:

A set of integers can be represented as a predicate, that is, a function from integers to booleans. Applying the function to an integer returns true if the integer is an element of the set, and false otherwise.

Thus the empty set is represented by the function that ALWAYS returns false:

```
SUB-EVAL==>
(define empty-set (lambda (any) ()))

--------
empty-set has been defined
```

And we can take an integer and a set and create a new set consisting of the old set plus the new integer through the function `add-to-intset`:

```
SUB-EVAL==>
(define
 add-to-intset
 (lambda
  (new-int intset)
  (lambda (some-int) (if (= some-int new-int) #t (intset some-int)))))

--------
add-to-intset has been defined
```

Using `add-to-intset` and `empty-set`, we can create the "set" $\{1,2,3\}$. We see here the first interesting evaluation of the example.

```
SUB-EVAL==>
(define
 example-intset
 (add-to-intset 3 (add-to-intset 2 (add-to-intset 1 empty-set))))
==[1][INST: add-to-intset]==>

(define
 example-intset
 ((lambda
   (new-int intset)
   (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
  3
  (add-to-intset 2 (add-to-intset 1 empty-set))))
```

This shows that the first evaluation step is by the rule INST, and consists of instantiating the (global) definition of **add-to-intset**. These rule names (INST, ...) are listed in the REVISED version of handout 6.

```
==[2][INST: add-to-intset]==>

(define
 example-intset
 ((lambda
   (new-int intset)
   (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
  3
  ((lambda
    (new-int intset)
    (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
   2
   (add-to-intset 1 empty-set))))
==[3][INST: add-to-intset]==>

(define
 example-intset
 ((lambda
   (new-int intset)
   (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
  3
  ((lambda
    (new-int intset)
    (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
   2
   ((lambda
     (new-int intset)
     (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
    1
    empty-set))))
==[4][INST: empty-set]==>

(define
 example-intset
 ((lambda
   (new-int intset)
   (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
  3
  ((lambda
    (new-int intset)
    (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
   2
   ((lambda
     (new-int intset)
     (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
    1
    (lambda (any) ())))))
==[5][LAM-APP]==>

(define
 example-intset
```

```
 ((lambda
   (new-int intset)
   (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
  3
  ((lambda
    (new-int intset)
    (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
   2
   (letrec
    ((new-int 1) (intset (lambda (any) ())))
    (lambda (some-int) (if (= some-int new-int) #t (intset some-int)))))))
```

Here we see the first non-INST rule applied. The LAM-APP rule turns the application of a `lambda` into a `letrec`, in this case binding `new-int` and `intset`.

```
==[6][LAM-APP]==>

(define
 example-intset
 ((lambda
   (new-int intset)
   (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
  3
  (letrec
   ((new-int 2)
    (intset
     (letrec
      ((new-int 1) (intset (lambda (any) ())))
      (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))))
   (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))))
==[7][FLAT]==>

(define
 example-intset
 ((lambda
   (new-int intset)
   (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
  3
  (letrec
   ((new-int 2)
    (new-int#1 1)
    (intset#1 (lambda (any) ()))
    (intset
     (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int)))))
   (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))))
==[8][LAM-APP]==>

(define
 example-intset
 (letrec
  ((new-int 3)
   (intset
    (letrec
     ((new-int 2)
```

```
      (new-int#1 1)
      (intset#1 (lambda (any) ()))
      (intset
       (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int)))))
     (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))))
  (lambda (some-int) (if (= some-int new-int) #t (intset some-int)))))
==[9][FLAT]==>

(define
 example-intset
 (letrec
  ((new-int 3)
   (new-int#2 2)
   (new-int#1 1)
   (intset#1 (lambda (any) ()))
   (intset#2
    (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int))))
   (intset
    (lambda (some-int) (if (= some-int new-int#2) #t (intset#2 some-int)))))
  (lambda (some-int) (if (= some-int new-int) #t (intset some-int)))))

--------
example-intset has been defined
```

Finally, we can test whether 2 is a member of **example-intset**:

```
SUB-EVAL==> (example-intset 2)
(example-intset 2)
==[1][INST: example-intset]==>

((letrec
  ((new-int 3)
   (new-int#2 2)
   (new-int#1 1)
   (intset#1 (lambda (any) ()))
   (intset#2
    (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int))))
   (intset
    (lambda (some-int) (if (= some-int new-int#2) #t (intset#2 some-int)))))
  (lambda (some-int) (if (= some-int new-int) #t (intset some-int))))
 2)
==[2][OUT]==>

(letrec
 ((new-int 3)
  (new-int#2 2)
  (new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int))))
  (intset
   (lambda (some-int) (if (= some-int new-int#2) #t (intset#2 some-int)))))
 ((lambda (some-int) (if (= some-int new-int) #t (intset some-int))) 2))
```

```
==[3][LAM-APP]==>

(letrec
 ((new-int 3)
  (new-int#2 2)
  (new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int))))
  (intset
   (lambda (some-int) (if (= some-int new-int#2) #t (intset#2 some-int)))))
 (letrec ((some-int 2)) (if (= some-int new-int) #t (intset some-int))))
==[4][INST: =][INST: some-int]==>

(letrec
 ((new-int 3)
  (new-int#2 2)
  (new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int))))
  (intset
   (lambda (some-int) (if (= some-int new-int#2) #t (intset#2 some-int)))))
 (letrec ((some-int 2)) (if (<<=>> 2 new-int) #t (intset some-int))))
```

Here we see that the evaluator has taken TWO STEPS, and has decided not to print the result of taking the first step. The `[4]` in the arrow indicates that the first step taken, `[INST: =]`, was step number 4. The second step, `[INST: some-int]`, was step number 5. The next step will be step number 6, as is indicated below.

```
==[6][INST: new-int][GC: new-int]==>

(letrec
 ((new-int#2 2)
  (new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int))))
  (intset
   (lambda (some-int) (if (= some-int new-int#2) #t (intset#2 some-int)))))
 (letrec ((some-int 2)) (if (<<=>> 2 3) #t (intset some-int))))
```

Here we have the first garbage collection. There is only one use of `new-int`, so once it has been instantiated, we can garbage collect the binding.

```
==[7][CONST: <<=>>][IF]==>

(letrec
 ((new-int#2 2)
  (new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int))))
```

```
  (intset
   (lambda (some-int) (if (= some-int new-int#2) #t (intset#2 some-int)))))
 (letrec ((some-int 2)) (intset some-int)))
```

Here in the first step after the garbage collection, we see that THE GARBAGE COLLECTION DID NOT TAKE UP ANY "STEPS"! That is, we can see from above that step 6 is `[INST: new-int]`, that the GC takes place immediately after that, and that step 7 is `[CONST: <<=>>]`. This is a matter of taste (which you may or may not agree with). At any rate, it immediately raises the question of what we mean in problem 2 by "the first evaluation step at which garbage collection occurs". For this example, we would like you to answer "6". We want the step number printed in the arrow between the expression which contains the binding and the expression where the binding has been garbage collected.

```
==[9][INST: intset][GC: intset]==>

(letrec
 ((new-int#2 2)
  (new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int)))))
 (letrec
  ((some-int 2))
  ((lambda (some-int) (if (= some-int new-int#2) #t (intset#2 some-int)))
   some-int)))
==[10][INST: some-int][GC: some-int]==>

(letrec
 ((new-int#2 2)
  (new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int)))))
 ((lambda (some-int) (if (= some-int new-int#2) #t (intset#2 some-int))) 2))
==[11][LAM-APP]==>

(letrec
 ((new-int#2 2)
  (new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int)))))
 (letrec ((some-int 2)) (if (= some-int new-int#2) #t (intset#2 some-int))))
==[12][INST: =][INST: some-int]==>

(letrec
 ((new-int#2 2)
  (new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int)))))
 (letrec ((some-int 2)) (if (<<=>> 2 new-int#2) #t (intset#2 some-int))))
==[14][INST: new-int#2][GC: new-int#2]==>

(letrec
```

```
 ((new-int#1 1)
  (intset#1 (lambda (any) ()))
  (intset#2
   (lambda (some-int) (if (= some-int new-int#1) #t (intset#1 some-int)))))
 (letrec ((some-int 2)) (if (<<=>> 2 2) #t (intset#2 some-int))))
==[15][CONST: <<=>>][IF][GC: new-int#1][GC: intset#1][GC: intset#2][GC: some-int]==>

#t
```

Finally, note here that there are TWO steps taken in this last arrow, a CONST and an IF. (Remember, the GC's do not count as steps.) Thus the total number of steps taken is 16, not 15 as you might think if you are not careful.

Problem Set 2

Reading assignment. Winskel § 4.1–4.3

Due: 6 October 1993.

Let A be the expression for recursive factorial using a fixpoint operator:

```
(letrec ((fact (letrec ((f (lambda (fact) (lambda (n)
                          (if (<= n 0)
                              1
                              (* n (fact (- n 1))))))))
               (letrec ((x (lambda (x)
                             (f (lambda (z) ((x x) z))))))
                (letrec ((fact (lambda (z) ((x x) z))))
                 (lambda (n) (if (<= n 0)
                                 1
                                 (* n (fact (- n 1))))))))))
        fact)
```

How can we precisely explain what an *occurrence* of a variable is in an expression?

One way is to think of an expression as a *tree*, i.e., a list without sharing, and describe which branches to follow successively from the root of the tree to get to the occurrence. For example, in A, the last occurrence of the variable `fact` is reached by taking the 3rd branch from the root (the caddr of the list), and the first occurrence of `fact` as an operator applied to `(- n 1)` is reached by successively choosing branches 2, 1, 2, 2, 1, 2, 3, 3, 4, 3, 1 (or taking cadr, then car, then cadr, then cadr, then car, then cadr, then caddr, ...). So to be precise, we could *define* an occurrence in an expression to be a sequence of numbers: the last occurrence of `fact` *is* the sequence $\langle 3\rangle$ and the first occurrence of `fact` as an operator applied to `(- n 1)` *is* the sequence $\langle 2,1,2,2,1,2,3,3,4,3,1\rangle$. This is called the *tree-path* representation of occurrences.

Let *which-var* be the partial function which takes an expression and an occurrence as arguments and returns the variable (if any) at that occurrence. For example,

$$\begin{aligned} \textit{which-var}(A,\langle 3\rangle) &= \texttt{fact}, \\ \textit{which-var}(A,\langle 2,1,2,2,1,2,3,3,4,3,1\rangle) &= \texttt{fact}, \\ \textit{which-var}(A,\langle 2,1,2,2,1,1\rangle) &= \texttt{f}. \end{aligned}$$

However, *which-var*$(A, 1)$ is undefined, since `letrec` is not a variable.

Problem 1. Give a mathematically precise inductive definition of *which-var*. (Or, if you prefer, hand in some Scheme code for *which-var* and a few examples of it running correctly.) ◀

Another way of defining "occurrence" comes from thinking of an expression as a string of characters. (We will consider any nonempty sequence of whitespace to be a single character.) For example:

- the $0^{th}, 8^{th}$, and 9th characters of A are "(";
- the 10th character of A is the "`f`" at the beginning of the first occurrence of the variable `fact` (this is a binding occurrence); and
- if n is the length of the string A, the $(n-5)^{th}$ character is the beginning of the last occurrence of `fact`.

In this *string index* representation, an occurrence *is* a nonegative integer.

Problem 2. Give a mathematically precise inductive definition of the partial function *str→tree-occur* which converts an expression and a string index into a tree-path. For example,

$$\textit{str}{\rightarrow}\textit{tree-occur}(A, 11) = \langle 2, 1, 1 \rangle,$$
$$\textit{str}{\rightarrow}\textit{tree-occur}(A, n-5) = \langle 3 \rangle.$$

It is not our intention that you struggle with the grubby details of parsing, worrying about whitespace, etc. You can make the following assumptions:

- Every input will be well-formed, so you don't need to do error checking. In particular, the input string will be a well-formed expression, with as little whitespace as possible, and the string index will be the index of the first character of some subexpression of the string.
- You are given a function *next-expr* that takes as input an expression in string form and an index into the string. The index should point to a subexpression. *next-expr* parses over the subexpression to find its end, then returns the string index of the very next subexpression in the string. For example, if $s =$ "`(foo (+1 bar) baz)`", then

$$\textit{next-expr}(s, 1) = 5$$
$$\textit{next-expr}(s, 5) = 14$$
$$\textit{next-expr}(s, 6) = 9$$

- You are given a function @ that concatenates sequences. For example,

$$\langle 1,2,3\rangle @\langle 4,5,6\rangle = \langle 1,2,3,4,5,6\rangle$$
$$\langle\rangle @\langle 4,5,6\rangle = \langle 4,5,6\rangle$$

(Alternately, hand in some Scheme code for *str→tree-occur* and a few examples of it running correctly.) ◀

Definition 1. Two expressions M and N are *equivalent* iff $\text{Numval}(C[M]) = \text{Numval}(C[N])$ for all contexts $C[\cdot]$. We write $M \approx N$ when M and N are equivalent.

Problem 3. Does the equivalence 2 ≈ `(+ 1 1)` hold? If the equivalence holds, you do not need to prove that it holds; but if it does not hold, you should give a counterexample.

Problem 4. Find an M and N such that $\text{FreeV}(M) \neq \text{FreeV}(N)$, and $M \approx N$. You do not have to *prove* that $M \approx N$. ◀

The notion of equivalence spelled out above is a natural one. It gives us an easy way of proving that two expressions are *not* equivalent: we simply exhibit a context in which the two terms do not evaluate to the same numeral.

However, proving that an equivalence *holds* is not so easy. We must prove that, when enclosed *by any context at all*, that the two expressions will not evaluate differently.

A powerful result called the Context Lemma can make proofs of equivalence easier. It makes precise our notion that the expressions of the functional kernel of Scheme represent functions. The key idea is that two functions are equal if and only if they return the same results when passed the same arguments.

Definition 2. The *applicative contexts* are defined inductively as follows:

$$\begin{aligned} A[\cdot] &::= [\cdot] \\ &\mid \ (A[\cdot]M_1 \cdots M_n) \end{aligned}$$

where $n \geq 0$ and $M_1, \ldots, M_n$ are expressions.

Let $\text{NBval}(M)$ denote the unique numeral *or boolean*, if any, which is the normal form of $C_{\text{init}}[M]$. $\text{NBval}(M)$ is undefined if M has a normal form which is not a numeral, or if M has no normal form.

We say expressions M and N are *applicatively equivalent* iff for all applicative contexts $A[\cdot]$,

$$\text{NBval}(A[(\texttt{lambda}\ (x_1 \cdots x_n)\ M)]) = \text{NBval}(A[(\texttt{lambda}\ (x_1 \cdots x_n)\ N)]),$$

where $x_1, \ldots, x_n$ are the free variables of M and N.

We write $M \underset{\text{appl}}{\approx} N$ when M and N are applicatively equivalent.

Lemma 1 (Context Lemma). $M \approx N$ iff $M \underset{\text{appl}}{\approx} N$.

To use the Context Lemma we must still consider an infinite number of contexts. But in practice, it turns out to be much simpler to prove equivalences using applicative contexts than using arbitrary contexts.

Problem 5. Give an example showing that the Context Lemma fails if we use Numval instead of NBval in the definition of $\underset{\text{appl}}{\approx}$. ◀

Problem 6. Show that the Context Lemma fails in the language extended with the operator `<<equal?>>`. `<<equal?>>` is defined on page 11 of Handout 6; you should assume that `<<equal?>>` returns `#f` in all cases where it is currently unspecified in Handout 6.

Problem 7. Winskel 4.13. ◀

Problem Set 1 Solutions

The problem set was worth 32 points. The max score was 25. (You don't need a 90% to get an A.)

A number of people had difficulty with the proofs (problems 3,4, and 5). I have tried to comment on common errors in these solutions. If you still have trouble understanding these problems after reading the solutions and your returned problem set, you should see me — these will not be the last proofs we will ask you to do.

Problem 1. [**4 pts**] List the rules that are applied in the substitution model evaluation of `(rec-factorial 5)`. You do not need to list the rules in order, or any rule twice.

Answer: The rules INST, LAM-APP, CONST, and IF are applied (as well as the garbage collection rules).

Problem 2. [**4 pts**] For each of the following, give the first evaluation step at which garbage collection occurs, and the binding that is garbage collected.

1. `(iter-factorial 5)`

   Answer: The first GC occurs between steps 4 and 5. The binding of `n` to 5 is collected.

2. `(y-iter-fact 5)`

   Answer: The first GC occurs between steps 8 and 9. Once again a binding of `n` to 5 is collected.

Problem 3. [**8 pts**] As a function of n, how many steps will it take the each of the following to evaluate the factorial of n in the substitution model?

1. `rec-factorial`

   Answer: The easiest way to do this problem was to examine the sample evaluation and look for patterns. For `(rec-factorial 5)` there was a pattern of "unwinding" that produced a sequence of multiplies, as can be seen in this edited version:

```
 ...
(<<*>> 5 ( ... ))
==[12]==> ...
(<<*>> 5 (<<*>> 4 ( ... )))
==[23]==> ...
(<<*>> 5 (<<*>> 4 (<<*>> 3 ( ... ))))
==[34]==> ...
```

Thus there is a loop of 11 steps for each n.

Moreover, after all the unwindings we end up with the expression

```
(<<*>> 5 (<<*>> 4 (<<*>> 3 (<<*>> 2 (<<*>> 1 1))))).
```

So the unwindings are followed by n multiplies. (Many people missed this.)

Finally, we note that it would take 6 steps to evaluate

```
(rec-factorial 0).
```

This can be seen by just replacing 5 by 0 in the sample evaluation, and seeing where the evaluation would change.

Thus `rec-factorial` calculates $n!$ in exactly $12n + 6$ steps. As a sanity check, we note that for $n = 5$ it should take 66 steps, just as in the sample evaluation.

The others can be calculated similarly; note that the final n multiplies occur only for `rec-factorial` and `y-fact`, and not for the iterative versions.

2. `iter-factorial`

   Answer: $13n + 10$.

3. `y-fact`

   Answer: $21n + 9$.

4. `y-iter-fact`

   Answer: $23n + 17$.

Problem 4. **[4 pts]** Winskel, Exercise 3.4: Prove by structural induction that the evaluation of arithmetic expressions always terminates.

Answer: We must prove, for any arithmetic expression a and state σ, that there exists some number m such that $\langle a, \sigma \rangle \rightarrow m$. We proceed by induction according to the structure of a:

$a \equiv n$: then for any σ, $\langle n, \sigma \rangle \rightarrow n$ by the evaluation rule for numbers. So let $m = n$.

$a \equiv a_0 + a_1$: by induction we have for any σ, there exist m_0 and m_1 such that $\langle a_0, \sigma \rangle \rightarrow m_0$ and $\langle a_1, \sigma \rangle \rightarrow m_1$. Then by the evaluation rule for sums, we have $\langle a, \sigma \rangle \rightarrow m$, where m is the sum of m_0 and m_1.

The cases $a \equiv a_0 - a_1$ and $a \equiv a_0 \times a_1$ follow similarly.

$a \equiv X$: then for any σ, $\langle X, \sigma \rangle \rightarrow \sigma(X)$ by the evaluation rule for locations. So let $m = \sigma(X)$. ■

Notes: What to put in a proof and how much to leave out are matters of convention and common sense. I give below the factors I considered in deciding what level of detail I would use in my proof.

I have not shown every case in detail. The cases $a \equiv a_0 - a_1$ and $a \equiv a_0 \times a_1$ are so similar to the case $a \equiv a_0 + a_1$ that including them would just mean copying the proof for $a \equiv a_0 + a_1$. It makes sense to leave them out. It is important, however, to say which cases are left out, and why: this assures the reader that you have thought about those cases, that the proof goes through for them, and indicates how the reader might do the proof himself. Otherwise, the reader might conclude that you forgot about those cases, and maybe that the proof does not even go through for them.

I did include the case $a \equiv X$, which some people omitted. The evaluation of $\langle X, \sigma \rangle$ is different from any other rule: it is the only one that uses σ. Thus the proof for this case is different from any other, and should be included.

I could have spelled out the induction hypothesis: "We will prove, for all arithmetic expressions a, that $P(a)$ holds, where

$$P(a) = \forall \sigma . \exists m . \langle a, \sigma \rangle \rightarrow m\text{''}.$$

Instead, I used a standard mathematical convention: I stated in English the property that I wished to prove, and the fact that I was going to prove it by induction. I provided enough information for the reader to reconstruct $P(a)$ for himself if desired.

It wouldn't be wrong to spell out $P(a)$ in your proof, but if you do so, you must get it right. A common mistake was to give an induction hypothesis with incorrect quantifiers. For instance in

$$P_{\mathbf{wrong1}}(a) = \langle a, \sigma \rangle \rightarrow m,$$

(no quantifiers given) it is clear that a is a parameter of $P_{\mathbf{wrong1}}$, but what about σ and m? If we guess that they should be quantified, what clue do we have that it should be $\forall \sigma$ and $\exists m$? Why not $\exists \sigma$ and $\forall m$?

Some people incorrectly quantified a in $P(a)$, for example,

$$P_{\mathrm{wrong2}}(a) = \forall a, \sigma . \exists m . \langle a, \sigma \rangle \rightarrow m.$$

The problem is that $\forall$ is a binding operator, just as `lambda` is a binding operator in Scheme. It is a declaration of a new variable, distinct from all other variables, including other variables of the same name. Thus the a introduced by the $\forall$ is distinct from the a appearing as a parameter in $P_{\mathrm{wrong2}}(a)$.

Note that the property $P_1(a)$ = "evaluation of a always terminates" is different from the property $P_2(a)$ = "evaluation of a is deterministic". $P_2(a)$ says that *if* $\langle a, \sigma \rangle$ evaluates to m and m', *then* $m = m'$. If P_2 holds of an arithmetic expression a, it *does not* mean that $\langle a, \sigma \rangle$ evaluates to some m. It could be that for all σ, $\langle a, \sigma \rangle$ evaluates to no m, in which case $P_2(a)$ holds, but $P_1(a)$ does not. Some people seemed to think that $P_2(a)$ implied $P_1(a)$.

Problem 5. **[8 pts]** Winskel, Exercise 3.5: Show that the evaluation of boolean expressions is firstly deterministic, and secondly total.

Answer: First we prove that for all b, σ, t_0, and t_1, if $\langle b, \sigma \rangle \rightarrow t_0, t_1$, then $t_0 = t_1$. We proceed by structural induction on b.

$b \equiv$ **true** or $b \equiv$ **false**: then only one rule applies to $\langle b, \sigma \rangle$, namely,

$$\langle b, \sigma \rangle \rightarrow b.$$

So if $\langle b, \sigma \rangle \rightarrow t_0, t_1$, then $t_0 = t_1 = b$.

$b \equiv \neg b'$: suppose $\langle b, \sigma \rangle \rightarrow t_0, t_1$. Then by the single evaluation rule for $\neg$, we must have $\langle b', \sigma \rangle \rightarrow t_0', t_1'$, where t_0' is the negation of t_0, and t_1' is the negation of t_1. But by induction we must have $t_0' = t_1'$, and therefore $t_0 = t_1$.

The cases $b \equiv b_0 \vee b_1$ and $b \equiv b_0 \wedge b_1$ are proved similarly.

$b \equiv a_0 = a_1$: suppose $\langle b, \sigma \rangle \rightarrow t_0, t_1$. Then by the single evaluation rule for $=$, we must have $\langle a_0, \sigma \rangle \rightarrow m_0, n_0$ and $\langle a_1, \sigma \rangle \rightarrow m_1, n_1$, where t_0 iff $m_0 = m_1$, and t_1 iff $n_0 = n_1$. But by Proposition 3.3 of Winskel, we must have $m_0 = n_0$, and $m_1 = n_1$. Thus $t_0 = t_1$.

The case $b \equiv a_0 \leq a_1$ is handled similarly. ■

Now we prove, for any boolean expression b and state σ, that there exists some t such that $\langle b, \sigma \rangle \rightarrow t$. Once again the proof is by structural induction on b.

$b \equiv$ **true** or $b \equiv$ **false**: then $\langle b, \sigma \rangle \rightarrow b$.

$b \equiv \neg b'$: by induction we have, for any σ, that there is a t' such that $\langle b', \sigma \rangle \rightarrow t'$. And then the rule for $\neg$ can be applied: $\langle b, \sigma \rangle \rightarrow t$, where t is the negation of t'.

The cases $b \equiv b_0 \vee b_1$ and $b \equiv b_0 \wedge b_1$ are proved similarly.

$b \equiv a_0 = a_1$: by the Problem 4, for any σ there exist m_0 and m_1 such that $\langle a_0, \sigma\rangle \rightarrow m_0$ and $\langle a_1, \sigma\rangle \rightarrow m_1$. Then we can apply the rule for $=$ to get $\langle b, \sigma\rangle \rightarrow t$, where t iff $m_0 = m_1$.

The case $b \equiv a_0 \leq a_1$ is handled similarly. ■

Notes: For the curious, let me spell out the induction hypotheses:

$$P_1(b) = \forall \sigma, t_0, t_1.\ \text{if } \langle b, \sigma\rangle \rightarrow t_0, t_1, \text{ then } t_0 = t_1$$

for the first part, and

$$P_2(b) = \forall \sigma. \exists t. \langle b, \sigma\rangle \rightarrow t$$

for the second.

Note that I am using b as the parameter of the properties P_1 and P_2, rather than a as in problem 4. This simply follows our convention (Winskel, p. 12) that a will be used to stand for arithmetic expressions, and b will be used for boolean expressions. A number of people ignored this and used a for boolean expressions, m and n for boolean values, etc. This will confuse any reader who is used to the conventions.

Also, I used both Proposition 3.3, from Winskel, and the result of Problem 4 in this proof. From Winskel's statement of the problem, it was already clear that this would be necessary. But note that I was careful to say exactly where I used the results. It is important to "flag", for the reader, places where previous, non-trivial results are being used.

Problem 6. **[4 pts]** Winskel, Exercise 3.6: What goes wrong when you try to prove the execution of commands is deterministic by using structural induction on commands?

Answer: Just about everyone got this wrong.

The correct answer is most easily seen by trying to do the proof. First, we must decide what is meant by "execution of commands is deterministic". Since commands "evaluate" to states, we guess that the induction hypothesis should be:

$$P(c) = \forall \sigma, \sigma_1, \sigma_2.\ \text{if } \langle c, \sigma\rangle \rightarrow \sigma_1, \sigma_2, \text{ then } \sigma_1 = \sigma_2.$$

So we try to prove $\forall c. P(c)$ by induction on the structure of commands. All of the cases go through easily, *except* the **while** case.

Suppose $c \equiv$ **while** b **do** c', and $\langle c, \sigma\rangle \rightarrow \sigma_1, \sigma_2$.

First we consider the ways in which

$$\langle c, \sigma \rangle \to \sigma_1, \tag{1}$$

$$\langle c, \sigma \rangle \to \sigma_2. \tag{2}$$

There are two rules for **while** (this is the first time we have run up against this). By Problem 5, we know that the evaluation of boolean expressions is total and deterministic. So either $\langle b, \sigma \rangle \to \textbf{false}$, or $\langle b, \sigma \rangle \to \textbf{true}$. In the first case, both (1) and (2) must have followed by the first rule for **while**. We will encounter no problems with this case.

But if $\langle b, \sigma \rangle \to \textbf{true}$, then both (1) and (2) must follow by the second **while** rule:

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c', \sigma \rangle \to \sigma_1' \quad \langle \textbf{while } b \textbf{ do } c', \sigma_1' \rangle \to \sigma_1}{\langle \textbf{while } b \textbf{ do } c', \sigma \rangle \to \sigma_1}$$

and

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c', \sigma \rangle \to \sigma_2' \quad \langle \textbf{while } b \textbf{ do } c', \sigma_2' \rangle \to \sigma_2}{\langle \textbf{while } b \textbf{ do } c', \sigma \rangle \to \sigma_2}$$

We can now try to apply the induction hypothesis. First, note that

$$\langle c', \sigma \rangle \to \sigma_1', \sigma_2'.$$

By induction on c' we can conclude that $\sigma_1' = \sigma_2'$. So let $\sigma' =_{\mathrm{def}} \sigma_1' = \sigma_2'$.

Then we have $\langle \textbf{while } b \textbf{ do } c', \sigma' \rangle \to \sigma_1, \sigma_2$. Can we now use induction to finish the proof?

The answer is no. Recall that in a proof by induction, we prove that the induction hypothesis holds of an element by assuming that it holds *for all smaller elements*. In this case, "smaller" means "subexpression". Thus we were able to assume that the induction hypothesis held of c', because c' is a subexpression of c. But we *cannot* assume that it holds of **while** b **do** c', because this is not a subexpression of c; it is c itself.

Note: A number of people thought the proof would fail because there are expressions using **while** that do not terminate. For example, for any σ, there is no σ' such that

$$\langle \textbf{while true do skip}, \sigma \rangle \to \sigma'.$$

But as I noted in the solution to Problem 4, "terminates" and "deterministic" are two very different properties. A command that never terminates is, in fact, deterministic: it never terminates in two different states. Thus non-termination does not imply non-determinism.

Instructions for Problem Sets

1 Form of Solutions

Each problem is to be done on a *separate sheet* of *three-hole punched paper.* If a problem requires more than one sheet, staple these sheets together, but keep each problem separate. Do not use red ink. Mark the top of the paper with:

- your name,
- "6.044J/18.423J",
- the assignment number,
- the problem number, and
- the date.

Try to be as clear and precise as possible in your presentations. Problem grades are based not only on getting the right answer or otherwise demonstrating that you understand how a solution goes, but also on your ability to explain the solution or proof in a way helpful to a reader.

If you have doubts about the way your homework has been graded, first see the TA. Other questions and suggestions will be welcomed by both the instructor and the TA.

Problem sets will be collected at the beginning of class; graded problem sets will be returned at the end of class. Solutions will generally be available with the graded problem sets, one week after their submission.

2 Collaboration and References

You must write your own problem solutions and other assigned course work in your own words and entirely alone. On the other hand, you are encouraged to discuss the problems with one or two classmates before you write your solutions. If you do so, please be sure to

> *indicate the members of your discussion group*

on your solution.

Similarly, you are welcome to use other texts and references in doing homework, but if you find that a solution to an assigned problem has been given in such a reference, you should nevertheless rewrite the solution in your own words and *cite your source.*

3 Late Policy

Late homeworks should be submitted to the TA. If they can be graded without inconvenience, they will be. Late homeworks that are not graded will be kept for reference until after the final. No homework will be accepted after the solutions have been given out.

One-step vs. "Natural" Semantics

The operational behavior of the language **IMP** is described in Winskel using inductive rules for deriving *evaluation assertions* of the form $\langle c, \sigma \rangle \rightarrow \sigma'$. This style of semantics has been called "natural" because it resembles so-called "natural deduction" inference rules of formal logic.

One can also give a term-rewriting style semantics for **IMP** using inductive rules for deriving *one-step*-rewriting assertions of the form $\langle c, \sigma \rangle \rightarrow_1 \sigma'$. The two styles of semantical specification are equivalent in the following precise sense:

Theorem. For all $c \in \mathbf{Com}$ and $\sigma, \sigma' \in \Sigma$,

$$\langle c, \sigma \rangle \rightarrow_1^* \sigma' \text{ iff } \langle c, \sigma \rangle \rightarrow \sigma'.$$

In the rest of this handout, we define the one-step semantics and then prove the above Theorem. We use **op** to range over syntactic operator symbols, and *op* to range over corresponding arithmetic or Boolean operations.

1 One-Step Rules for Arithmetic Expressions, Aexp

$$\langle X, \sigma \rangle \rightarrow_1 \langle \sigma(X), \sigma \rangle$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow_1 \langle a_0', \sigma \rangle}{\langle a_0 \ \mathbf{op}\ a_1, \sigma \rangle \rightarrow_1 \langle a_0' \ \mathbf{op}\ a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow_1 \langle a_1', \sigma \rangle}{\langle n \ \mathbf{op}\ a_1, \sigma \rangle \rightarrow_1 \langle n \ \mathbf{op}\ a_1', \sigma \rangle}$$

$$\langle n \ \mathbf{op}\ m, \sigma \rangle \rightarrow_1 \langle n \ op\ m, \sigma \rangle$$

op	*op*
$+$	the sum function
$-$	the subtraction function
$\times$	the multiplication function

Notice that

$$\langle 5+7, \sigma \rangle \rightarrow_1 \langle 12, \sigma \rangle$$

is an instance of the rule $\langle n \ \mathbf{op}\ m, \sigma \rangle \rightarrow_1 \langle n \ op\ m, \sigma \rangle$, but that

$$\langle 5+7, \sigma \rangle \rightarrow_1 \langle 5+7, \sigma \rangle$$

is *not* derivable at all.

2 One-Step Rules for Boolean Expressions, Bexp

$$\frac{\langle a_0, \sigma\rangle \rightarrow_1 \langle a_0', \sigma\rangle}{\langle a_0 \ \mathbf{op}\ a_1, \sigma\rangle \rightarrow_1 \langle a_0' \ \mathbf{op}\ a_1, \sigma\rangle}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow_1 \langle a_1', \sigma\rangle}{\langle n \ \mathbf{op}\ a_1, \sigma\rangle \rightarrow_1 \langle n \ \mathbf{op}\ a_1', \sigma\rangle}$$

$$\langle n \ \mathbf{op}\ m, \sigma\rangle \rightarrow_1 \langle n \ op\ m, \sigma\rangle$$

op	*op*
=	the equality predicate
≤	the less than or equal to predicate

We next have the rules for Boolean negation:

$$\frac{\langle b, \sigma\rangle \rightarrow_1 \langle b', \sigma\rangle}{\langle \neg b, \sigma\rangle \rightarrow_1 \langle \neg b', \sigma\rangle}$$

$$\langle \neg\, \mathbf{true}, \sigma\rangle \rightarrow_1 \langle \mathbf{false}, \sigma\rangle$$

$$\langle \neg\, \mathbf{false}, \sigma\rangle \rightarrow_1 \langle \mathbf{true}, \sigma\rangle$$

Finally we have the rules for binary Boolean operators. We use **op** and *op* to range over the symbols and functions in the chart following the rules. We let $t, t_0, t_1, \ldots$ range over the set $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$.

$$\frac{\langle b_0, \sigma\rangle \rightarrow_1 \langle b_0', \sigma\rangle}{\langle b_0 \ \mathbf{op}\ b_1, \sigma\rangle \rightarrow_1 \langle b_0' \ \mathbf{op}\ b_1, \sigma\rangle}$$

$$\frac{\langle b_1, \sigma\rangle \rightarrow_1 \langle b_1', \sigma\rangle}{\langle t_0 \ \mathbf{op}\ b_1, \sigma\rangle \rightarrow_1 \langle t_0 \ \mathbf{op}\ b_1', \sigma\rangle}$$

$$\langle t_0 \ \mathbf{op}\ t_1, \sigma\rangle \rightarrow_1 \langle t_0 \ op\ t_1, \sigma\rangle$$

op	*op*
∧	the conjunction operation (Boolean AND)
∨	the disjunction operation (Boolean OR)

3 One-Step Rules for Commands, Com

Atomic Commands:

$$\langle \mathbf{skip}, \sigma \rangle \rightarrow_1 \sigma$$

$$\frac{\langle a, \sigma \rangle \rightarrow_1 \langle a', \sigma \rangle}{\langle X := a, \sigma \rangle \rightarrow_1 \langle X := a', \sigma \rangle}$$

$$\langle X := n, \sigma \rangle \rightarrow_1 \sigma[n/X]$$

Sequencing:

$$\frac{\langle c_0, \sigma \rangle \rightarrow_1 \langle c_0', \sigma' \rangle}{\langle (c_0; c_1), \sigma \rangle \rightarrow_1 \langle (c_0'; c_1), \sigma' \rangle}$$

$$\frac{\langle c_0, \sigma \rangle \rightarrow_1 \sigma'}{\langle (c_0; c_1), \sigma \rangle \rightarrow_1 \langle c_1, \sigma' \rangle}.$$

Conditionals:

$$\frac{\langle b, \sigma \rangle \rightarrow_1 \langle b', \sigma \rangle}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma \rangle \rightarrow_1 \langle \mathbf{if}\, b'\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma \rangle}$$

$$\langle \mathbf{if\ true\ then}\, c_0\, \mathbf{else}\, c_1, \sigma \rangle \rightarrow_1 \langle c_0, \sigma \rangle$$

$$\langle \mathbf{if\ false\ then}\, c_0\, \mathbf{else}\, c_1, \sigma \rangle \rightarrow_1 \langle c_1, \sigma \rangle$$

While-loops:

$$\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma \rangle \rightarrow_1 \langle \mathbf{if}\, b\, \mathbf{then} (c; \mathbf{while}\, b\, \mathbf{do}\, c)\, \mathbf{else\ skip}, \sigma \rangle$$

4 Equivalence of One-Step and Natural Semantics

We now prove

Theorem. For all $c \in \mathbf{Com}$ and $\sigma, \sigma' \in \Sigma$,

$$\langle c, \sigma \rangle \rightarrow_1^* \sigma' \text{ iff } \langle c, \sigma \rangle \rightarrow \sigma'.$$

The proof proceeds in three stages:

(A) Prove $\langle a, \sigma\rangle \to_1^* \langle n, \sigma\rangle$ iff $\langle a, \sigma\rangle \to n$;

(B) Prove $\langle b, \sigma\rangle \to_1^* \langle t, \sigma\rangle$ iff $\langle b, \sigma\rangle \to t$, using **(A)**;

(C) Prove $\langle c, \sigma\rangle \to_1^* \sigma'$ iff $\langle c, \sigma\rangle \to \sigma'$, using **(A)** and **(B)**.

The proofs of all three parts are similar, except that where structural induction serves to prove a version of Lemma 1 below for expressions, induction on derivations is needed to prove it for commands. We will henceforth assume that **(A)** and **(B)** are proven, and present **(C)** only.

We first prove **(C)** from left to right, namely

$$\langle c, \sigma\rangle \to_1^* \sigma' \quad \Rightarrow \quad \langle c, \sigma\rangle \to \sigma'.$$

The proof is based on

Lemma 1. If $\langle c, \sigma\rangle \to_1 \langle c', \sigma''\rangle$ and $\langle c', \sigma''\rangle \to \sigma'$, then $\langle c, \sigma\rangle \to \sigma'$.

Assuming this lemma, a simple induction on the definition of the transitive closure, $\to_1{}^*$, of the one-step evaluation relation, $\to_1$, completes the proof of **(C)** $\Rightarrow$:

Base case: Suppose $\langle c, \sigma\rangle \to_1^* \sigma'$ because $\langle c, \sigma\rangle \to_1 \sigma'$. Then either c is **skip** and σ' is σ, or c is $X := n$ and σ' is $\sigma[n/X]$. In either case, $\langle c, \sigma\rangle \to \sigma'$ follows immediately by a natural evaluation axiom.

Induction: Suppose $\langle c, \sigma\rangle \to_1^* \sigma'$ because $\langle c, \sigma\rangle \to_1 \langle c', \sigma''\rangle$ for some c', σ'' such that $\langle c', \sigma''\rangle \to_1^* \sigma'$. Then by induction, we have that $\langle c', \sigma''\rangle \to \sigma'$. But now Lemma 1 immediately implies $\langle c, \sigma\rangle \to \sigma'$, as required.

So we need only prove Lemma 1, which we do by induction on the derivation of $\langle c', \sigma''\rangle \to \sigma'$. The induction breaks into cases according to the form of c.

First, we note that c cannot be either **skip** or $X := n$ where $n \in \mathbf{N}$, because in both these cases, $\langle c, \sigma\rangle$ moves in one step to a state rather than a command configuration $\langle c', \sigma''\rangle$, and the condition of the lemma is not satisfied. The other cases are:

•[c is of the form $X := a$ where $a \notin$ **Num**] There is a unique one-step rule by which $\langle X := a, \sigma\rangle \to_1 \langle c', \sigma''\rangle$ could be derived, and by this rule c' must be $X := a'$ where $\langle a, \sigma\rangle \to_1 \langle a', \sigma\rangle$, and σ'' must be σ. We are given that $\langle c', \sigma''\rangle \to \sigma'$, and there is only one possible derivation with this conclusion:

$$\frac{\langle a', \sigma\rangle \to n}{\langle X := a', \sigma\rangle \to \sigma[n/X]}$$

We conclude that

$$\sigma' \text{ must be } \sigma[n/X] \text{ for some } n \text{ such that } \langle a', \sigma\rangle \to n.$$

But $\langle a', \sigma\rangle \to n$ implies $\langle a', \sigma\rangle \to_1^* \langle n, \sigma\rangle$, by **(A)**. Thus, we have

$$\langle a, \sigma\rangle \to_1 \langle a', \sigma\rangle \to_1{}^* \langle n, \sigma\rangle.$$

By **(A)** again, we conclude that $\langle a, \sigma\rangle \to n$. Now the assignment rule for natural evaluation gives the derivation

$$\frac{\langle a, \sigma\rangle \to n}{\langle X := a, \sigma\rangle \to \sigma[n/X]}$$

So indeed, $\langle c, \sigma\rangle \to \sigma'$.

•[c is $(c_0; c_1)$] There are two ways that the given condition $\langle c, \sigma\rangle \to_1 \langle c', \sigma''\rangle$ could be derived, namely,

[$\langle c_0, \sigma\rangle \to_1 \sigma''$ and c' is c_1] Since $\langle c_0, \sigma\rangle \to_1 \sigma''$, the rules for $\to_1$ imply that either c_0 must be **skip** and $\sigma'' = \sigma$, or else c_0 is $X := n$ and $\sigma'' = \sigma[n/X]$. So

$$\frac{\langle \textbf{skip}, \sigma\rangle \to \sigma, \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle (\textbf{skip}; c_1), \sigma\rangle \to \sigma'}$$

or

$$\frac{\langle X := n, \sigma\rangle \to \sigma[n/X], \quad \langle c_1, \sigma[n/X]\rangle \to \sigma'}{\langle (X := n; c_1), \sigma\rangle \to \sigma'}$$

is a derivation of $\langle c, \sigma\rangle \to \sigma'$, as required.

[$\langle c_0, \sigma\rangle \to_1 \langle c_0', \sigma''\rangle$ and c' is $(c_0'; c_1)$] We have

$$d \Vdash \langle (c_0'; c_1), \sigma''\rangle \to \sigma',$$

for some d which must be of the form

$$\frac{\begin{matrix}\vdots & & \vdots \\ \langle c_0', \sigma''\rangle \to \sigma''' & & \langle c_1, \sigma'''\rangle \to \sigma'\end{matrix}}{\langle (c_0'; c_1), \sigma''\rangle \to \sigma'}$$

Thus we have a derivation of $\langle c_0', \sigma''\rangle \to \sigma'''$. This derivation is a subderivation of d, and we are given that $\langle c_0, \sigma\rangle \to_1 \langle c_0', \sigma''\rangle$, so by induction we deduce

$$\langle c_0, \sigma\rangle \to \sigma'''.$$

But that gives

$$\frac{\langle c_0, \sigma\rangle \to \sigma''', \quad \langle c_1, \sigma'''\rangle \to \sigma'}{\langle c, \sigma\rangle \to \sigma'}$$

•[c is **if** b **then** c_0 **else** c_1] There are two subcases:

[$b \in \mathbf{T}$ and c' is c_i] Then we have

$$\langle c, \sigma \rangle \rightarrow_1 \langle c_i, \sigma \rangle \quad \text{and} \quad \langle c_i, \sigma \rangle \rightarrow \sigma',$$

and we get the derivation

$$\frac{\langle b, \sigma \rangle \rightarrow b, \quad \langle c_i, \sigma \rangle \rightarrow \sigma'}{\langle c, \sigma \rangle \rightarrow \sigma'}$$

[$b \notin \mathbf{T}$] Then we have derivations

$$\frac{\langle b, \sigma \rangle \rightarrow_1 \langle b', \sigma \rangle}{\langle c, \sigma \rangle \rightarrow_1 \langle \mathbf{if}\ b'\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle}$$

and

$$\frac{\langle b', \sigma \rangle \rightarrow t, \quad \langle c_i, \sigma \rangle \rightarrow \sigma'}{\langle \mathbf{if}\ b'\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \rightarrow \sigma'}$$

for some $t \in \mathbf{T}$ and $i \in \{0, 1\}$. By **(B)**, $\langle b', \sigma \rangle \rightarrow t$ implies $\langle b', \sigma \rangle \rightarrow_1^* \langle t, \sigma \rangle$, so

$$\langle b, \sigma \rangle \rightarrow_1 \langle b', \sigma \rangle \rightarrow_1{}^* \langle t, \sigma \rangle$$

which, again by **(B)**, implies $\langle b, \sigma \rangle \rightarrow t$. So we have

$$\frac{\langle b, \sigma \rangle \rightarrow t, \quad \langle c_i, \sigma \rangle \rightarrow \sigma'}{\langle c, \sigma \rangle \rightarrow \sigma'}$$

•[c is **while** b **do** c''] There is a unique one-step derivation

$$\langle c, \sigma \rangle \rightarrow_1 \langle \mathbf{if}\ b\ \mathbf{then} (c''; c)\ \mathbf{else}\ \mathbf{skip}, \sigma \rangle,$$

so c' is **if** b **then**$(c''; c)$ **else skip** and σ'' is σ. But we know (Winskel Prop. 2.8) that $c \sim c'$, and are given that $\langle c', \sigma \rangle \rightarrow \sigma'$, so we conclude $\langle c, \sigma \rangle \rightarrow \sigma'$.

This completes the proof of Lemma 1. ■

It remains to prove the converse implication:

$$\langle c, \sigma \rangle \rightarrow \sigma' \quad \Rightarrow \quad \langle c, \sigma \rangle \rightarrow_1^* \sigma'.$$

The proof requires some simple facts about $\rightarrow_1{}^*$.

Lemma 2.

1. $\langle a,\sigma\rangle \rightarrow_1^* \langle a',\sigma'\rangle \quad \Rightarrow \quad \langle X := a,\sigma\rangle \rightarrow_1^* \langle X := a',\sigma'\rangle$,
2. $\langle b,\sigma\rangle \rightarrow_1^* \langle b',\sigma'\rangle \;\Rightarrow\; \langle \textbf{if}\, b\, \textbf{then}\, c_0\, \textbf{else}\, c_1,\sigma\rangle \rightarrow_1^* \langle \textbf{if}\, b'\, \textbf{then}\, c_0\, \textbf{else}\, c_1,\sigma'\rangle$,
3. $[\langle c_0,\sigma\rangle \rightarrow_1^* \sigma'' \;\&\; \langle c_1,\sigma''\rangle \rightarrow_1^* \sigma'] \quad \Rightarrow \quad \langle (c_0;c_1),\sigma\rangle \rightarrow_1^* \sigma'$.

Proof: All three parts follow by straightforward induction on the definition of the transitive closure, $\rightarrow_1{}^*$, of $\rightarrow_1$. We omit 1 and 2, and do only the slightly more complicated part 3. In particular, we prove 3 by induction on the derivation of $\langle c_0,\sigma\rangle \rightarrow_1^* \sigma''$.

Such a derivation must consist of $\langle c_0,\sigma\rangle \rightarrow_1 \gamma$ and a derivation of $\gamma \rightarrow_1{}^* \sigma''$ for some state or configuration γ. So there are two cases:

[γ is a state σ''] The result then holds easily, with $\langle c_0,\sigma\rangle \rightarrow_1 \sigma''$ implying that $\langle (c_0;c_1),\sigma\rangle \rightarrow_1 \langle c_1,\sigma''\rangle$. Combining this with our original presumption of $\langle c_1,\sigma''\rangle \rightarrow_1^* \sigma'$ gives the desired result.

[γ is a configuration $\langle c_0',\sigma_0\rangle$] Then, we can apply the induction hypothesis to the evaluation $\langle c_0',\sigma_0\rangle \rightarrow_1^* \sigma''$, and we conclude that $\langle (c_0';c_1),\sigma_0\rangle \rightarrow_1^* \sigma'$. But since $\langle c_0,\sigma\rangle \rightarrow_1 \langle c_0',\sigma_0\rangle$, we have by the definition of $\rightarrow_1$ that

$$\langle (c_0;c_1),\sigma\rangle \rightarrow_1 \langle (c_0';c_1),\sigma_0\rangle \rightarrow_1^* \sigma'.$$

So $\langle (c_0;c_1),\sigma\rangle \rightarrow_1^* \sigma'$ as required.

■

Proof: We now prove the converse implication of **(C)** by induction on the derivation d of $\langle c,\sigma\rangle \rightarrow \sigma'$. The induction breaks into cases according to the form of c.

•[c is **skip**] By the definition of $\rightarrow_1$ we have $\langle \textbf{skip},\sigma\rangle \rightarrow_1 \sigma$, so $\sigma' = \sigma$, and hence $\langle \textbf{skip},\sigma\rangle \rightarrow_1^* \sigma'$, as required.

•[c is of the form $X := a$] The derivation of $\langle c,\sigma\rangle \rightarrow \sigma'$ must be of the form

$$\frac{\begin{array}{c}\vdots\\ \langle a,\sigma\rangle \rightarrow n\end{array}}{\langle X := a,\sigma\rangle \rightarrow \sigma[n/X]}$$

so σ' is $\sigma[n/X]$. By **(A)**, $\langle a,\sigma\rangle \rightarrow n$ implies

$$\langle a,\sigma\rangle \rightarrow_1^* \langle n,\sigma\rangle.$$

But then by Lemma 2.1

$$\langle X := a, \sigma \rangle \to_1^* \langle X := n, \sigma \rangle.$$

Now $\langle X := n, \sigma \rangle \to_1 \sigma[n/X]$ is an $\to_1$ rule, so we have

$$\langle X := a, \sigma \rangle \to_1^* \sigma[n/X] = \sigma'.$$

•[c is $(c_0; c_1)$] Then d must be of the form

$$\frac{\begin{array}{cc} \vdots & \vdots \\ \langle c_0, \sigma \rangle \to \sigma'' & \langle c_1, \sigma'' \rangle \to \sigma' \end{array}}{\langle (c_0; c_1), \sigma \rangle \to \sigma'}$$

Let d_0 and d_1 be the left and right subderivations above. Then, by induction we have $\langle c_0, \sigma \rangle \to_1^* \sigma''$ and $\langle c_1, \sigma'' \rangle \to_1^* \sigma'$. Now by Lemma 2.3 we conclude

$$\langle (c_0; c_1), \sigma \rangle \to_1^* \sigma'.$$

•[c is $\textbf{if}\, b \,\textbf{then}\, c_0 \,\textbf{else}\, c_1$] The derivation d is of the form

$$\frac{\begin{array}{cc} \vdots & \vdots \\ \langle b, \sigma \rangle \to t & \langle c_i, \sigma \rangle \to \sigma' \end{array}}{\langle c, \sigma \rangle \to \sigma'}$$

for $t \in \mathbf{T}$. Let us call the right-hand subderivation d'. Since $\langle b, \sigma \rangle \to t$, we know by **(B)** that $\langle b, \sigma \rangle \to_1^* \langle t, \sigma \rangle$, and hence by Lemma 2.2

$$\langle \textbf{if}\, b \,\textbf{then}\, c_0 \,\textbf{else}\, c_1, \sigma \rangle \to_1^* \langle \textbf{if}\, t \,\textbf{then}\, c_0 \,\textbf{else}\, c_1, \sigma \rangle.$$

Then, by the induction hypothesis (applied to d') we have $\langle c_i, \sigma \rangle \to_1^* \sigma'$, so

$$\begin{aligned} \langle \textbf{if}\, b \,\textbf{then}\, c_0 \,\textbf{else}\, c_1, \sigma \rangle & \to_1{}^* \langle \textbf{if}\, t \,\textbf{then}\, c_0 \,\textbf{else}\, c_1, \sigma \rangle \\ & \to_1 \langle c_i, \sigma \rangle \\ & \to_1{}^* \sigma' \end{aligned}$$

•[c is $\textbf{while}\, b \,\textbf{do}\, c'$] The simpler subcase is when d is of the form

$$\frac{\begin{array}{c} \vdots \\ \langle b, \sigma \rangle \to \textbf{false} \end{array}}{\langle \textbf{while}\, b \,\textbf{do}\, c', \sigma \rangle \to \sigma}$$

Then $\sigma' = \sigma$, and, by (**B**), we have

$$\langle b, \sigma \rangle \rightarrow_1^* \langle \textbf{false}, \sigma \rangle,$$

so by Lemma 2.2

$$\langle \textbf{if}\, b\, \textbf{then}(c'; c)\, \textbf{else}\, \textbf{skip}, \sigma \rangle \rightarrow_1^* \langle \textbf{if}\, \textbf{false}\, \textbf{then}(c'; c)\, \textbf{else}\, \textbf{skip}, \sigma \rangle.$$

Thus,

$$\begin{aligned} \langle \textbf{while}\, b\, \textbf{do}\, c', \sigma \rangle &\rightarrow_1 \langle \textbf{if}\, b\, \textbf{then}(c'; c)\, \textbf{else}\, \textbf{skip}, \sigma \rangle \\ &\rightarrow_1{}^* \langle \textbf{if}\, \textbf{false}\, \textbf{then}(c'; c)\, \textbf{else}\, \textbf{skip}, \sigma \rangle \\ &\rightarrow_1 \langle \textbf{skip}, \sigma \rangle \\ &\rightarrow_1 \sigma. \end{aligned}$$

The other subcase is when d is of the form

$$\frac{\begin{array}{ccc} \vdots & \vdots & \vdots \\ \langle b, \sigma \rangle \rightarrow \textbf{true} & \langle c', \sigma \rangle \rightarrow \sigma'' & \langle c, \sigma'' \rangle \rightarrow \sigma' \end{array}}{\langle c, \sigma \rangle \rightarrow \sigma'}$$

As in the previous cases, from the subderivations we can conclude

$$\begin{aligned} &\langle b, \sigma \rangle \rightarrow_1{}^* \langle \textbf{true}, \sigma \rangle \\ &\langle c', \sigma \rangle \rightarrow_1{}^* \sigma'' \\ &\langle c, \sigma'' \rangle \rightarrow_1{}^* \sigma \end{aligned}$$

by (**B**) and induction hypothesis. Then, by Lemma 2.3, we can conclude

$$\langle (c'; c), \sigma \rangle \rightarrow_1^* \sigma'.$$

Putting the pieces together with the definition of $\rightarrow_1$, this gives us

$$\begin{aligned} \langle c, \sigma \rangle &\rightarrow_1 \langle \textbf{if}\, b\, \textbf{then}(c'; c)\, \textbf{else}\, \textbf{skip}, \sigma \rangle \\ &\rightarrow_1{}^* \langle \textbf{if}\, \textbf{true}\, \textbf{then}(c'; c)\, \textbf{else}\, \textbf{skip}, \sigma \rangle \\ &\rightarrow_1 \langle (c'; c), \sigma \rangle \\ &\rightarrow_1{}^* \sigma', \end{aligned}$$

thus proving the final case in the theorem.

■

Problem Set 3

Due: 13 October 1993.

QUIZ 1: In class, closed book, Friday, October 15.

Problem 1. Consider the following proposed definition of a possibly partial function $f : \mathbf{N} \to \mathbf{N}$:

$$f(0) = 1$$
$$f(1) = 2$$
$$f(f(n)) = (1 + f(n+2))^2$$

In this problem we examine whether this is a proper definition of a function f.

As noted in class, one way of testing whether such a "definition" is in fact well-defined is to transform it into a set of inductive rules. Let R be the set of rule instances obtained from the following rules:

$$\emptyset/(0,1),$$
$$\emptyset/(1,2),$$
$$\{(n,m),(n+2,m'')\}/(m,(1+m'')^2).$$

1. Explain why there must be a *least* binary relation $F \subseteq \mathbf{N} \times \mathbf{N}$ defined by R, and describe it explicitly.

2. Is F the graph of a function?

3. Show that F still does not satisfy the equations. (*Hint*: Since $F(F(0))$ is defined, the third equation implies that $F(2)$ is defined.)

4. Conclude that there is no partial function $f : \mathbf{N} \to \mathbf{N}$ satisfying the equations.

Problem 2. Winskel Exercise 3.13.

Problem 3. For $L \subseteq \textbf{Loc}$, the relation $=_L$ between states is defined by

$$\sigma_1 =_L \sigma_2 \text{ iff } \sigma_1(X) = \sigma_2(X) \text{ for all } X \in L.$$

The binary relation $\sim_L$ between arithmetic expressions is defined by

$$a_1 \sim_L a_2 \text{ iff } (\forall \sigma_1, \sigma_2, m \,.\, \sigma_1 =_L \sigma_2 \Rightarrow (\langle a_1, \sigma_1 \rangle \to m \text{ iff } \langle a_2, \sigma_2 \rangle \to m))$$

Prove that $a \sim_{\text{loc}(a)} a$ for all arithmetic expressions a. Use the induction hypothesis $P(a)$, where

$$P(a) \text{ iff } a \sim_{\text{loc}(a)} a$$

That is, you should prove $\forall a \in \textbf{Aexp}\,.P(a)$.

Optional: The operational behavior of **IMP** expressions and commands depends only on the locations they can read and write. We give here a precise definition of this dependence and verify it. (It may be helpful to look at Winskel Prop. 4.7.)

The locations writable by a command c are given by $\text{loc}_L(c)$, and are exactly those locations appearing on the left-hand side of an assignment command in c (see Winskel, p. 39).

The locations readable by a command c are exactly those locations appearing in a position *not* on the left-hand side of an assigment in c. Let $\text{loc}_{R'}(c)$ be the set of these locations.

For example, if

$$c_0 = \textbf{if } (X \leq Y) \textbf{ then } X := Y + 1 \textbf{ else } W := Z,$$

then $\text{loc}_L(c_0) = \{W, X\}$, and $\text{loc}_{R'}(c_0) = \{X, Y, Z\}$. Note that X is both in loc_L and in $\text{loc}_{R'}$. Also, $\text{loc}(c) = \text{loc}_L(c) \cup \text{loc}_{R'}(c)$ for any c.

The binary relation $\sim_L$ between commands is defined:

$$\begin{aligned} c_1 \sim_L c_2 \text{ iff } (\sigma_1 =_L \sigma_2 \Rightarrow \\ &(\langle c_1, \sigma_1 \rangle \to \sigma_1' \text{ for some } \sigma_1' \text{ iff } \langle c_2, \sigma_2 \rangle \to \sigma_2' \text{ for some } \sigma_2') \;\&\\ &((\langle c_1, \sigma_1 \rangle \to \sigma_1' \;\&\; \langle c_2, \sigma_2 \rangle \to \sigma_2') \Rightarrow \sigma_1' =_L \sigma_2')). \end{aligned}$$

1. Prove that $c \sim_{\text{loc}_{R'}(c)} c$ for all $c \in \textbf{Com}$. You should assume the result of the first part of the problem, and its counterpart for boolean expressions:

$$\forall a \in \textbf{Aexp}, L \supseteq \text{loc}(a).a \sim_L a$$
$$\forall b \in \textbf{Bexp}, L \supseteq \text{loc}(b).b \sim_L b$$

   You should use as induction hypothesis $P(d)$, where

$$\begin{aligned} P(d) \text{ iff } \forall c, \sigma, \sigma', \sigma_2, L.\ & \text{If } d \Vdash \langle c, \sigma \rangle \to \sigma',\ L \supseteq \text{loc}_{R'}(c), \text{ and } \sigma =_L \sigma_2, \\ & \text{then (i) } \Vdash \langle c, \sigma_2 \rangle \to \sigma_2' \text{ for some } \sigma_2'; \text{ and} \\ & \text{(ii) if } \Vdash \langle c, \sigma_2 \rangle \to \sigma_2', \text{ then } \sigma' =_L \sigma_2', \text{ for all } \sigma_2'. \end{aligned}$$

2. Conclude that $c \sim_{\mathrm{loc}(c)} c$. (This is not quite immediate. It may be helpful to appeal to Winskel Prop. 4.7.)

3. We referred to the natural semantics and the one-step semantics of **IMP** as "operational," but since **Loc** is an infinite set, and therefore so is each state, these "operational" semantics involve copying and updating infinite objects in derivations. Briefly explain how the previous result leads to a more truly "operational" version of natural and one-step semantics using finite portions of states. Then say precisely how the original versions of operational semantics can be retrieved from the versions using finite portions of states.

Problem 4. Winskel Exercise 4.3.

Problem Set 2 Solutions

This problem set was worth 34 points.

Problem 1. **[6 points]** Give a mathematically precise inductive definition of *which-var*.

Answer: First we give the domain and range of *which-var* :

$$\textit{which-var} : \langle \text{exp} \rangle \times \langle \text{seq} \rangle \rightarrow \langle \text{var} \rangle,$$

where $\langle \text{seq} \rangle$ is the set of sequences. Let s range over $\langle \text{seq} \rangle$, x range over $\langle \text{var} \rangle$, and M range over $\langle \text{exp} \rangle$.

Now we define *which-var* by the inductive rules:

$\textit{which-var}(x, \langle\,\rangle) = x$

$\textit{which-var}(\texttt{(if } M_1\ M_2\ M_3\texttt{)}, \langle i+1\rangle @ s') = \textit{which-var}(M_i, s')$ for $i \in \{1,2,3\}$

$\textit{which-var}(\texttt{(}M_1 \ldots M_n\texttt{)}, \langle i\rangle @ s') = \textit{which-var}(M_i, s')$ for $1 \leq i \leq n$

$\textit{which-var}(\texttt{(lambda (}x_1 \ldots x_n\texttt{) } M\texttt{)}, \langle 2, i\rangle) = x_i$ for $1 \leq i \leq n$

$\textit{which-var}(\texttt{(lambda (}x_1 \ldots x_n\texttt{) } M\texttt{)}, \langle 3\rangle @ s') = \textit{which-var}(M, s')$ for $n \geq 0$

$\textit{which-var}(\texttt{(letrec ((}x_1\ M_1\texttt{)} \ldots \texttt{(}x_n\ M_n\texttt{)) } M\texttt{)}, \langle 2, i, 1\rangle) = x_i$ for $1 \leq i \leq n$

$\textit{which-var}(\texttt{(letrec ((}x_1\ M_1\texttt{)} \ldots \texttt{(}x_n\ M_n\texttt{)) } M\texttt{)}, \langle 2, i, 2\rangle @ s')$
$= \textit{which-var}(M_i, s')$ for $1 \leq i \leq n$

$\textit{which-var}(\texttt{(letrec ((}x_1\ M_1\texttt{)} \ldots \texttt{(}x_n\ M_n\texttt{)) } M\texttt{)}, \langle 3\rangle @ s')$
$= \textit{which-var}(M, s')$ for $n \geq 0$

Problem 2. **[6 points]** Give a mathematically precise inductive definition of the partial function *str→tree-occur* which converts an expression and a string index into a tree-path.

Answer: In solving this problem, I started out by writing Scheme pseudo-code:

```
(define (strToTree s n start)
  if (= start n)                ; invariant: start <= n
     <>                         ; <> is the empty sequence
     (let loop ((j 1)
                (begin-mark (+1 start))
                (end-mark (next-expr s (+1 start))))
        (if (< n end-mark)
            (@ <j> (strToTree s n begin-mark))
            (loop (+1 j)
                  end-mark
                  (next-expr s end-mark)))))
```

We can compute $str\to tree\text{-}occur(s,n)$ by `(strToTree s n 0)`; it would not be hard to turn this into a real Scheme program.

It is straightforward, though somewhat inconvenient, to convert the Scheme pseudocode into mathematics:

Let **S** be the set of strings, and **N** be the set of integers. We were given the functions

$$\begin{aligned} next\text{-}expr &: \mathbf{S} \times \mathbf{N} \to \mathbf{N}, \\ @ &: \langle \text{seq} \rangle \times \langle \text{seq} \rangle \to \langle \text{seq} \rangle. \end{aligned}$$

We define $str\to tree\text{-}occur : \mathbf{S} \times \mathbf{N} \to \langle \text{seq} \rangle$ by

$$str\to tree\text{-}occur(s,n) = strToTree(s,n,0),$$

where $strToTree : \mathbf{S} \times \mathbf{N} \times \mathbf{N} \to \langle \text{seq} \rangle$ is defined by

$$strToTree(s,n,i) = \begin{cases} \langle\, \rangle & \text{if } i = n, \\ loop(s,n,1,i+1,next\text{-}expr(s,i+1)) & \text{otherwise.} \end{cases}$$

The function $loop : \mathbf{S} \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \langle \text{seq} \rangle$ is defined by

$$loop(s,n,j,b,e) = \begin{cases} \langle j \rangle @ strToTree(s,n,b) & \text{if } n < e, \\ loop(s,n,j+1,e,next\text{-}expr(s,e)) & \text{otherwise.} \end{cases}$$

Problem 3. [**4 points**] Does the equivalence $2 \approx$ (+ 1 1) hold?

Answer: No, because + is a free variable that can be rebound in an unexpected fashion. For example, if $C[\cdot]$ is the context

(letrec ((+ (lambda (a b) 0))) $[\cdot]$),

then Numval($C[2]$) = 2 but Numval(C[(+ 1 1)]) = 0.

On the other hand, $2 \approx$ (<<+>> 1 1) does hold.

Problem 4. [**4 points**] Find an M and N such that $\mathrm{FreeV}(M) \neq \mathrm{FreeV}(N)$, and $M \approx N$.

Answer: The simplest example that I can think of is

M = 3,
N = (if #t 3 foo).

Here M has no free variables, but FreeV(N) = {foo}. However, foo is never used in any execution of N; regardless of the value of foo, N will return 3.

Problem 5. [**4 points**] Give an example showing that the Context Lemma fails if we use Numval instead of NBval in the definition of $\underset{\text{appl}}{\approx}$.

Answer: There are simple contexts that can distinguish #t and #f numerically; for example, if $C[\cdot]$ is the context

(if $[\cdot]$ 6 2),

then Numval(C[#t]) = 6, but Numval(C[#f]) = 2. Thus #t $\not\approx$ #f.

However, there is no applicative context that can *numerically* distinguish #t and #f. (This is easy to see once you note that Numval(#t) and Numval(#f) are both undefined, because neither evaluates to a numeral. And therefore Numval(#t) = Numval(#f).)

Thus we would have #t $\underset{\text{appl}}{\approx}$ #f, but #t $\not\approx$ #f, contradicting the Context Lemma.

Problem 6. [**4 points**] Show that the Context Lemma fails in the language extended with the operator <<equal?>>. <<equal?>> is defined on page 11 of Handout 6; you should assume that <<equal?>> returns #f in all cases where it is currently unspecified in Handout 6.

Answer: We take as counterexample

$$M = \texttt{(lambda (x) x)},$$
$$N = \texttt{(lambda (y) y)}.$$

Clearly M and N are equal as functions. Thus $M \underset{\text{appl}}{\approx} N$.

However, by the definition of `<<equal?>>`, (`<<equal?>>` M N) evaluates to `#f`.

This lets us create a non-applicative context $C[\cdot]$ that numerically distinguishes M and N: if

$$C[\cdot] = \texttt{(if (<<equal?>> } [\cdot] \texttt{ (lambda (x) x)) 7 3)},$$

then

$$\text{Numval}(M) = 7,$$
$$\text{Numval}(N) = 3.$$

Thus $M \not\approx N$.

Problem 7. [6 points] Winskel 4.13.

Exhibit a monotonic operator that is not increasing.

Answer: Take F to be the operator mapping sets of integers to sets of integers defined as follows:

$$F(A) = \emptyset.$$

That is, F maps every set to the empty set.

Clearly F is monotonic: for any A and B, $F(A) \subseteq F(B)$ because $F(A) = F(B) = \emptyset$.

But F is not increasing, because if A is any nonempty set, $A \not\subseteq F(A) = \emptyset$. ■

Show that given any set A there is a least fixed point of $\overline{R}$ which includes A, and that this property can fail for monotonic operations.

Answer: We proceed almost exactly as in Winskel's proof of Proposition 4.12. We define a sequence of sets, $A_0, A_1, A_2, \ldots$, by repeatedly applying $\overline{R}$ to A:

$$\begin{aligned}
A_0 &= \overline{R}^0(A) = A, \\
A_1 &= \overline{R}(A_0) = \overline{R}^1(A) = \overline{R}(A), \\
A_2 &= \overline{R}(A_1) = \overline{R}^2(A) = \overline{R}(\overline{R}(A)), \\
&\vdots \\
A_{n+1} &= \overline{R}(A_n) = \overline{R}^{n+1}(A), \\
&\vdots
\end{aligned}$$

Let $A_\omega = \bigcup_{n\in\omega} A_n$. We will prove that A_ω is the least fixed point of $\overline{R}$ containing A:

Proposition:

1. $\overline{R}(A_\omega) = A_\omega$.
2. If $A \subseteq B$ and $\overline{R}(B) = B$, then $A_\omega \subseteq B$.

Proof:

1. Since $\overline{R}$ is increasing we already know that $A_\omega \subseteq \overline{R}(A_\omega)$. It remains to show $\overline{R}(A_\omega) \subseteq A_\omega$.

   Suppose $a \in \overline{R}(A_\omega)$; we wish to show $a \in A_\omega$. Now $a \in \overline{R}(A_\omega)$ iff $a \in A_\omega$ or $\exists X \subseteq A_\omega.(X/a) \in R$. So we only need show that

   $$(\exists X \subseteq A_\omega.(X/a) \in R) \Rightarrow a \in A_\omega,$$

   that is, we must show that A_ω is R-closed.

   Suppose $X \subseteq A_\omega$ and $(X/a) \in R$. Because X is finite, there is some $n \in \omega$ such that $X \subseteq A_n$. Then by the definition of $\overline{R}$, we have $a \in A_{n+1}$. Therefore $a \in A_\omega$, and $\overline{R}(A_\omega) \subseteq A_\omega$.

2. Suppose $A \subseteq B$ and $\overline{R}(B) = B$. We show by mathematical induction that for all $n \in \omega$, $A_n \subseteq B$.

   By hypothesis $A_0 = A \subseteq B$.

   Now assume $A_n \subseteq B$. Since $\overline{R}$ is monotonic, we have

   $$\overline{R}(A_n) \subseteq \overline{R}(B).$$

   But $\overline{R}(A_n) = A_{n+1}$ and $\overline{R}(B) = B$, so $A_{n+1} \subseteq B$.

   Thus we can conclude that $A_\omega \subseteq B$.

Finally, we show that this can fail for monotonic operators, by counterexample. Namely, it fails for the operator F defined above. Since F maps every set to the empty set, there can be only one fixpoint of F: the empty set. Thus if A is nonempty, there is no fixpoint of F containing A. ■

Problem Set 3 Solutions

Problem 1. Consider the following proposed definition of a possibly partial function $f : \mathbf{N} \to \mathbf{N}$:

$$f(0) = 1$$
$$f(1) = 2$$
$$f(f(n)) = (1 + f(n+2))^2$$

In this problem we examine whether this is a proper definition of a function f.

As noted in class, one way of testing whether such a "definition" is in fact well-defined is to transform it into a set of inductive rules. Let R be the set of rule instances obtained from the following rules:

$$\emptyset/(0,1),$$
$$\emptyset/(1,2),$$
$$\{(n,m),(n+2,m'')\}/(m,(1+m'')^2).$$

1. Explain why there must be a *least* binary relation $F \subseteq \mathbf{N} \times \mathbf{N}$ defined by R, and describe it explicitly.
2. Is F the graph of a function?
3. Show that F still does not satisfy the equations. (*Hint*: Since $F(F(0))$ is defined, the third equation implies that $F(2)$ is defined.)
4. Conclude that there is no partial function $f : \mathbf{N} \to \mathbf{N}$ satisfying the equations.

Answer:

1. The fact that there must be a least relation F was proved in chapter 4 of Winskel; see Proposition 4.1. The least relation F is exactly $I_R = \{ (0,1),(1,2) \}$.
2. Yes, F is the graph of a (partial) function.
3. Any f that satisfies the equations surely satisfies $f(0) = 1$, and $f(1) = 2$. And F indeed satisfies these conditions. But now consider the third equation, letting $n = 0$. The equation says

$$f(f(0)) = (1 + f(2))^2,$$

or, rewriting the left-hand side,

$$f(1) = (1 + f(2))^2.$$

This equation implies that $f(1)$ is defined if and only if $f(2)$ is defined. And as we have argued, $f(1)$ *is* defined and is equal to 2. But now consider F: it is not defined on 2. Therefore F does not satisfy the equations.

4. As we have seen, any f that satisfies the equations must satisfy

$$f(1) = (1 + f(2))^2.$$

Since $f(1) = 2$ we must have

$$f(2) = \sqrt{2} - 1.$$

But there is no natural number we can assign to $f(2)$ so that this holds ($f(2)$ would have to be a real). Thus there is no f satisfying the equations.

Problem 2. Winskel Exercise 3.13.

Answer: For an arithmetic expression a, $\mathrm{loc}(a)$ is defined by structural induction as follows:

$$\begin{array}{rclrcl}
\mathrm{loc}(n) & = & \emptyset, & \mathrm{loc}(a_0 + a_1) & = & \mathrm{loc}(a_0) \cup \mathrm{loc}(a_1), \\
\mathrm{loc}(X) & = & \{X\}, & \mathrm{loc}(a_0 - a_1) & = & \mathrm{loc}(a_0) \cup \mathrm{loc}(a_1), \\
 & & & \mathrm{loc}(a_0 \times a_1) & = & \mathrm{loc}(a_0) \cup \mathrm{loc}(a_1).
\end{array}$$

For a boolean expression b, $\mathrm{loc}(b)$ is defined by structural induction as follows:

$$\begin{array}{rclrcl}
\mathrm{loc}(\mathbf{true}) & = & \emptyset, & \mathrm{loc}(a_0 = a_1) & = & \mathrm{loc}(a_0) \cup \mathrm{loc}(a_1), \\
\mathrm{loc}(\mathbf{false}) & = & \emptyset, & \mathrm{loc}(a_0 \leq a_1) & = & \mathrm{loc}(a_0) \cup \mathrm{loc}(a_1), \\
\mathrm{loc}(\neg b') & = & \mathrm{loc}(b'), & \mathrm{loc}(b_0 \vee b_1) & = & \mathrm{loc}(b_0) \cup \mathrm{loc}(b_1), \\
 & & & \mathrm{loc}(b_0 \wedge b_1) & = & \mathrm{loc}(b_0) \cup \mathrm{loc}(b_1).
\end{array}$$

For a command c, $\mathrm{loc}_R(c)$ is defined by structural induction as follows:

$$\begin{array}{rcl}
\mathrm{loc}_R(\mathbf{skip}) & = & \emptyset, \\
\mathrm{loc}_R(X := a) & = & \mathrm{loc}(a), \\
\mathrm{loc}_R(c_0; c_1) & = & \mathrm{loc}_R(c_0) \cup \mathrm{loc}_R(c_1), \\
\mathrm{loc}_R(\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1) & = & \mathrm{loc}_R(c_0) \cup \mathrm{loc}_R(c_1), \\
\mathrm{loc}_R(\mathbf{while}\ b\ \mathbf{do}\ c') & = & \mathrm{loc}_R(c').
\end{array}$$

Problem 3. For $L \subseteq \mathbf{Loc}$, the relation $=_L$ between states is defined by

$$\sigma_1 =_L \sigma_2 \text{ iff } \sigma_1(X) = \sigma_2(X) \text{ for all } X \in L.$$

The binary relation $\sim_L$ between arithmetic expressions is defined by

$$a_1 \sim_L a_2 \text{ iff } \left(\forall \sigma_1, \sigma_2, m \,.\, \sigma_1 =_L \sigma_2 \Rightarrow (\langle a_1, \sigma_1\rangle \to m \text{ iff } \langle a_2, \sigma_2\rangle \to m)\right).$$

Prove that $a \sim_{\mathrm{loc}(a)} a$ for all arithmetic expressions a. Use the induction hypothesis $P(a)$, where

$$P(a) \text{ iff } a \sim_{\mathrm{loc}(a)} a.$$

That is, you should prove $\forall a \in \mathbf{Aexp}\,.P(a)$.

Answer: We proceed by cases on the structure of a.

$a \equiv n$: then $\forall \sigma.\langle a, \sigma\rangle \to n$, and this is the only possible derivation. So clearly $P(a)$ holds.

$a \equiv X$: then if $\sigma_1 =_{\mathrm{loc}(a)} \sigma_2$, we have $\sigma_1(X) = \sigma_2(X)$. Let $n = \sigma_1(X)$. There is exactly one rule for the evaluation of $a \equiv X$, from which we determine

$$\langle a, \sigma_1\rangle \to n,$$
$$\langle a, \sigma_2\rangle \to n,$$

and these are the only derivations starting with $\langle a, \sigma_1\rangle$ or $\langle a, \sigma_2\rangle$. Thus $P(a)$ holds.

$a \equiv a_1 + a_2$: Suppose $\langle a, \sigma_1\rangle \to m$. We will show $\langle a, \sigma_2\rangle \to m$. (The reverse implication is proved in the same way.)

Since $\langle a, \sigma_1\rangle \to m$, we must have

$$\langle a_1, \sigma_1\rangle \to m_1,$$
$$\langle a_2, \sigma_1\rangle \to m_2,$$

for some m_1, m_2 that sum to m. By induction we know $P(a_1)$ and $P(a_2)$; so

$$\langle a_1, \sigma_2\rangle \to m_1,$$
$$\langle a_2, \sigma_2\rangle \to m_2,$$

Therefore we can conclude $\langle a, \sigma_2\rangle \to m$.

The other cases ($a \equiv a_1 - a_2, a \equiv a_1 \times a_2$) follow similarly. ■

Optional: The operational behavior of **IMP** expressions and commands depends only on the locations they can read and write. We give here a precise definition of this dependence and verify it. (It may be helpful to look at Winskel Prop. 4.7.)

The locations writable by a command c are given by $\text{loc}_L(c)$, and are exactly those locations appearing on the left-hand side of an assignment command in c (see Winskel, p. 39).

The locations readable by a command c are exactly those locations appearing in a position *not* on the left-hand side of an assigment in c. Let $\text{loc}_{R'}(c)$ be the set of these locations.

For example, if

$$c_0 = \textbf{if } (X \leq Y) \textbf{ then } X := Y + 1 \textbf{ else } W := Z,$$

then $\text{loc}_L(c_0) = \{W, X\}$, and $\text{loc}_{R'}(c_0) = \{X, Y, Z\}$. Note that X is both in loc_L and in $\text{loc}_{R'}$. Also, $\text{loc}(c) = \text{loc}_L(c) \cup \text{loc}_{R'}(c)$ for any c.

The binary relation $\sim_L$ between commands is defined:

$$\begin{aligned} c_1 \sim_L c_2 \text{ iff } (\sigma_1 =_L \sigma_2 \Rightarrow \\ & ((\langle c_1, \sigma_1\rangle \rightarrow \sigma_1' \text{ for some } \sigma_1' \text{ iff } \langle c_2, \sigma_2\rangle \rightarrow \sigma_2' \text{ for some } \sigma_2') \ \& \\ & (((\langle c_1, \sigma_1\rangle \rightarrow \sigma_1' \ \& \ \langle c_2, \sigma_2\rangle \rightarrow \sigma_2') \Rightarrow \sigma_1' =_L \sigma_2')). \end{aligned}$$

1. Prove that $c \sim_{\text{loc}_{R'}(c)} c$ for all $c \in \textbf{Com}$. You should assume the result of the first part of the problem, and its counterpart for boolean expressions:

   $$\forall a \in \textbf{Aexp}\,.a \sim_{\text{loc}(a)} a$$
   $$\forall b \in \textbf{Bexp}\,.b \sim_{\text{loc}(b)} b$$

   You should use as induction hypothesis $P(d)$, where

   $$\begin{aligned} P(d) \text{ iff } \forall c, \sigma, \sigma', \sigma_2, L.\ & \text{If } d \Vdash \langle c, \sigma\rangle \rightarrow \sigma',\ L \supseteq \text{loc}_{R'}(c), \text{ and } \sigma =_L \sigma_2, \\ & \text{then (i)} \Vdash \langle c, \sigma_2\rangle \rightarrow \sigma_2' \text{ for some } \sigma_2'; \text{ and} \\ & \text{(ii) if} \Vdash \langle c, \sigma_2\rangle \rightarrow \sigma_2', \text{ then } \sigma' =_L \sigma_2', \text{ for all } \sigma_2'. \end{aligned}$$

2. Conclude that $c \sim_{\text{loc}(c)} c$. (This is not quite immediate. It may be helpful to appeal to Winskel Prop. 4.7.)

3. We referred to the natural semantics and the one-step semantics of **IMP** as "operational," but since **Loc** is an infinite set, and therefore so is each state, these "operational" semantics involve copying and updating infinite objects in derivations. Briefly explain how the previous result leads to a more truly "operational" version of natural and one-step semantics using finite portions of states. Then say precisely how the original versions of operational semantics can be retrieved from the versions using finite portions of states.

Answer: To appear.

Problem 4. Winskel Exercise 4.3.

For rule instances R, show that the set Z defined by

$$Z = \bigcap \{\ Q \mid Q \text{ is } R\text{-closed}\ \}$$

is R-closed. What is the set Z?

Answer: First, note that the set

$$W = \{\ Q \mid Q \text{ is } R\text{-closed}\ \}$$

is non-empty (it contains at least I_R). This is important, because $\bigcap \emptyset$ is not well-defined. So we have verified that $Z = \bigcap W$ is indeed a well-defined set.

Now suppose (X/y) is an instance of a rule in R, and that $X \subseteq Z$. Then by the definition of Z, $X \subseteq Q$ for all $Q \in W$. Since each $Q \in W$ is R-closed, $y \in Q$ for all $Q \in W$. Therefore $y \in \bigcap W = Z$, and Z is R-closed.

Furthermore, Z is the *smallest* R-closed set: by definition, it is contained in all R-closed sets. But it was proved in Winskel (Prop. 4.1) that I_R is the smallest R-closed set. Therefore we know $Z = I_R$. ∎

Quiz 1 and Solutions from 1991

Instructions. For your reference, there is an appendix listing the definitions of the "evaluates to" relation $\rightarrow$, and the one-step rewriting relation $\rightarrow_1$ on configurations of the language **IMP**.

For Problems 1 and 2, let w be the **IMP** command

$$\textbf{while}\,45 \leq X\,\textbf{do}\,X := X - 3;\; Y := X - 1;\; X := Y - 1$$

and let σ be a state such that $\sigma(X) = 1000$ and $\sigma(Y) = 2000$.

Problem 1 [10 points]. According to the inductive definition of evaluation, the assertion

$$\langle w, \sigma \rangle \rightarrow \sigma[40/X][41/Y]$$

has a unique derivation. How many instances of the sequencing rule scheme (seq $\rightarrow$) given below appear in this derivation? *384*

$$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'', \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma'}{\langle (c_0; c_1), \sigma \rangle \rightarrow \sigma'} \qquad (\text{seq} \rightarrow)$$

Note: The quiz did not ask for any explanation. One will be asked for on problem set 4.

Problem 2 [15 points]. By definition, $\langle w, \sigma \rangle \rightarrow_1^* \sigma[40/X][41/Y]$ because there is a (unique) sequence of the form:

$$\langle w, \sigma \rangle \rightarrow_1 \langle c_1, \sigma_1 \rangle \rightarrow_1 \langle c_2, \sigma_2 \rangle \rightarrow_1 \cdots \rightarrow_1 \langle c_n, \sigma_n \rangle \rightarrow_1 \sigma[40/X][41/Y]$$

where n happens to be 2500.

Notice that σ_1 must equal σ, and c_1 must be

$$\textbf{if}\,45 \leq X\,\textbf{then}\,(X := X - 3;\; Y := X - 1;\; X := Y - 1;\; w)\,\textbf{else}\,\textbf{skip}.$$

2(a) [6 points]. What are

c_2? **if** $45 \leq 1000$ **then** $(X := X - 3;\ Y := X - 1;\ X := Y - 1;\ w$ **else skip**

σ_2? σ

c_3? **if true then** $(X := X - 3;\ Y := X - 1;\ X := Y - 1;\ w)$ **else skip**

σ_3? σ

c_n? **skip**

σ_n? $\sigma[40/X][41/Y]$

The answers for c_3 and σ_3 were graded relative to the answers for c_2 and σ_2.

2(b) [4 points]. How many c_i's are of the form **while** b **do** c? *193* *Actually the correct answer is really 192. We forgot that the first configuration in the chain ($\langle w, \sigma \rangle$) was not explicitly described to be c_0. Thus the first* **while** *in the chain does not really contribute to the count. If we had let $\langle c_0, \sigma_0 \rangle = \langle w, \sigma \rangle$ then there would not have been a problem. Full credit was given for either answer, unless it was clear that 192 was arrived at via a mistake (and thus two wrongs making a right).*

2(c) [5 points]. There are k times as many c_i's which are of the form

if b' **then** c **else** c'

than are of the form

while b'' **do** c''.

What is k? *3* *Due to the slight miscounting in the preceding answer the correct answer is really* $(193 * 3)/192 \approx 3.0156$. *Credit was given for either answer.*

Problem 3 [20 points]. It was noted in class that every **Aexp** configuration evaluates to a number. Likewise, one can prove by *structural induction on* **Bexp** that every **Bexp** configuration evaluates to a truth value, namely,

$$\text{for all } \langle b, \sigma\rangle, \text{ there is a } t \in \{\mathbf{true}, \mathbf{false}\} \text{ such that } \langle b, \sigma\rangle \to t.$$

3(a) [10 points]. List the cases of the structural induction and indicate what must be shown for each case.

The base cases are:

[$b \equiv t \in \{\mathbf{true}, \mathbf{false}\}$] *We must show that, under no additional assumptions, there exists a* $t' \in \{\mathbf{true}, \mathbf{false}\}$ *such that* $\langle t, \sigma\rangle \to t'$.

[$b \equiv a_0 = a_1$] *We must show that there exists a* $t \in \{\mathbf{true}, \mathbf{false}\}$ *such that* $\langle a_0 = a_1, \sigma\rangle \to t$. *To do so, we may use the analogous property for* **Aexp**, viz. *to assume that there exist* n_0 *and* n_1 *such that* $\langle a_0, \sigma\rangle \to n_0$ *and* $\langle a_1, \sigma\rangle \to n_1$.

[$b \equiv a_0 \leq a_1$] *Similar to the preceding case.*

The non-base cases are:

[$b \equiv \neg b'$] *Under the assumption that there exists a* $t \in \{\mathbf{true}, \mathbf{false}\}$ *such that* $\langle b', \sigma\rangle \to t$, *we must show that there exists a* $t' \in \{\mathbf{true}, \mathbf{false}\}$ *such that* $\langle \neg b', \sigma\rangle \to t'$.

[$b \equiv b_0 \wedge b_1$] *Under the assumption that there exist* $t_0, t_1 \in \{\mathbf{true}, \mathbf{false}\}$ *such that* $\langle b_0, \sigma\rangle \to t_0$ *and* $\langle b_1, \sigma\rangle \to t_1$, *we must show that there exists a* $t \in \{\mathbf{true}, \mathbf{false}\}$ *such that* $\langle b_0 \wedge b_1, \sigma\rangle \to t$.

[$b \equiv b_0 \vee b_1$] *Similar to the preceding case.*

3(b) [10 points]. Pick a non-base case and prove it!

We pick the non base-case [$b \equiv b_0 \wedge b_1$].

Under the inductive assumption that there exist $t_0, t_1 \in \{\mathbf{true}, \mathbf{false}\}$ *such that* $\langle b_0, \sigma\rangle \to t_0$ *and* $\langle b_1, \sigma\rangle \to t_1$, *we must show that there exists a* $t \in \{\mathbf{true}, \mathbf{false}\}$ *such that* $\langle b_0 \wedge b_1, \sigma\rangle \to t$. *But by rule (and* $\to$*), there is such a* t*, namely the conjunction of* t_0 *and* t_1.

Problem 4 [25 points]. We define a "parallel evals to" relation, $\hookrightarrow$, which is a variation of the "evals to" relation, $\rightarrow$. The rules to define $\hookrightarrow$ are obtained from the rules defining $\rightarrow$ by replacing all occurrences of "$\rightarrow$" by "$\hookrightarrow$". In addition, there is one further "parallel if" rule:

$$\frac{\langle c_0, \sigma\rangle \hookrightarrow \sigma', \quad \langle c_1, \sigma\rangle \hookrightarrow \sigma'}{\langle \textbf{if}\, b\, \textbf{then}\, c_0\, \textbf{else}\, c_1, \sigma\rangle \hookrightarrow \sigma'} \qquad (\text{par-if} \hookrightarrow)$$

4(a) [5 points]. Give a simple example of a command, c, such that $\langle c, \sigma\rangle \hookrightarrow \sigma$ has more than one derivation for any state σ.

$\textbf{if}\, b\, \textbf{then}\, c'\, \textbf{else}\, c'$, *for any* c'

Although the *definition* of $\hookrightarrow$ differs from that of $\rightarrow$, it turns out to specify the *same relation* on configurations as $\rightarrow$. The nontrivial direction of this remark is the implication

$$\langle c, \sigma\rangle \hookrightarrow \sigma' \;\Rightarrow\; \langle c, \sigma\rangle \rightarrow \sigma'.$$

This implication can be proved by induction on the definition of $\hookrightarrow$ (that is by rule induction on the rules for $\hookrightarrow$).

4(b) [10 points]. Briefly explain what the cases of the induction are, and why there is only one non-trivial case.

There is one case for each of the inference rules of $\hookrightarrow$ on **Com**-*configurations (or on* **Aexp**, **Bexp**, *and* **Com** *configurations. This was slightly ambiguous but unimportant, since either reading gave the same definition of $\hookrightarrow$).*

So there are the base cases for the **Com**-*configuration rules: (skip $\hookrightarrow$), (assign $\hookrightarrow$).*

The inductive cases are for the rules: (seq $\hookrightarrow$), (if-true $\hookrightarrow$), (if-false $\hookrightarrow$) (while-false $\hookrightarrow$), (while-false $\hookrightarrow$), and finally a case for the new rule (par-if $\hookrightarrow$).

The only non-trivial case is for the new rule (par-if $\hookrightarrow$), because the other rules for $\hookrightarrow$ are the same as the corresponding rules for $\rightarrow$. In particular, if $\langle c, \sigma\rangle \hookrightarrow \sigma'$ follows from some ($\hookrightarrow$)-rule, R, other than (par-if $\rightarrow$), then the antecedents if any, of R which involve $\hookrightarrow$, each implies by induction, the corresponding antecedent with "$\hookrightarrow$" replaced by "$\rightarrow$", so $\langle c, \sigma\rangle \rightarrow \sigma'$ follows trivially by the $\rightarrow$-version of R.

4(c) [10 points]. Prove the non-trivial case. (You may assume the results mentioned in Problem 3.)

So, we suppose that $\langle c, \sigma\rangle \hookrightarrow \sigma'$ because of the rule (par-if $\hookrightarrow$).

Then c must be of the form **if** b **then** c_0 **else** c_1*, where* $\langle c_0, \sigma \rangle \hookrightarrow \sigma'$ *and* $\langle c_1, \sigma \rangle \hookrightarrow \sigma'$ *(so by induction, we may assume that* $\langle c_0, \sigma \rangle \rightarrow \sigma'$ *and* $\langle c_1, \sigma \rangle \rightarrow \sigma'$*).*

By Problem 3, we know that there exists a $t \in \{\mathbf{true}, \mathbf{false}\}$ *such that* $\langle b, \sigma \rangle \rightarrow t$*. This gives us two cases based on* t*.*

Suppose $t \equiv \mathbf{true}$*. Since* $\langle b, \sigma \rangle \rightarrow \mathbf{true}$*, rule (if-true* $\rightarrow$*) applies, and so then* $\langle c, \sigma \rangle \rightarrow \sigma'$*.*

The case of $t \equiv \mathbf{false}$ *works similarly. Since* $\langle b, \sigma \rangle \rightarrow \mathbf{false}$*, rule (if-false* $\rightarrow$*) applies, and so then* $\langle c, \sigma \rangle \rightarrow \sigma'$*.*

A Appendix

We use n, sometimes with subscripts as in n_0, n_1, to denote arbitrary elements of **Num**. Similarly, we assume $X, Y \in$ **Loc**; $a \in$ **Aexp**, $t \in \{\mathbf{true}, \mathbf{false}\}$; $b \in$ **Bexp**; $c \in$ **Com**; and $\sigma \in$ the set of states.

A.1 "Evals to" Rules for IMP

Notice that we give a name for each rule in parentheses to its right.

A.1.1 Aexp Rules

$$\langle n, \sigma\rangle \rightarrow n \qquad (\text{num} \rightarrow)$$

$$\langle X, \sigma\rangle \rightarrow \sigma(n) \qquad (\text{loc} \rightarrow)$$

$$\frac{\langle a_0, \sigma\rangle \rightarrow n_0, \quad \langle a_1, \sigma\rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma\rangle \rightarrow n} \qquad (\text{plus} \rightarrow)$$

where n is the sum of n_0 and n_1.

Similarly, there are rules (times $\rightarrow$) and (minus $\rightarrow$).

A.1.2 Bexp Rules

$$\langle t, \sigma\rangle \rightarrow t \qquad (\text{bool} \rightarrow)$$

$$\frac{\langle a_0, \sigma\rangle \rightarrow n_0, \quad \langle a_1, \sigma\rangle \rightarrow n_1}{\langle a_0 = a_1, \sigma\rangle \rightarrow t} \qquad (\text{equal} \rightarrow)$$

where $t \equiv$ **true** if n_0 and n_1 are equal, otherwise $t \equiv$ **false**.

Similarly, there is a rule ($\leq \rightarrow$).

$$\frac{\langle b, \sigma\rangle \rightarrow t}{\langle \neg b, \sigma\rangle \rightarrow t'} \qquad (\text{not} \rightarrow)$$

where t' is the negation of t.

$$\frac{\langle b_0, \sigma\rangle \rightarrow t_0, \quad \langle b_1, \sigma\rangle \rightarrow t_1}{\langle b_0 \wedge b_1, \sigma\rangle \rightarrow t} \qquad (\text{and} \rightarrow)$$

where t is **true** if $t_0 \equiv$ **true** and $t_1 \equiv$ **true**, and is **false** otherwise.

Similarly, there is a rule (or $\rightarrow$).

A.1.3 Com Rules

$$\langle \mathbf{skip}, \sigma\rangle \to \sigma \qquad (\text{skip} \to)$$

$$\frac{\langle a, \sigma\rangle \to n}{\langle X := a, \sigma\rangle \to \sigma[n/X]} \qquad (\text{assign} \to)$$

$$\frac{\langle c_0, \sigma\rangle \to \sigma'', \quad \langle c_1, \sigma''\rangle \to \sigma'}{\langle (c_0; c_1), \sigma\rangle \to \sigma'} \qquad (\text{seq} \to)$$

$$\frac{\langle b, \sigma\rangle \to \mathbf{true}, \quad \langle c_0, \sigma\rangle \to \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \to \sigma'} \qquad (\text{if-true} \to)$$

$$\frac{\langle b, \sigma\rangle \to \mathbf{false}, \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \to \sigma'} \qquad (\text{if-false} \to)$$

$$\frac{\langle b, \sigma\rangle \to \mathbf{false}}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma\rangle \to \sigma} \qquad (\text{while-false} \to)$$

$$\frac{\langle b, \sigma\rangle \to \mathbf{true}, \quad \langle c, \sigma\rangle \to \sigma'', \quad \langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma''\rangle \to \sigma'}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma\rangle \to \sigma'} \qquad (\text{while-true} \to)$$

A.2 Rewriting rules for IMP

A.2.1 Aexp Rules

$$\langle X, \sigma\rangle \to_1 \langle \sigma(X), \sigma\rangle \qquad (\text{loc} \to_1)$$

$$\frac{\langle a_0, \sigma\rangle \to_1 \langle a_0', \sigma\rangle}{\langle a_0 + a_1, \sigma\rangle \to_1 \langle a_0' + a_1, \sigma\rangle} \qquad (\text{plus-left} \to_1)$$

$$\frac{\langle a, \sigma\rangle \to_1 \langle a', \sigma\rangle}{\langle n + a, \sigma\rangle \to_1 \langle n + a', \sigma\rangle} \qquad (\text{plus-right} \to_1)$$

$$\langle n_0 + n_1, \sigma\rangle \to_1 \langle n, \sigma\rangle \qquad (\text{plus-num} \to_1)$$

where n is the sum of n_0 and n_1.

Similarly, there are rules (times-left $\to_1$), (times-right $\to_1$), (times-num $\to_1$), (minus-left $\to_1$), (minus-right $\to_1$), and (minus-num $\to_1$).

A.2.2 Bexp Rules

$$\frac{\langle a_0,\sigma\rangle \rightarrow_1 \langle a_0',\sigma\rangle}{\langle a_0 = a_1,\sigma\rangle \rightarrow_1 \langle a_0' = a_1,\sigma\rangle} \qquad \text{(equal-left} \rightarrow_1\text{)}$$

$$\frac{\langle a,\sigma\rangle \rightarrow_1 \langle a',\sigma\rangle}{\langle n = a,\sigma\rangle \rightarrow_1 \langle n = a',\sigma\rangle} \qquad \text{(equal-right} \rightarrow_1\text{)}$$

$$\langle n_0 = n_1,\sigma\rangle \rightarrow_1 \langle t,\sigma\rangle \qquad \text{(equal-num} \rightarrow_1\text{)}$$

where $t \equiv$ **true** if n_0 and n_1 are equal, otherwise $t \equiv$ **false**.

Similarly, there are rules ($\leq$-left $\rightarrow_1$), ($\leq$-right $\rightarrow_1$), and ($\leq$-num $\rightarrow_1$).

$$\frac{\langle b,\sigma\rangle \rightarrow_1 \langle b',\sigma\rangle}{\langle \neg b,\sigma\rangle \rightarrow_1 \langle \neg b',\sigma\rangle} \qquad \text{(not-eval-arg} \rightarrow_1\text{)}$$

$$\langle \neg t,\sigma\rangle \rightarrow_1 \langle t',\sigma\rangle \qquad \text{(not-bool} \rightarrow_1\text{)}$$

where t' is the negation of t.

$$\frac{\langle b_0,\sigma\rangle \rightarrow_1 \langle b_0',\sigma\rangle}{\langle b_0 \wedge b_1,\sigma\rangle \rightarrow_1 \langle b_0' \wedge b_1,\sigma\rangle} \qquad \text{(and-left} \rightarrow_1\text{)}$$

$$\frac{\langle b,\sigma\rangle \rightarrow_1 \langle b',\sigma\rangle}{\langle t \wedge b,\sigma\rangle \rightarrow_1 \langle t \wedge b',\sigma\rangle} \qquad \text{(and-right} \rightarrow_1\text{)}$$

$$\langle t_0 \wedge t_1,\sigma\rangle \rightarrow_1 \langle t,\sigma\rangle \qquad \text{(and-bool} \rightarrow_1\text{)}$$

where $t \equiv$ **true** if $t_0 \equiv$ **true** and $t_1 \equiv$ **true**, otherwise $t \equiv$ **false**.

Similarly there are rules (or-left $\rightarrow_1$), (or-right, $\rightarrow_1$) and (or-bool $\rightarrow_1$).

A.2.3 Com Rules

$$\langle \mathbf{skip},\sigma\rangle \rightarrow_1 \sigma \qquad \text{(skip} \rightarrow_1\text{)}$$

$$\frac{\langle a,\sigma\rangle \rightarrow_1 \langle a',\sigma\rangle}{\langle X := a,\sigma\rangle \rightarrow_1 \langle X := a',\sigma\rangle} \qquad \text{(assign-eval-arg} \rightarrow_1\text{)}$$

$$\langle X := n,\sigma\rangle \rightarrow_1 \sigma[n/X] \qquad \text{(assign-num} \rightarrow_1\text{)}$$

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \langle c_0', \sigma'\rangle}{\langle (c_0; c_1), \sigma\rangle \rightarrow_1 \langle (c_0'; c_1), \sigma'\rangle} \qquad \text{(seq-start } \rightarrow_1)$$

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \sigma'}{\langle (c_0; c_1), \sigma\rangle \rightarrow_1 \langle c_1, \sigma'\rangle} \qquad \text{(seq-finish } \rightarrow_1)$$

$$\frac{\langle b, \sigma\rangle \rightarrow_1 \langle b', \sigma\rangle}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \rightarrow_1 \langle \mathbf{if}\, b'\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle} \qquad \text{(if-eval-guard } \rightarrow_1)$$

$$\langle \mathbf{if\ true\ then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \rightarrow_1 \langle c_0, \sigma\rangle \qquad \text{(if-true } \rightarrow_1)$$

$$\langle \mathbf{if\ false\ then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \rightarrow_1 \langle c_1, \sigma\rangle \qquad \text{(if-false } \rightarrow_1)$$

$$\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma\rangle \rightarrow_1 \langle \mathbf{if}\, b\, \mathbf{then}(c; \mathbf{while}\, b\, \mathbf{do}\, c)\, \mathbf{else\ skip}, \sigma\rangle \qquad \text{(while } \rightarrow_1)$$

Quiz 1 and Solutions from 1992

(This was a **closed book, closed notes** exam. There were four (4) problems, worth 25 points each.)

Recall from Problem Set 2 that $\mathbf{IMP}_r$ is the extension of **IMP** by a further expression construct, $c\,\mathbf{resultis}\,a$, where c is a command and a is an arithmetic expression. On this quiz, arithmetic expressions, $a, a_0, \ldots$, Boolean expressions, $b, b_0, \ldots$, and commands, $c, c_0, \ldots$, are understood to be those of $\mathbf{IMP}_r$. The natural evaluation rules ($\rightarrow_r$) and one-step ($\rightarrow_{r,1}$) rules of $\mathbf{IMP}_r$ were attached in an appendix to this quiz. To avoid clutter, the subscript "r" on the arrows will henceforth be dropped.

In $\mathbf{IMP}_r$, every command is equivalent to an assignment statement, namely

$$c \quad \sim \quad X := c\,\mathbf{resultis}\,X$$

(Recall that $c_1 \sim c_2$ means that for all σ, σ',

$$\langle c_1, \sigma\rangle \rightarrow \sigma' \text{ iff } \langle c_2, \sigma\rangle \rightarrow \sigma'.)$$

Problem 1 [25 points]. One way to prove this equivalence would be by appeal to the one-step semantics. Thus, if

$$\langle c, \sigma\rangle \rightarrow_1 \langle c_1, \sigma_1\rangle \rightarrow_1 \cdots \rightarrow_1 \langle c_n, \sigma_n\rangle \rightarrow_1 \sigma'$$

for some sequence of configurations $\langle c_i, \sigma_i\rangle$ and $n \geq 1$, then

$$\langle X := c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow_1 \langle c'_1, \sigma'_1\rangle \rightarrow_1 \cdots \rightarrow_1 \langle c'_n, \sigma'_n\rangle \rightarrow_1 \cdots \rightarrow_1 \langle c'_{n+k}, \sigma'_{n+k}\rangle \rightarrow_1 \sigma'$$

for some sequence of configurations $\langle c'_i, \sigma'_i\rangle$ and $k \geq 1$.

Note that c'_{n+k} is an assignment of the form $X := m$.

1(a) [18 points]. Which of the following correctly describes m? (**circle** *all* those which are correct):

1. n
2. $n+1$
3. $n+k$
4. $n+k+1$
5. $\sigma(X)$
6. (circled) $\sigma'(X)$
7. $\sigma(X)+k$
8. $\sigma'(X)+k$
9. $\sigma'_n(X)$
10. (circled) $\sigma'_{n+k}(X)$

1(b) [7 points]. What is k? $k =$ 2

From the evaluation rules for $\rightarrow_1$, *we know that every* $\langle c'_i, \sigma'_i \rangle$ *must be exactly* $\langle X := c_i\ \mathbf{resultis}\ X, \sigma_i \rangle$, *for* $i \leq n$. *Further, since* $\langle c_n, \sigma_n \rangle \rightarrow_1 \sigma'$, *we know that* c_n *is one of* **skip**, $X := m$, *or* $Y := m'$ *(for some numeral* m'*), where* Y *is a different location from* X. *This gives us three possible classes of evaluations, starting with* $\langle c'_n, \sigma'_n \rangle$:

$$\begin{aligned}\langle X := (\mathbf{skip}\ \mathbf{resultis}\ X), \sigma_n \rangle &\rightarrow_1 \langle X := X, \sigma_n \rangle \\ &\rightarrow_1 \langle X := m, \sigma_n \rangle \\ &\rightarrow_1 \sigma_n\end{aligned}$$

or

$$\begin{aligned}\langle X := (X := m\ \mathbf{resultis}\ X), \sigma_n \rangle &\rightarrow_1 \langle X := X, \sigma_n[m/X] \rangle \\ &\rightarrow_1 \langle X := m, \sigma_n[m/X] \rangle \\ &\rightarrow_1 \sigma_n[m/X]\end{aligned}$$

or

$$\begin{aligned}\langle X := (Y := m'\ \mathbf{resultis}\ X), \sigma_n \rangle &\rightarrow_1 \langle X := X, \sigma_n[m'/Y] \rangle \\ &\rightarrow_1 \langle X := m, \sigma_n[m'/Y] \rangle \\ &\rightarrow_1 \sigma_n[m'/Y]\end{aligned}$$

Note, first, that all of these are of the form

$$\langle c'_n, \sigma'_n \rangle \rightarrow_1 \langle c'_{n+1}, \sigma'_{n+1} \rangle \rightarrow_1 \langle c'_{n+2}, \sigma'_{n+2} \rangle \rightarrow_1 \sigma',$$

so $k = 2$. *Next, each* c'_{n+2} *is of the form* $X := \sigma'_{n+2}(X)$, *so answer* 10 *applies. Finally, note that in each case* $\sigma' = \sigma'_{n+2}$, *so answer* 6 *applies as well. Answer* 9 *is true in some cases (like the first and third), but not true in general, as the second case shows.*

Problem 2 [25 points]. There is another, direct way to prove this equivalence: describe how to find a derivation of

2(a) [12 points].

$$\langle X := c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow \sigma'$$

from a derivation of

$$\langle c, \sigma\rangle \rightarrow \sigma',$$

Given a derivation $d \Vdash \langle c, \sigma\rangle \rightarrow \sigma'$, *we have the following derivation:*

$$\cfrac{\cfrac{d \qquad \langle X, \sigma'\rangle \rightarrow \langle \sigma'(X), \sigma'\rangle}{\langle c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow \langle \sigma'(X), \sigma'\rangle}}{\langle X := c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow \sigma'[\sigma'(X)/X]}$$

Since $\sigma'[\sigma'(X)/X] = \sigma'$, *we have a derivation of*

$$\langle X := c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow \sigma'.$$

2(b) [13 points]. and vice-versa.

Given a derivation $d \Vdash \langle X := c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow \sigma'$, *we know that* d *must end with the rule*

$$\frac{\langle c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow \langle m, \sigma''\rangle}{\langle X := c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow \sigma'}$$

where $\sigma' = \sigma''[m/X]$.

Then, the derivation of $\langle c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow \langle m, \sigma''\rangle$ *must end with the rule*

$$\frac{\langle c, \sigma\rangle \rightarrow \sigma''' \quad \langle X, \sigma'''\rangle \rightarrow \langle m, \sigma''\rangle}{\langle c\,\mathbf{resultis}\,X, \sigma\rangle \rightarrow \langle m, \sigma''\rangle}$$

Now,

$$\langle X, \sigma'''\rangle \rightarrow \langle m, \sigma''\rangle$$

is an axiom; but for this axiom to hold, it must be the case that $\sigma'''(X) = m$ *and* $\sigma'' = \sigma'''$. *But this means that*

$$\sigma''' = \sigma'''[m/X] = \sigma''[m/X] = \sigma'.$$

Thus, we know that d is of the form:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \langle c,\sigma\rangle \to \sigma'\end{array} \qquad \langle X,\sigma'\rangle \to \langle \sigma'(X),\sigma'\rangle}{\langle c\,\mathbf{resultis}\,X,\sigma\rangle \to \langle \sigma'(X),\sigma'\rangle}}{\langle X := c\,\mathbf{resultis}\,X,\sigma\rangle \to \sigma'}$$

which contains a derivation for $\langle c,\sigma\rangle \to \sigma'$ *as a subderivation.*

Problem 3 [25 points]. We define four (4) sets of rule instances over the natural numbers $\omega = \{ 0, 1, 2, \dots \}$:

$$R_1 ::= \quad \frac{}{7} \quad \frac{n}{n+2} \quad \frac{n, n+1}{n+2} \qquad \text{for } n \in \omega$$

$$R_2 ::= \quad \frac{}{7} \quad \frac{}{8} \quad \frac{n}{n+2} \quad \frac{n, n+1}{n+2} \qquad \text{for } n \in \omega$$

$$R_3 ::= \quad \frac{}{1} \quad \frac{n}{n+2} \quad \frac{1, 3, 5, \dots, 2n+1}{2n} \qquad \text{for } n \in \omega,\ n \geq 1$$

$$R_4 ::= \quad \frac{}{4} \quad \frac{}{6} \quad \frac{n, m}{n+m} \qquad \text{for } m, n \in \omega$$

3(a) [16 points]. For $i = 1, 2, 3, 4$ give a simple description of I_{R_i}, the set of numbers derivable from R_i.

$I_{R_1} =$ *The odd integers* ≥ 7

$I_{R_2} =$ *The integers* ≥ 7

$I_{R_3} =$ *The integers* ≥ 1 (or ≥ 0)

$I_{R_4} =$ *The even integers* ≥ 4

3(b) [9 points]. Which i satisfy the property that every number in I_{R_i} has a unique R_i-derivation? For the others, what is the smallest integer with two derivations?

	unique derivation (yes/no)	If "no," smallest integer
R_1	*Yes*	
R_2	*No*	9
R_3	*No*	4
R_4	*No*	12

Problem 4 [25 points]. Say that $b \in \mathbf{Bexp}_r$ is *side-effect free* iff, for all states σ_1, σ_2,

$$\langle b, \sigma_1 \rangle \to \langle t, \sigma_2 \rangle \quad \Rightarrow \quad \sigma_1 = \sigma_2.$$

4(a) [20 points]. Suppose b is side-effect free. Carefully prove that

$$\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma \rangle \to \sigma' \quad \Rightarrow \quad \langle b, \sigma' \rangle \to \langle \mathbf{false}, \sigma' \rangle.$$

This is a very simple induction on derivations.

Suppose $d \Vdash \langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma \rangle \to \sigma'$. *Then:*

Base case: *If the last rule of* d *is*

$$\frac{\langle b, \sigma \rangle \to \langle \mathbf{false}, \sigma' \rangle}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma \rangle \to \sigma'}$$

then, since b *is side-effect free,* $\sigma = \sigma'$, *so we have* $\langle b, \sigma' \rangle \to \langle \mathbf{false}, \sigma' \rangle$.

Induction: *If the last rule of* d *is not the while-false rule, then it must be while-true:*

$$\frac{\langle b, \sigma \rangle \to \langle \mathbf{true}, \sigma'' \rangle \quad \langle c, \sigma'' \rangle \to \sigma''' \quad \langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma''' \rangle \to \sigma'}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma \rangle \to \sigma'}$$

But, since the derivation of $\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma''' \rangle \to \sigma'$ *is a subderivation of* d, *by induction we have*

$$\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma''' \rangle \to \sigma' \quad \Rightarrow \quad \langle b, \sigma' \rangle \to \langle \mathbf{false}, \sigma' \rangle.$$

So $\langle b, \sigma' \rangle \to \langle \mathbf{false}, \sigma' \rangle$.

4(b) [5 points]. Give a simple b (with side-effects) which is a counterexample to the implication of part 4(a). $\boxed{(\textbf{if}\, X = 0\, \textbf{then}\, X := 1\, \textbf{else}\, X := 0\, \textbf{resultis}\, X) = 0}$

It was required to find a **Bexp**, *b, such that*

$$\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma \rangle \to \sigma' \qquad \textit{and} \qquad \langle b, \sigma' \rangle \not\to \langle \textbf{false}, \sigma' \rangle$$

for some c, σ and σ′. The easiest way is simply to ensure that b always changes the state; thus, if b is $(X := X + 1\, \textbf{resultis}\, 0) = 1$*, then, if* $\sigma(X) = 0$ *we have*

$$\langle \textbf{while}\, b\, \textbf{do}\, \textbf{skip}, \sigma \rangle \to \sigma[1/X] \qquad \textit{and} \qquad \langle b, \sigma[1/X] \rangle \to \langle \textbf{false}, \sigma[2/X] \rangle$$

The expression we have given in the answer box is slightly more complex, and actually yields **true** *as well as changing the state. With b as in the answer box and* $\sigma(X) = 0$*:*

$$\langle \textbf{while}\, b\, \textbf{do}\, \textbf{skip}, \sigma \rangle \to \sigma[1/X] \qquad \textit{and} \qquad \langle b, \sigma[1/X] \rangle \to \langle \textbf{true}, \sigma[0/X] \rangle$$

Yet another possibility would be for $\langle b, \sigma' \rangle$ *to fail to evaluate to anything in state* σ'*, as with* $((\textbf{while}\, X = 1\, \textbf{do}\, \textbf{skip}); X := X + 1\, \textbf{resultis}\, 0) = 1$.

Finally, there is the possibility of a **Bexp**, *b, such that*

$$\langle b, \sigma' \rangle \to \langle \textbf{true}, \sigma' \rangle.$$

We leave this as an extra-credit exercise.

Appendix A: Natural Evaluation Rules for $\mathbf{IMP}_r$

As mentioned earlier in the quiz, we'll be omitting the r subscripts to reduce clutter. Otherwise, these definitions are the same as those given in Problem Set 2.

Rules for Aexp

$$\langle n, \sigma \rangle \rightarrow \langle n, \sigma \rangle$$

$$\langle X, \sigma \rangle \rightarrow \langle \sigma(X), \sigma \rangle$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow \langle n_0, \sigma'' \rangle, \quad \langle a_1, \sigma'' \rangle \rightarrow \langle n_1, \sigma' \rangle}{\langle a_0 + a_1, \sigma \rangle \rightarrow \langle n, \sigma' \rangle}$$

where n is the sum of n_0 and n_1.

There are similar rules for $\times$ and $-$.

$$\frac{\langle c, \sigma \rangle \rightarrow \sigma'', \quad \langle a, \sigma'' \rangle \rightarrow \langle n, \sigma' \rangle}{\langle c\, \mathbf{resultis}\, a, \sigma \rangle \rightarrow \langle n, \sigma' \rangle}$$

Rules for Bexp

$$\langle t, \sigma \rangle \rightarrow \langle t, \sigma \rangle$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow \langle n_0, \sigma'' \rangle, \quad \langle a_1, \sigma'' \rangle \rightarrow \langle n_1, \sigma' \rangle}{\langle a_0 = a_1, \sigma \rangle \rightarrow \langle t, \sigma' \rangle}$$

where $t = \mathbf{true}$ if n_0 and n_1 are equal, otherwise $t = \mathbf{false}$.

There is a similar rule for $\leq$.

$$\frac{\langle b, \sigma \rangle \rightarrow \langle t, \sigma' \rangle}{\langle \neg b, \sigma \rangle \rightarrow \langle t', \sigma' \rangle}$$

where t' is the negation of t.

$$\frac{\langle b_0, \sigma \rangle \rightarrow \langle t_0, \sigma'' \rangle, \quad \langle b_1, \sigma'' \rangle \rightarrow \langle t_1, \sigma' \rangle}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow \langle t, \sigma' \rangle}$$

where t is $\mathbf{true}$ if $t_0 = \mathbf{true}$ and $t_1 = \mathbf{true}$, and is $\mathbf{false}$ otherwise.

There is a similar rule for $\vee$.

Rules for Com

$$\langle \mathbf{skip}, \sigma \rangle \rightarrow \sigma$$

$$\frac{\langle a, \sigma \rangle \rightarrow \langle n, \sigma' \rangle}{\langle X := a, \sigma \rangle \rightarrow \sigma'[n/X]}$$

$$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'', \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma'}{\langle (c_0; c_1), \sigma \rangle \rightarrow \sigma'}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \langle \mathbf{true}, \sigma'' \rangle, \quad \langle c_0, \sigma'' \rangle \rightarrow \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma \rangle \rightarrow \sigma'}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \langle \mathbf{false}, \sigma'' \rangle, \quad \langle c_1, \sigma'' \rangle \rightarrow \sigma'}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma \rangle \rightarrow \sigma'}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \langle \mathbf{false}, \sigma' \rangle}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma \rangle \rightarrow \sigma'}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \langle \mathbf{true}, \sigma'' \rangle, \quad \langle c, \sigma'' \rangle \rightarrow \sigma''', \quad \langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma''' \rangle \rightarrow \sigma'}{\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma \rangle \rightarrow \sigma'}$$

Appendix B: One-Step Evaluation Rules for IMP$_r$

As in previous handouts, we will use **op** to range over syntactic operator symbols, and *op* to range over corresponding arithmetic or Boolean operations. Also, as elsewhere in the quiz, we will be dropping the r subscripts to avoid clutter. The language involved is still **IMP**$_r$, however.

Rules for Arithmetic Expressions, Aexp

$$\langle X, \sigma\rangle \rightarrow_1 \langle \sigma(X), \sigma\rangle$$

$$\frac{\langle c, \sigma\rangle \rightarrow_1 \langle c', \sigma'\rangle}{\langle c\,\mathbf{resultis}\,a, \sigma\rangle \rightarrow_1 \langle c'\,\mathbf{resultis}\,a, \sigma'\rangle}$$

$$\frac{\langle c, \sigma\rangle \rightarrow_1 \sigma'}{\langle c\,\mathbf{resultis}\,a, \sigma\rangle \rightarrow_1 \langle a, \sigma'\rangle}$$

$$\frac{\langle a_0, \sigma\rangle \rightarrow_1 \langle a_0', \sigma'\rangle}{\langle a_0\ \mathbf{op}\ a_1, \sigma\rangle \rightarrow_1 \langle a_0'\ \mathbf{op}\ a_1, \sigma'\rangle}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow_1 \langle a_1', \sigma'\rangle}{\langle n\ \mathbf{op}\ a_1, \sigma\rangle \rightarrow_1 \langle n\ \mathbf{op}\ a_1', \sigma'\rangle}$$

$$\langle n\ \mathbf{op}\ m, \sigma\rangle \rightarrow_1 \langle n\ op\ m, \sigma\rangle$$

op	*op*
+	the sum function
−	the subtraction function
×	the multiplication function

Rules for Boolean Expressions, Bexp

$$\frac{\langle a_0, \sigma\rangle \rightarrow_1 \langle a_0', \sigma'\rangle}{\langle a_0\ \mathbf{op}\ a_1, \sigma\rangle \rightarrow_1 \langle a_0'\ \mathbf{op}\ a_1, \sigma'\rangle}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow_1 \langle a_1', \sigma'\rangle}{\langle n\ \mathbf{op}\ a_1, \sigma\rangle \rightarrow_1 \langle n\ \mathbf{op}\ a_1', \sigma'\rangle}$$

$$\langle n\ \mathbf{op}\ m, \sigma\rangle \rightarrow_1 \langle n\ op\ m, \sigma\rangle$$

op	*op*
$=$	the equality predicate
$\leq$	the less than or equal to predicate

We next have the rules for Boolean negation:

$$\frac{\langle b, \sigma\rangle \rightarrow_1 \langle b', \sigma'\rangle}{\langle \neg b, \sigma\rangle \rightarrow_1 \langle \neg b', \sigma'\rangle}$$

$$\langle \neg\, \mathbf{true}, \sigma\rangle \rightarrow_1 \langle \mathbf{false}, \sigma\rangle$$

$$\langle \neg\, \mathbf{false}, \sigma\rangle \rightarrow_1 \langle \mathbf{true}, \sigma\rangle$$

Finally we have the rules for binary Boolean operators. We let $t, t_0, t_1, \ldots$ range over the set $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$.

$$\frac{\langle b_0, \sigma\rangle \rightarrow_1 \langle b_0', \sigma'\rangle}{\langle b_0 \mathbf{\ op\ } b_1, \sigma\rangle \rightarrow_1 \langle b_0' \mathbf{\ op\ } b_1, \sigma'\rangle}$$

$$\frac{\langle b_1, \sigma\rangle \rightarrow_1 \langle b_1', \sigma'\rangle}{\langle t_0 \mathbf{\ op\ } b_1, \sigma\rangle \rightarrow_1 \langle t_0 \mathbf{\ op\ } b_1', \sigma'\rangle}$$

$$\langle t_0 \mathbf{\ op\ } t_1, \sigma\rangle \rightarrow_1 \langle t_0 \ op\ t_1, \sigma\rangle$$

op	*op*
$\wedge$	the conjunction operation (Boolean AND)
$\vee$	the disjunction operation (Boolean OR)

Rules for Commands, Com

Atomic Commands:

$$\langle \mathbf{skip}, \sigma\rangle \rightarrow_1 \sigma$$

$$\frac{\langle a, \sigma\rangle \rightarrow_1 \langle a', \sigma'\rangle}{\langle X := a, \sigma\rangle \rightarrow_1 \langle X := a', \sigma'\rangle}$$

$$\langle X := n, \sigma\rangle \rightarrow_1 \sigma[n/X]$$

Sequencing:

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \langle c_0', \sigma'\rangle}{\langle (c_0; c_1), \sigma\rangle \rightarrow_1 \langle (c_0'; c_1), \sigma'\rangle}$$

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \sigma'}{\langle (c_0; c_1), \sigma\rangle \rightarrow_1 \langle c_1, \sigma'\rangle}$$

Conditionals:

$$\frac{\langle b, \sigma\rangle \rightarrow_1 \langle b', \sigma'\rangle}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \rightarrow_1 \langle \textbf{if } b' \textbf{ then } c_0 \textbf{ else } c_1, \sigma'\rangle}$$

$$\langle \textbf{if true then } c_0 \textbf{ else } c_1, \sigma\rangle \rightarrow_1 \langle c_0, \sigma\rangle$$

$$\langle \textbf{if false then } c_0 \textbf{ else } c_1, \sigma\rangle \rightarrow_1 \langle c_1, \sigma\rangle$$

While-loops:

$$\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \rightarrow_1 \langle \textbf{if } b \textbf{ then}(c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}, \sigma\rangle$$

Quiz 1

Instructions. This is a **closed book, closed note** exam. There are three (3) problems, on pages 2–7 of this booklet. Write your solutions for all problems on this exam sheet in the spaces provided, including your *name on each sheet.* Don't accidentally skip a page. Ask for further blank sheets if you need them.

There are a number of appendices, containing information on **IMP** and the Substitution Model of Scheme. You have seen all of this material before; it is included for reference only.

GOOD LUCK!

NAME: ____________________

Problem	Points	Score
1	30	
2	30	
3	40	
Total	100	

Problem 1 [30 points]. For each of the following expressions of the functional kernel of Scheme, write down a Substitution Model normal form for the expression, or write that the expression has no normal form. (The Substitution Model is summarized in Appendix D, and normal forms are defined on p. 16.)

1.
```
(letrec ((x 2) (y 3))
  (<<+>> x y))
```

2.
```
(letrec ((x 2)
         (y ((lambda (x) (<<+>> 1 x)) 4)))
  (<<+>> x y))
```

3.
```
(letrec ((f (lambda (z) (<<+>> x z))))
  (lambda (x) (f x)))
```

4.
```
(letrec ((x (lambda () x))) x)
```

5.
```
(letrec ((x (lambda () x))) 1)
```

6.
```
(letrec ((y x) (x y))
  (<<+>> 1 x))
```

Problem 2 [30 points]. In this problem we ask you to add an **until** operator to the language **IMP**. (**IMP** is summarized in Appendices A and B.)

First we extend the syntax of commands as follows:

$$c ::= \ldots \mid \textbf{do } c \textbf{ until } b.$$

Informally, **do** c **until** b executes c repeatedly until b evaluates to **true**. We expect that

$$\langle \textbf{do } c \textbf{ until } b, \sigma \rangle \to \sigma' \quad \text{iff} \quad \langle (c; \textbf{if } b \textbf{ then skip else } (\textbf{do } c \textbf{ until } b)), \sigma \rangle \to \sigma'.$$

Note that c is always executed at least once.

2(a) [10 points]. Complete the definition of the **until** operator by filling in the boxes in the following rules:

$$\frac{\langle c, \boxed{} \rangle \to \boxed{} \quad \langle b, \boxed{} \rangle \to \boxed{}}{\langle \textbf{do } c \textbf{ until } b, \sigma \rangle \to \sigma'} \quad \text{[until-1]}$$

$$\frac{\langle c, \boxed{} \rangle \to \boxed{} \quad \langle b, \boxed{} \rangle \to \boxed{} \quad \langle \textbf{do } c \textbf{ until } b, \boxed{} \rangle \to \boxed{}}{\langle \textbf{do } c \textbf{ until } b, \sigma \rangle \to \sigma'} \quad \text{[until-2]}$$

2(b) [10 points]. Suppose

$$c_{\mathrm{do}} \equiv \mathbf{do}\ X := X - 1\ \mathbf{until}\ X \leq 0,$$

and suppose σ_{a} is a state such that $\sigma_{\mathrm{a}}(X) = 2$. Then $\Vdash \langle c_{\mathrm{do}}, \sigma_{\mathrm{a}} \rangle \rightarrow \sigma_{\mathrm{a}}[0/X]$.

Draw the derivation tree for $\Vdash \langle c_{\mathrm{do}}, \sigma_{\mathrm{a}} \rangle \rightarrow \sigma_{\mathrm{a}}[0/X]$.

2(c) [10 points]. Now suppose σ_{b} is a state such that $\sigma_{\mathrm{b}}(X) = n$, where $n \geq 1$. Then $\Vdash \langle c_{\mathrm{do}}, \sigma_{\mathrm{b}} \rangle \rightarrow \sigma_{\mathrm{b}}[0/X]$.

How many times is the rule [until-1] used in the derivation of $\Vdash \langle c_{\mathrm{do}}, \sigma_{\mathrm{b}} \rangle \rightarrow \sigma_{\mathrm{b}}[0/X]$?

How many times is the rule [until-2] used in the derivation of $\Vdash \langle c_{\mathrm{do}}, \sigma_{\mathrm{b}} \rangle \rightarrow \sigma_{\mathrm{b}}[0/X]$?

Problem 3 [40 points]. Recall the following sets, which were defined in Winskel's book (the exact definitions can be found in Appendix C):

- $\text{loc}(a)$ is the set of locations appearing in an arithmetic expression a;
- $\text{loc}(b)$ is the set of locations appearing in a boolean expression b;
- $\text{loc}(c)$ is the set of locations appearing in a command c; and
- $\text{loc}_L(c)$ is the set of locations appearing on the left-hand side of assignment statements in command c.

Note that if $X \notin \text{loc}_L(c)$, then the value of X cannot be changed by the execution of c. Intuitively, then, $\text{loc}_L(c)$ is the set of locations which might be *written* during the evaluation of c.

We want you to define $\text{loc}_{\text{read}}(c)$, the set of locations in command c which might be *read* during the evaluation of c. For example, if

$$c_{\text{a}} \equiv \textbf{if } W \leq 0 \textbf{ then } X := Y + 1 \textbf{ else } Z := X + Y,$$

then we should have $\text{loc}_{\text{read}}(c_{\text{a}}) = \{W, X, Y\}$.

3(a) [16 points]. Give a definition of $\text{loc}_{\text{read}}(c)$ by induction on the structure of commands. We have filled in the case $c \equiv c_0; c_1$ for you; you must fill in the other cases. You may use $\text{loc}_L(c)$, $\text{loc}(a)$, $\text{loc}(b)$, or $\text{loc}(c)$ in your definition if you so desire.

- $\text{loc}_{\text{read}}(c_0; c_1) \;=\; \text{loc}_{\text{read}}(c_0) \cup \text{loc}_{\text{read}}(c_1)$.
- $\text{loc}_{\text{read}}(\textbf{skip}) \;=$
- $\text{loc}_{\text{read}}(X := a) \;=$
- $\text{loc}_{\text{read}}(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) \;=$
- $\text{loc}_{\text{read}}(\textbf{while } b \textbf{ do } c') \;=$

3(b) [4 points]. Give a simple command c such that

$$\text{loc}(c) = \text{loc}_L(c) = \text{loc}_{\textbf{read}}(c) \neq \emptyset.$$

3(c) [20 points]. It can be proved, by structural induction on commands c, that

$$\text{loc}(c) = \text{loc}_{\textbf{read}}(c) \cup \text{loc}_L(c).$$

We have filled in the case $c \equiv c_0; c_1$ of the proof for you; you must fill in the proof for the cases $c \equiv X := a$ and $c \equiv \textbf{while } b \textbf{ do } c'$. (You need not prove the remaining cases, $c \equiv \textbf{skip}$ and $c \equiv \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1$.)

- $c \equiv c_0; c_1$: By definition,

$$\text{loc}(c) = \text{loc}(c_0) \cup \text{loc}(c_1).$$

And by induction,

$$\text{loc}(c_0) = \text{loc}_{\textbf{read}}(c_0) \cup \text{loc}_L(c_0),$$
$$\text{loc}(c_1) = \text{loc}_{\textbf{read}}(c_1) \cup \text{loc}_L(c_1).$$

Therefore

$$\text{loc}(c) = \text{loc}_{\textbf{read}}(c_0) \cup \text{loc}_{\textbf{read}}(c_1) \cup \text{loc}_L(c_0) \cup \text{loc}_L(c_1),$$

and by the definition of $\text{loc}_{\textbf{read}}$ and loc_L,

$$\text{loc}(c) = \text{loc}_{\textbf{read}}(c) \cup \text{loc}_L(c),$$

as desired.

- $c \equiv X := a$:

- $c \equiv$ **while** b **do** c':

Appendix A: Syntax of IMP

For **Aexp**:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

For **Bexp**:

$$b ::= \textbf{true} \mid \textbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

For **Com**:

$$c ::= \textbf{skip} \mid X := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c$$

Appendix B: Semantics of IMP

Evaluation of **Aexp**'s:

$$\langle n, \sigma \rangle \to n$$

$$\langle X, \sigma \rangle \to \sigma(X)$$

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n}$$ where n is the sum of n_0 and n_1.

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 - a_1, \sigma \rangle \to n}$$ where n is the result of subtracting n_1 from n_0.

$$\frac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \times a_1, \sigma \rangle \to n}$$ where n is the product of n_0 and n_1.

Evaluation of **Bexp**'s:

$$\langle \mathbf{true}, \sigma \rangle \to \mathbf{true}$$

$$\langle \mathbf{false}, \sigma \rangle \to \mathbf{false}$$

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to \mathbf{true}}$$ if n and m are equal.

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to \mathbf{false}}$$ if n and m are unequal.

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 \le a_1, \sigma \rangle \to \mathbf{true}}$$ if n is less than or equal to m.

$$\frac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 \le a_1, \sigma \rangle \to \mathbf{false}}$$ if n is not less than or equal to m.

$$\frac{\langle b, \sigma \rangle \to \mathbf{true}}{\langle \neg b, \sigma \rangle \to \mathbf{false}} \qquad \frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \neg b, \sigma \rangle \to \mathbf{true}}$$

$$\frac{\langle b_0, \sigma \rangle \to t_0 \quad \langle b_1, \sigma \rangle \to t_1}{\langle b_0 \wedge b_1, \sigma \rangle \to t}$$
where t is **true** if $t_0 \equiv \mathbf{true}$ and $t_1 \equiv \mathbf{true}$, and is **false** otherwise.

$$\frac{\langle b_0, \sigma \rangle \to t_0 \quad \langle b_1, \sigma \rangle \to t_1}{\langle b_0 \vee b_1, \sigma \rangle \to t}$$
where t is **true** if $t_0 \equiv \mathbf{true}$ or $t_1 \equiv \mathbf{true}$, and is **false** otherwise.

Execution of **Com**'s:

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma$$

$$\frac{\langle a, \sigma \rangle \to m}{\langle X := a, \sigma \rangle \to \sigma[m/X]}$$

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma'' \rangle \to \sigma'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma'}$$

Appendix C: Location functions

For an arithmetic expression a, $\text{loc}(a)$ is defined by structural induction as follows:

$$\begin{array}{rclrcl} \text{loc}(n) &=& \emptyset, & \text{loc}(a_0 + a_1) &=& \text{loc}(a_0) \cup \text{loc}(a_1), \\ \text{loc}(X) &=& \{X\}, & \text{loc}(a_0 - a_1) &=& \text{loc}(a_0) \cup \text{loc}(a_1), \\ & & & \text{loc}(a_0 \times a_1) &=& \text{loc}(a_0) \cup \text{loc}(a_1). \end{array}$$

For a boolean expression b, $\text{loc}(b)$ is defined by structural induction as follows:

$$\begin{array}{rclrcl} \text{loc}(\textbf{true}) &=& \emptyset, & \text{loc}(a_0 = a_1) &=& \text{loc}(a_0) \cup \text{loc}(a_1), \\ \text{loc}(\textbf{false}) &=& \emptyset, & \text{loc}(a_0 \le a_1) &=& \text{loc}(a_0) \cup \text{loc}(a_1), \\ \text{loc}(\neg b') &=& \text{loc}(b'), & \text{loc}(b_0 \vee b_1) &=& \text{loc}(b_0) \cup \text{loc}(b_1), \\ & & & \text{loc}(b_0 \wedge b_1) &=& \text{loc}(b_0) \cup \text{loc}(b_1). \end{array}$$

For a command c, $\text{loc}(c)$ is defined by structural induction as follows:

$$\begin{aligned} \text{loc}(\textbf{skip}) &= \emptyset, \\ \text{loc}(X := a) &= \{X\} \cup \text{loc}(a), \\ \text{loc}(c_0; c_1) &= \text{loc}(c_0) \cup \text{loc}(c_1), \\ \text{loc}(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) &= \text{loc}(b) \cup \text{loc}(c_0) \cup \text{loc}(c_1), \\ \text{loc}(\textbf{while } b \textbf{ do } c') &= \text{loc}(b) \cup \text{loc}(c'). \end{aligned}$$

For a command c, $\text{loc}_L(c)$ is defined by structural induction as follows:

$$\begin{aligned} \text{loc}_L(\textbf{skip}) &= \emptyset, \\ \text{loc}_L(X := a) &= \{X\}, \\ \text{loc}_L(c_0; c_1) &= \text{loc}_L(c_0) \cup \text{loc}_L(c_1), \\ \text{loc}_L(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) &= \text{loc}_L(c_0) \cup \text{loc}_L(c_1), \\ \text{loc}_L(\textbf{while } b \textbf{ do } c') &= \text{loc}_L(c'). \end{aligned}$$

Appendix D: The Substitution Model of the Functional Kernel of Scheme

1 Syntax

We follow the grammatical conventions of the Revised[4] Scheme Report, using x, y, z to denote variables, and M, N, B to denote expressions. We use * to indicate zero or more occurrences of a phrase type and $^+$ for one or more occurrences.

```
           ⟨keyword⟩ ::= ⟨binding-keyword⟩ | ⟨nonbinding-keyword⟩
⟨nonbinding-keyword⟩ ::= if
   ⟨binding-keyword⟩ ::= lambda | letrec
               ⟨exp⟩ ::= (if ⟨exp⟩ ⟨exp⟩ ⟨exp⟩)
                       | (⟨exp⟩+)
                       | (lambda (⟨formals⟩) ⟨exp⟩)
                       | (letrec (⟨bindings⟩) ⟨exp⟩)
                       | ⟨constant⟩
                       | ⟨var⟩
          ⟨bindings⟩ ::= (⟨var⟩ ⟨exp⟩)*          (all ⟨var⟩'s must be distinct)
           ⟨formals⟩ ::= ⟨var⟩*                  (all ⟨var⟩'s must be distinct)
          ⟨constant⟩ ::= ⟨numeral⟩ | ⟨boolean⟩ | ⟨scheme-constant⟩ | ⟨system-constant⟩
           ⟨numeral⟩ ::= 0 | -1 | 3.14159 | ...
           ⟨boolean⟩ ::= #t | #f
   ⟨scheme-constant⟩ ::= ⟨built-in⟩ | ⟨rule-defined⟩
   ⟨system-constant⟩ ::= identifiers of the form <<···>> that are not ⟨scheme-constant⟩'s
               ⟨var⟩ ::= identifiers other than ⟨constant⟩'s or ⟨keyword⟩'s
          ⟨built-in⟩ ::= <<+>> | <<->> | <<*>> | <</>> | << = >> | << <= >>
                       | << < >> | <<gcd>> | <<sin>> | <<quotient>>
                       | <<zero?>> | <<abs>> | <<1+>> | <<sqrt>> | <<not>> | ...
      ⟨rule-defined⟩ ::= <<boolean?>> | <<number?>> | <<procedure?>>
```

The ⟨scheme-constant⟩'s correspond to familiar Scheme procedures. The ⟨built-in⟩'s correspond to basic operations on numerals as specified in the Revised[4] Scheme Report. The ⟨system-constant⟩'s correspond to other procedures which may be added to the system by importing external code.

2 Functional Values

The purpose of evaluating an expression is to obtain its "value."

```
     ⟨lambda-val⟩ ::= (lambda (⟨formals⟩) ⟨exp⟩)
⟨letrec-free-val⟩ ::= ⟨lambda-val⟩ | ⟨constant⟩
          ⟨value⟩ ::= ⟨letrec-free-val⟩ | (letrec ((⟨var⟩ ⟨letrec-free-val⟩)*) ⟨value⟩)
```

The ⟨letrec-free-val⟩'s will play a particularly important role in specifying Scheme's evaluation rules. We let $V, V_1, \ldots$ denote ⟨letrec-free-val⟩'s.

3 Free and Bound Variable Occurrences

We define the *free* and *bound occurrences* of variables in an expression M.

(1) M is a ⟨constant⟩.

$$\text{FreeO}(M) = \emptyset$$
$$\text{BoundO}(M) = \emptyset$$

(2) M is x.

$$\text{FreeO}(M) = \{\text{the occurrence of } x \text{ in } M\}$$
$$\text{BoundO}(M) = \emptyset$$

(3) M is `(`$key\ N_1 \cdots$`)` or `(`$N_1 \cdots$`)`.

$$\text{FreeO}(M) = \text{FreeO}(N_1, \cdots)$$
$$\text{BoundO}(M) = \text{BoundO}(N_1, \cdots)$$

where *key* is a ⟨nonbinding-keyword⟩.

(4) M is `(lambda (`$x_1 \ldots$`)` N`)`.

$$\text{FreeO}(M) = \{\, o \in \text{FreeO}(N) \mid o \text{ is } \textit{not} \text{ an occurrence of one of } x_1, \ldots\}$$
$$\text{BoundO}(M) = \{\, o \in \text{FreeO}(N) \mid o \text{ } \textit{is} \text{ an occurrence of one of } x_1, \ldots\} \cup \text{BoundO}(N)$$

(5) M is `(letrec ((`$x_1\ N_1$`)`$\cdots$`)`B`)`.

$$\text{FreeO}(M) = \{\, o \in \text{FreeO}(B) \cup \text{FreeO}(N_1) \cup \cdots \mid o \text{ is } \textit{not} \text{ an occurrence of one of } x_1, \ldots\}$$
$$\text{BoundO}(M) = \{\, o \in \text{FreeO}(B) \cup \text{FreeO}(N_1) \cup \cdots \mid o \text{ } \textit{is} \text{ an occurrence of one of } x_1, \ldots\} \cup \text{BoundO}(B) \cup \text{BoundO}(N_1) \cup \cdots$$

A variable is *free* in an expression if it has a free occurrence in the expression. It is free in a set of expressions if it is free in any of them. We write FreeV(M) for the set of variables free in M; similarly for FreeV($\{M_1, \ldots\}$).

4 Substitutions

A *substitution*, σ, is formally defined to be a total function whose domain is a finite set of ⟨var⟩'s and whose range is a set of ⟨exp⟩'s.

Any substitution σ determines an inductively defined function from ⟨exp⟩'s to ⟨exp⟩'s. We write $M\sigma$ for the result of applying this function to M. More precisely, $M\sigma$ is defined by induction simultaneously on the structure of M and the size of domain(σ). The base Case (1) of the definition ensures that the function on expressions determined by σ acts the same on variables in domain(σ) as σ itself.

(1)

$$x\sigma = \begin{cases} \sigma(x) & \text{if } x \in \text{domain}(\sigma), \\ x & \text{otherwise.} \end{cases}$$

(2) If domain(σ) $= \emptyset$, then

$$M\sigma = M$$

For any set, $\mathcal{F}$, of ⟨var⟩'s, we write $\sigma \restriction \mathcal{F}$ for the restriction of σ to $\mathcal{F} \cap \text{domain}(\sigma)$.

(3)

$$M\sigma = M\,(\sigma \restriction \text{FreeV}(M))$$

Thus, if $\text{FreeV}(M) = \emptyset$, then $M\sigma = M$.

(4)

$$\begin{aligned} (M_1\ \cdots)\sigma &= (M_1\sigma\ \cdots) \\ (key\ M_1\ \cdots)\sigma &= (key\ M_1\sigma \cdots) \end{aligned}$$

where *key* is a ⟨nonbinding-keyword⟩.

Scheme is specified to obey the "static" scoping conventions of familiar mathematical notation. This means that when M has binding constructs, free variables in expressions being substituted into M should not get bound by binding constructs in M. Such unintended binding is prevented by selectively renaming bound variables in M to be "fresh" variables. (Substituting without renaming would model the "dynamic" scoping of pre-Scheme Lisp dialects, leading to notorious "false capture" or "funarg" problems.)

A *renaming* is defined to be a substitution which maps the variables in its domain to *distinct* ⟨var⟩'s. A renaming, τ, is *fresh* with respect to a set of expressions and substitutions if none of the variables in its range have free or binding occurrences in any of the expressions or domains and ranges of the substitutions in the set. When the relevant set is clear, we simply say that τ is a *fresh renaming.*

For substitutions σ, τ, the *extension* of τ by σ, written $\tau + \sigma$, is the substitution combining σ and τ, with τ given priority where the domains overlap. That is,

$$(\tau + \sigma)\,(y) = \begin{cases} \tau(y) & \text{if } y \in \text{domain}(\tau), \\ \sigma(y) & \text{if } y \in \text{domain}(\sigma) - \text{domain}(\tau). \end{cases}$$

Note that in general $\tau + \sigma \neq \sigma + \tau$.

The next cases in the definition of $M\sigma$ are simplified by assuming that $\text{domain}(\sigma) \subseteq \text{FreeV}(M)$. By Case (3), there is no loss of generality in this assumption.

(5)

$$(\texttt{lambda}\ (x_1\ x_2 \cdots)\ M)\sigma = (\texttt{lambda}\ (x_1\tau\ x_2\tau \cdots)\ M(\tau + \sigma))$$

where τ is a fresh renaming whose domain is $\{x_1, \ldots\} \cap \text{FreeV}(\text{range}(\sigma))$.

(6)

$$(\texttt{letrec}\ ((x_1\ M_1) \cdots)\ B)\sigma = (\texttt{letrec}\ ((x_1\tau\ M_1(\tau + \sigma))\ \cdots)\ B(\tau + \sigma))$$

where τ is a fresh renaming whose domain is $\{x_1, \ldots\} \cap \text{FreeV}(\text{range}(\sigma))$.

5 Rewrite Rules

In Revised[4] Scheme, only **#f** counts as false in conditional expressions. Other values such as numerals and procedures count as true.

$$\langle\text{false-value}\rangle ::= \texttt{\#f}$$

- IF

$$\texttt{(if } V\ M_1\ M_2\texttt{)} \longrightarrow \begin{cases} M_2 & \text{if } V \text{ is a } \langle\text{false-value}\rangle, \\ M_1 & \text{otherwise.} \end{cases}$$

To capture the Revised[4] Scheme behavior of ⟨built-in⟩'s, we introduce a partial function, System-Eval : ⟨exp⟩ ⇀ ⟨exp⟩.

- CNST: *built-in and system constants*

$$\texttt{(}cnst\ V_1 \cdots\texttt{)} \longrightarrow \text{System-Eval}\,\texttt{((}cnst\ V_1 \cdots\texttt{))}$$

  where *cnst* is a ⟨built-in⟩ or ⟨system-constant⟩, and System-Eval ((*cnst* $V_1 \cdots$)) is defined.

For example,

$$\text{System-Eval}\,\texttt{((<<+>> 3 -7.1 9))} = \texttt{4.9}$$

Errors are somewhat messy to incorporate into the Substitution Model, so we take System-Eval to be undefined in cases where Revised[4] Scheme specifies errors. For example, an expression such as `(<<+>> #t)` which generates an immediate Revised[4] Scheme error will not match the lefthand side of any rewrite rule.

System-Eval also handles ⟨system-constant⟩'s generated by calls to external code which return procedures. This allows the Substitution Model implementation to interface with real Scheme or with compiled code.

- LAM: *lambda-application*

$$\texttt{((lambda (}x_1 \ldots\texttt{) } B\texttt{) } M_1 \ldots\texttt{)} \longrightarrow \texttt{(letrec ((}x_1\tau\ M_1\texttt{)...) } B\tau\texttt{)}$$

  where τ is a fresh renaming whose domain is $\{x_1, \ldots\} \cap \text{FreeV}(\{M_1, \ldots\})$.

We write $\{x \leftarrow M\}$ for the substitution whose domain is $\{x\}$ and which maps x to M.

- letrec

  INST: *letrec-instantiation*

$$\texttt{(letrec (}\langle\text{bindings}\rangle\ \texttt{(}x\ V\texttt{)}\cdots\texttt{) } B\{z \leftarrow x\}\texttt{)} \longrightarrow$$
$$\texttt{(letrec (}\langle\text{bindings}\rangle\ \texttt{(}x\ V\texttt{)}\cdots\texttt{) } B\{z \leftarrow V\}\texttt{)}$$

  where B has exactly one free occurrence of z.

  OUT: *letrec-out*:

$$\texttt{(}M_1 \ldots M_k\ \texttt{(letrec (}\langle\text{bindings}\rangle\texttt{) } B\texttt{) } M_{k+1} \ldots M_n\texttt{)} \longrightarrow$$
$$\texttt{(letrec (}\langle\text{bindings}\rangle\texttt{) (}z_1 \ldots z_k\ B\ z_{k+1} \ldots z_n\texttt{))}\sigma$$

$$\texttt{(}key\ M_1 \ldots M_k\ \texttt{(letrec (}\langle\text{bindings}\rangle\texttt{) } B\texttt{) } M_{k+1} \ldots M_n\texttt{)} \longrightarrow$$
$$\texttt{(letrec (}\langle\text{bindings}\rangle\texttt{) (}key\ z_1 \ldots z_k\ B\ z_{k+1} \ldots z_n\texttt{))}\sigma$$

  where σ is the substitution whose domain is a set $\{z_1, \ldots, z_n\}$ of fresh variables, $\sigma(z_i) = M_i$, and *key* is a ⟨nonbinding-keyword⟩.

  FLAT: *letrec-flatten*:

$$\texttt{(letrec (}\langle\text{bindings}\rangle_1\ \texttt{(}x\ \texttt{(letrec (}\langle\text{bindings}\rangle_2\texttt{) } B_0\texttt{))}\ \langle\text{bindings}\rangle_3\texttt{) } B\texttt{)} \longrightarrow$$
$$\texttt{(letrec (}\langle\text{bindings}\rangle_1\ \texttt{(}\langle\text{bindings}\rangle_2\tau\texttt{) (}x\ B_0\tau\texttt{)}\ \langle\text{bindings}\rangle_3\texttt{) } B\texttt{)}$$

  where τ is a fresh renaming whose domain is $\mathcal{F}_2 \cap (\mathcal{F}_1 \cup \{x\} \cup \mathcal{F}_3 \cup \text{FreeV}(B))$ and $\mathcal{F}_i$ is the set of binding variables in $\langle\text{binding}\rangle_i$.

- BOOL?

$$(\texttt{<<boolean?>>}\ V) \longrightarrow \begin{cases} \texttt{\#t} & \text{if } V \text{ is } \texttt{\#t} \text{ or } \texttt{\#f}, \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

- NUM?

$$(\texttt{<<number?>>}\ V) \longrightarrow \begin{cases} \texttt{\#t} & \text{if } V \text{ is a } \langle\text{numeral}\rangle, \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

- PROC?

$$(\texttt{<<procedure?>>}\ V) \longrightarrow \begin{cases} \texttt{\#t} & \text{if } V \text{ is a } \langle\text{lambda-val}\rangle, \langle\text{scheme-constant}\rangle, \\ & \quad\text{or } \langle\text{system-constant}\rangle, \\ \texttt{\#f} & \text{otherwise.} \end{cases}$$

6 Garbage Collection Rules

$$(\texttt{letrec ()}\ B) \longrightarrow B$$
$$(\texttt{letrec ((}x_1\ M\texttt{)}\cdots\texttt{)}\ B\texttt{)} \longrightarrow (\texttt{letrec ((}x_{i_1}\ M_{i_1}\texttt{)}\cdots\texttt{)}\ B\texttt{)}$$

where $x_{i_1}, x_{i_2}, \ldots$ are the live variables in the *letrec* expression. These variables are defined inductively by the conditions:

(1) all variables in FreeV(B) are live, and

(2) x_i is live if M_i is *not* a value, and

(3) if x_i is live, then all variables in FreeV(M_i) are live.

7 Contexts and Rewriting

A *context* is an ⟨exp⟩ with exactly one free occurrence of a designated variable called *the hole.* The hole is written as [ ]. If C is a context, we write $C[M]$ for the expression that results from replacing the hole in C by M *without any renaming.*

If $M_1 = C[N_1]$ and $M_2 = C[N_2]$ for some context C, and "$N_1 \longrightarrow N_2$" matches any rewrite rule or garbage collection rule above, then we say M_1 *rewrites in one step to* M_2, written

$$M_1 \underset{\text{rew}}{\longrightarrow} M_2\,.$$

In particular, when "$N_1 \longrightarrow N_2$" matches one of the garbage collecting rules, we write

$$M_1 \underset{\text{grbg}}{\longrightarrow} M_2\,.$$

We say M *rewrites to* N, written $M \underset{\text{rew}}{\longrightarrow}^* N$ if either $M = N$, or else

$$M \underset{\text{rew}}{\longrightarrow} M_1 \underset{\text{rew}}{\longrightarrow} \cdots \underset{\text{rew}}{\longrightarrow} N$$

for some expressions $M_1, \ldots$; similarly for $\underset{\text{grbg}}{\longrightarrow}^*$. An expression, M, is a *normal form* when there is no N such that $M \underset{\text{rew}}{\longrightarrow} N$. For example, according to the above rules of our Substitution Model, all ⟨constant⟩'s and ⟨var⟩'s are normal forms.

In general, an expression may have many distinct normal forms reflecting the different ways rewrite rules might be applied to it. Despite this, *numerical* normal forms are unique:

Theorem (Determinacy). If an expression M has a normal form which is a numeral, then that numeral is the *only* normal form of M.

Determinacy implies that in calculating a *numerical* value of an expression, there is no need to consider where and which rewrite rules to apply. This is a fundamental property distinguishing the functional kernel of Scheme from the fuller language with side-effects.

The Determinacy Theorem is far from obvious, and considerable ingenuity is needed to prove it.

8 Correctness

The *initial global context*, C_{init}, is defined to be the context in which variables are bound to all the corresponding constants, viz.,

C_{init} ::= `(letrec ((+ <<+>>)... (sqrt <<sqrt>>) ... (apply <<apply>>) ...) [ ])`.

Let Numval(M) denote the unique numeral, if any, which is the normal form of $C_{\text{init}}[M]$. Numval(M) is undefined if M has a normal form which is not a numeral, or if M has no normal form.

Theorem (*Numerical Correctness*). Let M be an $\langle\text{exp}\rangle$ without scheme- or system-constants whose value in the initial environment is specified in Revised[4] Scheme. If the specified value is a numeral, then Numval(M) is defined and equals the numeral. Conversely, if Numval(M) is defined, then it equals the value specified in Revised[4] Scheme when M is evaluated in Scheme's initial environment.

Note that the Theorem leaves open the possibility that Numval(M) may be defined in cases when M does not have a specified value in Revised[4] Scheme.

Proving a theorem like this which relates the (denotational semantics) specification in the Revised[4] Scheme Report with the rewriting rules of our Substitution Model requires sophisticated proof methods from programming language theory.

6.044J/18.423J: Computability, Programming, and Logic — Handout 19
Massachusetts Institute of Technology — 18 October 1993

Quiz 1 Solution

(This was a closed book, closed note exam. There were three (3) problems, worth 100 points total).

Problem 1 [30 points]. For each of the following expressions of the functional kernel of Scheme, write down a Substitution Model normal form for the expression, or write that the expression has no normal form. (The Substitution Model is summarized in Appendix D, and normal forms are defined on p. 16.)

1.
```
(letrec ((x 2) (y 3))
  (<<+>> x y))
```

   Answer: 5.

2.
```
(letrec ((x 2)
         (y ((lambda (x) (<<+>> 1 x)) 4)))
  (<<+>> x y))
```

   Answer: 7

3.
```
(letrec ((f (lambda (z) (<<+>> x z))))
  (lambda (x) (f x)))
```

   Answer: `(lambda (y) (<<+>> x y))`.

   Because the definition of **f** contains a free variable **x**, the inner declaration of **x** must be renamed to avoid capturing the free **x**. (We have used **y** here but any variable other than **x** is acceptable.)

4.
```
(letrec ((x (lambda () x))) x)
```

   Answer: The expression has no normal form.

5.
```
(letrec ((x (lambda () x))) 1)
```

   Answer: 1.

   Because **x** is bound to a value, and it is free in the body of the `letrec`, it can be garbage collected.

6.
```
(letrec ((y x) (x y))
  (<<+>> 1 x))
```

   Answer:

```
(letrec ((y x) (x y))
  (<<+>> 1 x))
```

The expression IS a normal form. Neither **x** nor **y** is bound to a value (variables are not values), thus INST cannot be applied. (Tricky; no one got this one right.)

Problem 2 [30 points]. In this problem we ask you to add an **until** operator to the language **IMP**. (**IMP** is summarized in Appendices A and B.)

First we extend the syntax of commands as follows:

$$c ::= \ldots \mid \textbf{do } c \textbf{ until } b.$$

Informally, **do** c **until** b executes c repeatedly until b evaluates to **true**. We expect that

$$\langle \textbf{do } c \textbf{ until } b, \sigma\rangle \to \sigma' \quad \text{iff} \quad \langle (c; \textbf{if } b \textbf{ then skip else } (\textbf{do } c \textbf{ until } b)), \sigma\rangle \to \sigma'.$$

Note that c is always executed at least once.

2(a) [10 points]. Complete the definition of the **until** operator by filling in the boxes in the following rules:

Answer:

$$\frac{\langle c, \boxed{\sigma}\rangle \to \boxed{\sigma'} \quad \langle b, \boxed{\sigma'}\rangle \to \boxed{\textbf{true}}}{\langle \textbf{do } c \textbf{ until } b, \sigma\rangle \to \sigma'} \quad \text{[until-1]}$$

$$\frac{\langle c, \boxed{\sigma}\rangle \to \boxed{\sigma''} \quad \langle b, \boxed{\sigma''}\rangle \to \boxed{\textbf{false}} \quad \langle \textbf{do } c \textbf{ until } b, \boxed{\sigma''}\rangle \to \boxed{\sigma'}}{\langle \textbf{do } c \textbf{ until } b, \sigma\rangle \to \sigma'} \quad \text{[until-2]}$$

2(b) [10 points]. Suppose

$$c_{\text{do}} \equiv \textbf{do } X := X - 1 \textbf{ until } X \leq 0,$$

and suppose $\sigma_{\mathbf{a}}$ is a state such that $\sigma_{\mathbf{a}}(X) = 2$. Then $\Vdash \langle c_{\text{do}}, \sigma_{\mathbf{a}}\rangle \to \sigma_{\mathbf{a}}[0/X]$.

Draw the derivation tree for $\Vdash \langle c_{\text{do}}, \sigma_{\mathbf{a}}\rangle \to \sigma_{\mathbf{a}}[0/X]$.

Answer: We will abbreviate $X := X - 1$ as c_1, and $X \leq 0$ as b. Then the derivation tree looks like:

$$\frac{\langle c_1, \sigma_a\rangle \to \sigma_a[1/X], \quad \langle b, \sigma_a[1/X]\rangle \to \mathbf{false}, \quad \dfrac{\langle c_1, \sigma_a[1/X]\rangle \to \sigma_a[0/X], \quad \langle b, \sigma_a[0/X]\rangle \to \mathbf{true}}{\langle c_{do}, \sigma_a[1/X]\rangle \to \sigma_a[0/X]}}{\langle c_{do}, \sigma_a\rangle \to \sigma_a[0/X]}$$

We did not mean to require you to draw the derivation trees for the boolean and arithmetic subexpressions of c_{do} (this wasn't evident from the phrasing of the question). No points were taken off for any mistakes made there.

By way of illustration we give the derivation tree for the boolean evaluation

$$\langle b, \sigma_a[1/X]\rangle \to \mathbf{false} \equiv \langle X \leq 0, \sigma_a[1/X]\rangle \to \mathbf{false},$$

namely,

$$\frac{\langle X, \sigma_a[1/X]\rangle \to 1, \quad \langle 0, \sigma_a[1/X]\rangle \to 0}{\langle X \leq 0, \sigma_a[1/X]\rangle \to \mathbf{false}}.$$

2(c) [10 points]. Now suppose σ_b is a state such that $\sigma_b(X) = n$, where $n \geq 1$. Then $\Vdash \langle c_{do}, \sigma_b\rangle \to \sigma_b[0/X]$.

How many times is the rule [until-1] used in the derivation of $\Vdash \langle c_{do}, \sigma_b\rangle \to \sigma_b[0/X]$?

Answer: One time.

How many times is the rule [until-2] used in the derivation of $\Vdash \langle c_{do}, \sigma_b\rangle \to \sigma_b[0/X]$?

Answer: $(n-1)$ times.

Problem 3 [40 points]. Recall the following sets, which were defined in Winskel's book (the exact definitions can be found in Appendix C):

- $\text{loc}(a)$ is the set of locations appearing in an arithmetic expression a;
- $\text{loc}(b)$ is the set of locations appearing in a boolean expression b;
- $\text{loc}(c)$ is the set of locations appearing in a command c; and
- $\text{loc}_L(c)$ is the set of locations appearing on the left-hand side of assignment statements in command c.

Note that if $X \notin \text{loc}_L(c)$, then the value of X cannot be changed by the execution of c. Intuitively, then, $\text{loc}_L(c)$ is the set of locations which might be *written* during the evaluation of c.

We want you to define $\text{loc}_{\text{read}}(c)$, the set of locations in command c which might be *read* during the evaluation of c. For example, if

$$c_a \equiv \textbf{if } W \leq 0 \textbf{ then } X := Y + 1 \textbf{ else } Z := X + Y,$$

then we should have $\text{loc}_{\text{read}}(c_a) = \{W, X, Y\}$.

3(a) [16 points]. Give a definition of $\text{loc}_{\text{read}}(c)$ by induction on the structure of commands. We have filled in the case $c \equiv c_0; c_1$ for you; you must fill in the other cases. You may use $\text{loc}_L(c)$, $\text{loc}(a)$, $\text{loc}(b)$, or $\text{loc}(c)$ in your definition if you so desire.

- $\text{loc}_{\text{read}}(c_0; c_1) = \text{loc}_{\text{read}}(c_0) \cup \text{loc}_{\text{read}}(c_1)$.

Answer:

- $\text{loc}_{\text{read}}(\textbf{skip}) = \emptyset$
- $\text{loc}_{\text{read}}(X := a) = \text{loc}(a)$
- $\text{loc}_{\text{read}}(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) = \text{loc}(b) \cup \text{loc}_{\text{read}}(c_0) \cup \text{loc}_{\text{read}}(c_1)$
- $\text{loc}_{\text{read}}(\textbf{while } b \textbf{ do } c') = \text{loc}(b) \cup \text{loc}_{\text{read}}(c')$

3(b) [4 points]. Give a simple command c such that

$$\text{loc}(c) = \text{loc}_L(c) = \text{loc}_{\text{read}}(c) \neq \emptyset.$$

Answer: One correct answer is $c \equiv X := X$.

3(c) [20 points]. It can be proved, by structural induction on commands c, that

$$\text{loc}(c) = \text{loc}_{\text{read}}(c) \cup \text{loc}_L(c).$$

We have filled in the case $c \equiv c_0; c_1$ of the proof for you; you must fill in the proof for the cases $c \equiv X := a$ and $c \equiv \textbf{while } b \textbf{ do } c'$. (You need not prove the remaining cases, $c \equiv \textbf{skip}$ and $c \equiv \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1$.)

- $c \equiv c_0; c_1$: By definition,

$$\text{loc}(c) = \text{loc}(c_0) \cup \text{loc}(c_1).$$

And by induction,

$$\text{loc}(c_0) = \text{loc}_{\textbf{read}}(c_0) \cup \text{loc}_L(c_0),$$
$$\text{loc}(c_1) = \text{loc}_{\textbf{read}}(c_1) \cup \text{loc}_L(c_1).$$

Therefore

$$\text{loc}(c) = \text{loc}_{\textbf{read}}(c_0) \cup \text{loc}_{\textbf{read}}(c_1) \cup \text{loc}_L(c_0) \cup \text{loc}_L(c_1),$$

and by the definition of $\text{loc}_{\textbf{read}}$ and loc_L,

$$\text{loc}(c) = \text{loc}_{\textbf{read}}(c) \cup \text{loc}_L(c),$$

as desired.

Answer:

- $c \equiv X := a$: By definition,

$$\text{loc}(X := a) = \{X\} \cup \text{loc}(a).$$

Since by definition,

$$\text{loc}_{\textbf{read}}(X := a) = \text{loc}(a),$$
$$\text{loc}_L(X := a) = \{X\},$$

clearly we have

$$\text{loc}(X := a) = \text{loc}_{\textbf{read}}(X := a) \cup \text{loc}_L(X := a),$$

as desired.

- $c \equiv \textbf{while } b \textbf{ do } c'$: By definition,

$$\text{loc}(c) = \text{loc}(b) \cup \text{loc}(c').$$

And by induction, $\text{loc}(c') = \text{loc}_{\textbf{read}}(c') \cup \text{loc}_L(c')$. Thus we have

$$\text{loc}(c) = \text{loc}(b) \cup \text{loc}_{\textbf{read}}(c') \cup \text{loc}_L(c').$$

Since by definition,

$$\text{loc}_{\textbf{read}}(c) = \text{loc}(b) \cup \text{loc}_{\textbf{read}}(c'),$$
$$\text{loc}_L(c) = \text{loc}_L(c'),$$

we have $\text{loc}(c) = \text{loc}_{\textbf{read}}(c) \cup \text{loc}_L(c')$.

Grade Statistics for Quiz 1

Number of quizzes taken: 11

Grade range: 61–95

Mean: 83

Histogram:

```
  0–4:
  5–9:
10–14:
15–19:
20–24:
25–29:
30–34:
35–39:
40–44:
45–49:
50–54:
55–59:
60–64:  *
65–69:  *
70–74:  *
75–79:
80–84:  *
85–89:  ***
90–94:  **
95–99:  **
```

Proof of Compositionality for IMP

The Meaning of IMP Programs

We can define an abstract "meaning" for arithmetic expressions, boolean expressions, and commands based on their evaluation. (The language **IMP** and its evaluation are summarized in Appendices.)

The meaning of an arithmetic expression a is the function $[\![a]\!] : \Sigma \to \mathbf{N}$ defined as follows:

$$[\![a]\!](\sigma) = n, \text{ where } n \text{ is the unique number such that } \langle a, \sigma\rangle \to n.$$

Similarly, the meaning of a boolean expression b is the function $[\![b]\!] : \Sigma \to \mathbf{T}$ defined as follows:

$$[\![b]\!](\sigma) = t, \text{ where } t \text{ is the unique truth value such that } \langle b, \sigma\rangle \to t.$$

Note that $[\![a]\!]$ and $[\![b]\!]$ are well-defined because the evaluation of arithmetic and boolean expressions is deterministic and total.

The execution of commands is deterministic but not necessarily total. A configuration $\langle c, \sigma\rangle$ might not terminate; but if it does, it will terminate in a unique state σ'. Thus we can define the meaning of c to be the partial function $[\![c]\!] : \Sigma \rightharpoonup \Sigma$ determined by:

$$[\![c]\!](\sigma) = \sigma' \text{ iff } \langle c, \sigma\rangle \to \sigma'.$$

Contexts

Informally, a context is a "term with a hole". For example, an arithmetic expression context is an arithmetic expression with a hole; when the hole is filled in with an arithmetic expression, the result is a new arithmetic expression. A full definition of arithmetic expression contexts follows:

$$\begin{aligned} A[\cdot] ::= {} & [\cdot] \\ \mid {} & a \\ \mid {} & A_1[\cdot]\ \underline{\text{op}}\ A_2[\cdot] \end{aligned}$$

(Here $\underline{\text{op}}$ ranges over $\{+, -, \times\}$).

[Note that I am using a slightly different definition of context than the one Albert used in class. My definition allows a context to have zero, one, or more

holes. This makes proofs slightly easier (by reducing the number of cases that need to be considered), but is not an important difference.]

If $A[\cdot]$ is a context and a is an arithmetic expression, then $A[a]$ is the arithmetic expression obtained by "plugging in" a into the holes of $A[\cdot]$. It can be defined by induction on the structure of $A[\cdot]$:

- If $A[\cdot] \equiv [\cdot]$, then $A[a] \equiv a$.
- If $A[\cdot] \equiv a'$, then $A[a] \equiv a'$.
- If $A[\cdot] \equiv A_1[\cdot] \ \underline{\text{op}}\ A_2[\cdot]$, then $A[a] \equiv A_1[a] \ \underline{\text{op}}\ A_2[a]$.

For example, if $A'[\cdot]$ is defined by

$$A'[\cdot] \equiv [\cdot] + ([\cdot] + 5),$$

then

$$A'[6] \equiv 6 + (6 + 5).$$

The boolean contexts and command contexts are defined similarly:

$$\begin{aligned} B[\cdot] ::= &\ [\cdot] \\ \mid &\ b \\ \mid &\ \neg B[\cdot] \\ \mid &\ B_1[\cdot] \wedge B_2[\cdot] \\ \mid &\ B_1[\cdot] \vee B_2[\cdot] \end{aligned}$$

$$\begin{aligned} C[\cdot] ::= &\ [\cdot] \\ \mid &\ c \\ \mid &\ C_1[\cdot]; C_2[\cdot] \\ \mid &\ \textbf{if } b \textbf{ then } C_1[\cdot] \textbf{ else } C_2[\cdot] \\ \mid &\ \textbf{while } b \textbf{ do } C'[\cdot] \end{aligned}$$

The "plugging in" operation for boolean and command contexts is defined in the same way as for arithmetic contexts.

Compositionality

Contexts are a technical device that lets us talk about "pieces" of an expression or program. Most important, they give us a precise method of removing a sub-part from a program and replacing it by another another sub-part.

For example, one sub-part of an accounting package P might be a sorting algorithm c. If c' is another, faster sorting algorithm, we will be very interested in whether we can replace c with c' in P without changing the answers reported by P. With contexts we can express this situation by:

> "The program $P \equiv C[c]$ for some context $C[\cdot]$, and we want to know whether $P' \equiv C[c']$ computes the same answers."

Compositionality is a property that says that if the *meaning* of c is the same as the meaning of c', then it is safe to replace c by c' in $C[\cdot]$ — *for any* $C[\cdot]$*!* In a compositional language we can reason about sub-parts of programs (like sorting packages, etc.) based solely on their meanings, and be confident that our reasoning will hold no matter how the sub-parts are used.

We will first prove compositionality for arithmetic expressions. We will need the following lemma:

Lemma 1. For all arithmetic expressions a_1, a_2 and operators $\underline{\text{op}}$,

$$[\![a_1 \ \underline{\text{op}}\ a_2]\!] = [\![a_1]\!] \text{ op } [\![a_2]\!].$$

Proof: Similar to the proof of Lemma 2 below. ■

Theorem 1 (Compositionality of arithmetic expressions). For all arithmetic expressions a_1, a_2,

$$[\![a_1]\!] = [\![a_2]\!] \Rightarrow (\forall A[\cdot] \,.\, ([\![A[a_1]]\!] = [\![A[a_2]]\!])).$$

Proof: Assume $[\![a_1]\!] = [\![a_2]\!]$. We will show that for all $A[\cdot]$, $([\![A[a_1]]\!] = [\![A[a_2]]\!])$. The proof is by induction on the structure of $A[\cdot]$:

- $A[\cdot] \equiv [\cdot]$: Then $A[a_1] = a_1$ and $A[a_2] = a_2$, so

$$([\![A[a_1]]\!] = [\![A[a_2]]\!]) \text{ iff } ([\![a_1]\!] = [\![a_2]\!]),$$

  and the right-hand side holds by assumption.

- $A[\cdot] \equiv a$: Then $A[a_1] = a = A[a_2]$, so $[\![A[a_1]]\!] = [\![a]\!] = [\![A[a_2]]\!]$.

- $A[\cdot] \equiv A_1[\cdot] \ \underline{\text{op}}\ A_2[\cdot]$: Then

$$A[a_1] = A_1[a_1] \ \underline{\text{op}}\ A_2[a_1],$$

  by the definition of "plugging in" an expression into a context. Thus

$$[\![A[a_1]]\!] = [\![A_1[a_1] \ \underline{\text{op}}\ A_2[a_1]]\!].$$

Then by Lemma 1 above we have

$$[\![A[a_1]]\!] = [\![A_1[a_1]]\!] \text{ op } [\![A_2[a_1]]\!]. \quad (1)$$

Now $A_1[\cdot]$ and $A_2[\cdot]$ are smaller than $A[\cdot]$, so we can apply the induction hypothesis to them. That is, we have

$$[\![A_1[a_1]]\!] = [\![A_1[a_2]]\!],$$
$$[\![A_2[a_1]]\!] = [\![A_2[a_2]]\!].$$

Substituting into (1), we have

$$[\![A[a_1]]\!] = [\![A_1[a_2]]\!] \text{ op } [\![A_2[a_2]]\!].$$

Now we work backwards: by Lemma 1 we have

$$[\![A[a_1]]\!] = [\![A_1[a_2] \ \underline{\text{op}}\ A_2[a_2]]\!],$$

and by the definition of plugging in,

$$[\![A[a_1]]\!] = [\![A[a_2]]\!],$$

as desired.

■

We can prove compositionality for boolean expressions in much the same way:

Theorem 2 (Compositionality of boolean expressions). For all boolean expressions b_1, b_2,

$$[\![b_1]\!] = [\![b_2]\!] \Rightarrow (\forall B[\cdot]\ .\ [\![B[b_1]]\!] = [\![B[b_2]]\!]).$$

Proof: Omitted. ■

Just as Lemma 1 related the syntactic operators of arithmetic expressions to certain mathematical functions, we can show that the syntactic connectives of commands have mathematical counterparts:

Lemma 2. For all commands c_1 and c_2,

$$[\![(c_1; c_2)]\!] = [\![c_2]\!] \circ [\![c_1]\!].$$

Proof: We show that for all σ and σ', $[\![(c_1;c_2)]\!](\sigma) = \sigma'$ iff $([\![c_2]\!] \circ [\![c_1]\!])(\sigma) = \sigma'$.

$[\![(c_1;c_2)]\!](\sigma) = \sigma'$

iff	$\langle (c_1;c_2), \sigma\rangle \to \sigma'$	by definition of $[\![\cdot]\!]$,
iff	$\langle c_1, \sigma\rangle \to \sigma''$ and $\langle c_2, \sigma''\rangle \to \sigma'$ for some σ''	by the evaluation rule for ";",
iff	$[\![c_1]\!](\sigma) = \sigma''$ and $[\![c_2]\!](\sigma'') = \sigma'$ for some σ'',	by definition of $[\![\cdot]\!]$,
iff	$([\![c_2]\!] \circ [\![c_1]\!])(\sigma) = \sigma'$	by the definition of "$\circ$". ■

Finally, we are ready to prove compositionality for the full language.

Theorem 3 (Compositionality of IMP). For all commands c_1, c_2,

$$[\![c_1]\!] = [\![c_2]\!] \Rightarrow (\forall C[\cdot] \,.\; [\![C[c_1]]\!] = [\![C[c_2]]\!]).$$

Proof: Assume $[\![c_1]\!] = [\![c_2]\!]$. We want to show that for all command contexts $C[\cdot]$, $[\![C[c_1]]\!] = [\![C[c_2]]\!]$. The proof is by induction on the structure of $C[\cdot]$:

- $C[\cdot] \equiv [\cdot]$. Then $[\![C[c_1]]\!] = [\![c_1]\!]$ and $[\![C[c_2]]\!] = [\![c_2]\!]$, and these are equal by assumption.
- $C[\cdot] \equiv c$, for some command c. Then $[\![C[c_1]]\!] = [\![C[c_2]]\!] = [\![c]\!]$.
- $C[\cdot] \equiv (C_1[\cdot]; C_2[\cdot])$. Then

$$\begin{array}{rcll} [\![C[c_1]]\!] & = & [\![(C_1[c_1]; C_2[c_1])]\!] & \text{by definition of "plugging in"}, \\ & = & [\![C_2[c_1]]\!] \circ [\![C_1[c_1]]\!] & \text{by Lemma 2}, \\ & = & [\![C_2[c_2]]\!] \circ [\![C_1[c_2]]\!] & \text{by induction on contexts}, \\ & = & [\![(C_1[c_2]; C_2[c_2])]\!] & \text{by Lemma 2}, \\ & = & [\![C[c_2]]\!] & \text{by definition of "plugging in"}. \end{array}$$

- $C[\cdot] \equiv$ **while** b **do** $C'[\cdot]$.

  Unless we have a lemma (like Lemma 2 above) relating the meaning of a **while** command to the meanings of its sub-parts, we will not be able to prove this case by induction on the structure of contexts alone. [See the solution to Problem 4 from Problem Set 1, as well as section 3.4 of Winskel's book for further discussion.] Since such a lemma would involve a complicated mathematical function (the equivalent of $\circ$ for **while**), we will take another approach.

  We instead do an entire *sub-proof* by induction on derivations.

  We prove that for all d, σ, and σ',

$$d \Vdash \langle \textbf{while } b \textbf{ do } C'[c_1], \sigma\rangle \to \sigma' \text{ iff } \Vdash \langle \textbf{while } b \textbf{ do } C'[c_2], \sigma\rangle \to \sigma'.$$

  The proof is by induction on the structure of d:

  - d is a derivation of the form

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } C'[c_1], \sigma\rangle \to \sigma}$$

    (so $\sigma' = \sigma$). Then clearly

$$\frac{\langle b, \sigma\rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } C'[c_2], \sigma\rangle \to \sigma}$$

    as desired.

– d is a derivation of the form

$$\dfrac{\langle b,\sigma\rangle \to \mathbf{true} \quad \dfrac{\vdots}{\langle C'[c_1],\sigma\rangle \to \sigma''} \quad \dfrac{\vdots}{\langle \mathbf{while}\ b\ \mathbf{do}\ C'[c_1],\sigma''\rangle \to \sigma'}}{\langle \mathbf{while}\ b\ \mathbf{do}\ C'[c_1],\sigma\rangle \to \sigma'}$$

Then

$$\begin{gathered} \Vdash \langle b,\sigma\rangle \to \mathbf{true}, \\ \Vdash \langle C'[c_1],\sigma\rangle \to \sigma'', \\ d_1 \Vdash \langle \mathbf{while}\ b\ \mathbf{do}\ C'[c_1],\sigma''\rangle \to \sigma'. \end{gathered}$$

Now

$$\begin{array}{lll} \multicolumn{3}{l}{\Vdash \langle C'[c_1],\sigma\rangle \to \sigma''} \\ \quad \text{iff} & [\![C'[c_1]]\!](\sigma) = \sigma'' & \text{by definition of } [\![\cdot]\!], \\ \quad \text{iff} & [\![C'[c_2]]\!](\sigma) = \sigma'' & \text{by induction on contexts,} \\ \quad \text{iff} & \Vdash \langle C'[c_2],\sigma\rangle \to \sigma'' & \text{by definition of } [\![\cdot]\!]. \end{array}$$

And

$$d_1 \Vdash \langle \mathbf{while}\ b\ \mathbf{do}\ C'[c_1],\sigma''\rangle \to \sigma' \quad \text{iff} \quad \Vdash \langle \mathbf{while}\ b\ \mathbf{do}\ C'[c_2],\sigma''\rangle \to \sigma',$$

by induction on *derivations*. Thus we have all of

$$\begin{gathered} \Vdash \langle b,\sigma\rangle \to \mathbf{true}, \\ \Vdash \langle C'[c_2],\sigma\rangle \to \sigma'', \\ \Vdash \langle \mathbf{while}\ b\ \mathbf{do}\ C'[c_2],\sigma''\rangle \to \sigma', \end{gathered}$$

and can conclude

$$\Vdash \langle \mathbf{while}\ b\ \mathbf{do}\ C'[c_2],\sigma\rangle \to \sigma',$$

as desired.

- $C[\cdot] \equiv \mathbf{if}\ b\ \mathbf{then}\ C_1[\cdot]\ \mathbf{else}\ C_2[\cdot]$: we omit this case.

∎

Appendix A: Syntax of IMP

For **Aexp**:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

For **Bexp**:

$$b ::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

For **Com**:

$$c ::= \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

Appendix B: Semantics of IMP

Evaluation of **Aexp**'s:

$\langle n, \sigma \rangle \to n$

$\langle X, \sigma \rangle \to \sigma(X)$

$\dfrac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n}$ where n is the sum of n_0 and n_1.

$\dfrac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 - a_1, \sigma \rangle \to n}$ where n is the result of subtracting n_1 from n_0.

$\dfrac{\langle a_0, \sigma \rangle \to n_0 \quad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 \times a_1, \sigma \rangle \to n}$ where n is the product of n_0 and n_1.

Evaluation of **Bexp**'s:

$\langle \mathbf{true}, \sigma \rangle \to \mathbf{true}$

$\langle \mathbf{false}, \sigma \rangle \to \mathbf{false}$

$\dfrac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to \mathbf{true}}$ if n and m are equal.

$\dfrac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to \mathbf{false}}$ if n and m are unequal.

$\dfrac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 \leq a_1, \sigma \rangle \to \mathbf{true}}$ if n is less than or equal to m.

$\dfrac{\langle a_0, \sigma \rangle \to n \quad \langle a_1, \sigma \rangle \to m}{\langle a_0 \leq a_1, \sigma \rangle \to \mathbf{false}}$ if n is not less than or equal to m.

$\dfrac{\langle b, \sigma \rangle \to \mathbf{true}}{\langle \neg b, \sigma \rangle \to \mathbf{false}} \qquad \dfrac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \neg b, \sigma \rangle \to \mathbf{true}}$

$\dfrac{\langle b_0, \sigma \rangle \to t_0 \quad \langle b_1, \sigma \rangle \to t_1}{\langle b_0 \wedge b_1, \sigma \rangle \to t}$
where t is **true** if $t_0 \equiv \mathbf{true}$ and $t_1 \equiv \mathbf{true}$, and is **false** otherwise.

$\dfrac{\langle b_0, \sigma \rangle \to t_0 \quad \langle b_1, \sigma \rangle \to t_1}{\langle b_0 \vee b_1, \sigma \rangle \to t}$
where t is **true** if $t_0 \equiv \mathbf{true}$ or $t_1 \equiv \mathbf{true}$, and is **false** otherwise.

Execution of **Com**'s:

$$\langle \textbf{skip}, \sigma \rangle \to \sigma$$

$$\frac{\langle a, \sigma \rangle \to m}{\langle X := a, \sigma \rangle \to \sigma[m/X]}$$

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{false} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma'' \rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma \rangle \to \sigma'}$$

Problem Set 4

Due: 27 October 1993.

Problem 1. Winskel Exercise 6.1.

Problem 2. Winskel Exercise 6.2.

Problem 3. Winskel Exercise 6.13.

Problem 4. Winskel Exercise 6.17.

Problem Set 5

Due: Friday, November 5, 1993.

Problem 1. Lemma 1 from Handout 22 says that each syntactic operator of arithmetic expressions has a corresponding operator on the meaning of arithmetic expressions. For this problem, we ask you to prove Lemma 1 for the syntactic operator $+$ and its counterpart $+_{\mathrm{p}}$, the *pointwise sum* operator.

We define $+_{\mathrm{p}} : (\Sigma \rightarrow \mathbf{N}) \times (\Sigma \rightarrow \mathbf{N}) \rightarrow (\Sigma \rightarrow \mathbf{N})$ as follows:

$$(m_1 +_{\mathrm{p}} m_2)(\sigma) = n,$$

where n is the sum of $m_1(\sigma)$ and $m_2(\sigma)$.

Prove that for all arithmetic expressions a_1, a_2,

$$[\![a_1 + a_2]\!] = [\![a_1]\!] +_{\mathrm{p}} [\![a_2]\!].$$

Problem 2. Let $\mathrm{cond} : (\Sigma \rightarrow \mathbf{T}) \times (\Sigma \rightarrow \Sigma) \times (\Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma)$ be defined as follows:

$$\mathrm{cond}(m_b, m_{c_1}, m_{c_2})(\sigma) = \begin{cases} m_{c_1}(\sigma) & \text{if } m_b(\sigma) = \mathbf{true}, \\ m_{c_2}(\sigma) & \text{if } m_b(\sigma) = \mathbf{false}. \end{cases}$$

(a) Prove that

$$[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!] = \mathrm{cond}([\![b]\!], [\![c_1]\!], [\![c_2]\!]).$$

(b) We omitted the **if** case of the proof of compositionality for **IMP** (Theorem 3 from Handout 22). Using the result from (a), prove the missing **if** case (i.e., the case $C[\cdot] \equiv \mathbf{if}\ b\ \mathbf{then}\ C_1[\cdot]\ \mathbf{else}\ C_2[\cdot]$).

Problem 3. Prove Lemma 6.8 from Winskel's book.

Problem 4.

(a) Using the definition of validity, prove that for all assertions A and integer variables j,

$$\text{if} \models A, \text{ then} \models \forall j.A.$$

(b) Give an assertion A, state σ, and interpretation I such that

$$\sigma \models^I A,$$

but

$$\sigma \not\models^I \forall j.A.$$

(c) Conclude that for your A from part (b), $A \Rightarrow \forall j.A$ is not valid.

Problem Set 4 Solution

Problem 1. Winskel Exercise 6.1. [I restate the problem here in slightly different form than Winskel.]

Write down an assertion *Prime*$(i) \in$ **Assn** with one free integer variable i which expresses that i is a prime number, *i.e.* it is required that:

$$\sigma \models^I \mathit{Prime}(i) \text{ iff } I(i) \text{ is a prime number.}$$

Answer: First we define, for any integer variables j and k, the assertion "j divides k":

$$j \textit{ divides } k \equiv \exists \ell . j \times \ell = k.$$

There are many possible definitions of *Prime*(i); this is one example:

$$\mathit{Prime}(i) \equiv (2 \leq i) \wedge \big((\forall j . 2 \leq j \wedge j \textit{ divides } i) \Rightarrow i = j\big).$$

Problem 2. Winskel Exercise 6.2. [There was a typo in Winskel's statement of the problem, so I restate it here.]

Define a formula *LCM*$(i, j, k) \in$ **Assn** with free integer variables variables i,j, and k, which means "i is the least common multiple of j and k," *i.e.* it is required that:

$$\sigma \models^I \mathit{LCM}(i, j, k) \text{ iff } I(i) \text{ is the least common multiple of } I(j) \text{ and } I(k).$$

(Hint: The least common multiple of two numbers is the smallest non-negative integer divisible by both.)

Answer: First we define the assertion *CM*(i, j, k), meaning "i is a common multiple of j and k":

$$\mathit{CM}(i, j, k) \equiv (k \textit{ divides } i) \wedge (j \textit{ divides } i).$$

As Winskel has stated it, the least common multiple is the smallest non-negative common multiple. For example, the least common multiple of -1 and -2 is 2. So we have the following definition of *LCM*:

$$\mathit{LCM}(i, j, k) \equiv (0 \leq i) \wedge \mathit{CM}(i, j, k) \wedge (\forall x . (0 \leq x) \wedge \mathit{CM}(x, j, k) \Rightarrow i \leq x).$$

Problem 3. Winskel Exercise 6.13.

Prove, using the Hoare rules, the correctness of the partial correctness assertion:

$$
\begin{array}{l}
\{1 \leq N\} \\
P := 0; \\
C := 1; \\
(\textbf{while } C \leq N \textbf{ do } P := P + M; C := C + 1) \\
\{P = M \times N\}
\end{array}
$$

Answer: By the rules for assignment and sequencing, we can get

$$
\frac{\{1 \leq N \wedge 0 = 0 \wedge 1 = 1\}P := 0\{1 \leq N \wedge P = 0 \wedge 1 = 1\} \quad \{1 \leq N \wedge P = 0 \wedge 1 = 1\}C := 1\{1 \leq N \wedge P = 0 \wedge C = 1\}}{\{1 \leq N \wedge 0 = 0 \wedge 1 = 1\}P := 0; C := 1\{1 \leq N \wedge P = 0 \wedge C = 1\}}
$$

which the rule of consequence can simplify to

$$\{1 \leq N\}P := 0; C := 1\{1 \leq N \wedge P = 0 \wedge C = 1\}.$$

For any assertion I, the assignment and sequencing rules give

$$
\frac{\{I[C+1/C][P+M/P]\}P := P + M\{I[C+1/C]\} \quad \{I[C+1/C]\}C := 1\{I\}}{\{I[C+1/C][P+M/P]\}P := P + M; C := C + 1\{I\}}
$$

Now, let I be $(P = M \times (C-1)) \wedge (C \leq N + 1)$. Then,

$$
\begin{aligned}
I[C+1/C][P+M/P] &\equiv (P + M = M \times (C + 1 - 1)) \wedge (C + 1 \leq N + 1) \\
&\equiv (P = M \times (C - 1)) \wedge (C \leq N) \\
&\equiv (P = M \times (C - 1)) \wedge (C \leq N + 1) \wedge (C \leq N) \\
&\equiv I \wedge (C \leq N)
\end{aligned}
$$

And thus, we have

$$\{I \wedge (C \leq N)\}P := P + M; C := C + 1\{I\},$$

so, by the rule for while-loops

$$
\frac{\{I \wedge (C \leq N)\}P := P + M; C := C + 1\{I\}}{\{I\}\textbf{while}\, C \leq N\, \textbf{do}\, P := P + M; C := C + 1\{I \wedge \neg(C \leq N)\}}
$$

Finally, since

$$(1 \leq N \wedge P = 0 \wedge C = 1) \Rightarrow I$$

and

$$(I \wedge \neg(C \leq N)) \Rightarrow (P = N \times M)$$

we can use the rule of consequence to get

$$\begin{array}{l}\{1 \leq N \wedge P = 0 \wedge C = 1\} \\ \textbf{while}\, C \leq N \,\textbf{do}\, P := P + M; C := C + 1 \\ \{P = N \times M\}\end{array}$$

and the rule for sequencing to get

$$\begin{array}{l}\{1 \leq N\} \\ P := 0; C := 1; (\textbf{while}\, C \leq N \,\textbf{do}\, P := P + M; C := C + 1) \\ \{P = M \times N\}.\end{array}$$

■

Problem 4. Winskel Exercise 6.17.

Provide a Hoare rule for the repeat construct and prove it sound.

Answer: The repeat construct is defined by the following rules:

$$\frac{\langle c, \sigma\rangle \to \sigma' \quad \langle b, \sigma'\rangle \to \textbf{true}}{\langle \textbf{repeat}\ c\ \textbf{until}\ b, \sigma\rangle \to \sigma'} \quad \text{[repeat-1]}$$

$$\frac{\langle c, \sigma\rangle \to \sigma'' \quad \langle b, \sigma''\rangle \to \textbf{false} \quad \langle \textbf{repeat}\ c\ \textbf{until}\ b, \sigma''\rangle \to \sigma'}{\langle \textbf{repeat}\ c\ \textbf{until}\ b, \sigma\rangle \to \sigma'} \quad \text{[repeat-2]}$$

We will prove that the following Hoare rule is sound:

$$\frac{\{A\}c\{A\}}{\{A\}\textbf{repeat}\ c\ \textbf{until}\ b\{A \wedge b\}}$$

Assume that $\models \{A\}c\{A\}$, and let $r \equiv \textbf{repeat}\ c\ \textbf{until}\ b$. We want to show that $\models \{A\}r\{A \wedge b\}$. This is equivalent to showing that

$$\forall \sigma, I.(\sigma \models^I A) \Rightarrow (\llbracket r \rrbracket \sigma \models^I A \wedge b).$$

Since $\llbracket r \rrbracket \sigma = \sigma'$ iff $\langle r, \sigma\rangle \to \sigma'$, this is equivalent to showing

$$\forall d, \sigma, I.(\sigma \models^I A \wedge d \Vdash \langle r, \sigma\rangle \to \sigma') \Rightarrow \sigma' \models^I A \wedge b.$$

We show this by induction on d. That is, for all d, we show that

$$\forall \sigma, I.(\sigma \models^I A \wedge d \Vdash \langle r, \sigma\rangle \to \sigma') \Rightarrow \sigma' \models^I A \wedge b. \tag{1}$$

- Suppose d is a derivation ending with the rule [repeat-1]:

$$d = \cfrac{\cfrac{\vdots}{\langle c,\sigma\rangle \to \sigma'} \quad \langle b,\sigma'\rangle \to \textbf{true}}{\langle \textbf{repeat}\ c\ \textbf{until}\ b,\sigma\rangle \to \sigma'}$$

Then if $\sigma \models^I A$, since $\{A\}c\{A\}$ and $\langle c,\sigma\rangle \to \sigma'$, we have $\sigma' \models^I A$.
And since $\langle b,\sigma'\rangle \to \textbf{true}$, by Proposition 6.4 we have $\sigma' \models^I b$.
Thus $\sigma' \models^I (A \wedge b)$, proving the induction hypothesis (1).

- Suppose d is a derivation ending with the rule [repeat-2]:

$$d = \cfrac{\cfrac{\vdots}{\langle c,\sigma\rangle \to \sigma''} \quad \langle b,\sigma''\rangle \to \textbf{false} \quad \cfrac{\vdots}{\langle \textbf{repeat}\ c\ \textbf{until}\ b,\sigma''\rangle \to \sigma'}}{\langle \textbf{repeat}\ c\ \textbf{until}\ b,\sigma\rangle \to \sigma'}$$

Then if $\sigma \models^I A$, since $\{A\}c\{A\}$ and $\langle c,\sigma\rangle \to \sigma''$, we have $\sigma'' \models^I A$.
And then by the induction hypothesis (1) applied to the subderivation d':

$$d' = \cfrac{\vdots}{\langle \textbf{repeat}\ c\ \textbf{until}\ b,\sigma''\rangle \to \sigma'},$$

we have $\sigma' \models^I (A \wedge b)$, as desired.

■

Problem Set 5 Solution

Problem 1. Lemma 1 from Handout 22 says that each syntactic operator of arithmetic expressions has a corresponding operator on the meaning of arithmetic expressions. For this problem, we ask you to prove Lemma 1 for the syntactic operator $+$ and its counterpart $+_{\mathrm{p}}$, the *pointwise sum* operator.

We define $+_{\mathrm{p}} : (\Sigma \to \mathbf{N}) \times (\Sigma \to \mathbf{N}) \to (\Sigma \to \mathbf{N})$ as follows:

$$(m_1 +_{\mathrm{p}} m_2)(\sigma) = n,$$

where n is the sum of $m_1(\sigma)$ and $m_2(\sigma)$.

Prove that for all arithmetic expressions a_1, a_2,

$$[\![a_1 + a_2]\!] = [\![a_1]\!] +_{\mathrm{p}} [\![a_2]\!].$$

Answer: Both $[\![a_1 + a_2]\!]$ and $([\![a_1]\!] +_{\mathrm{p}} [\![a_2]\!])$ are functions from states to numbers. To show that the two functions are equal, we simply show that they agree on all arguments. That is, for all states σ, we show

$$[\![a_1 + a_2]\!](\sigma) = ([\![a_1]\!] +_{\mathrm{p}} [\![a_2]\!])(\sigma).$$

We reason as follows:

$[\![a_1 + a_2]\!](\sigma) = n$
iff $\langle a_1 + a_2, \sigma \rangle \to n$ (by definition of $[\![\cdot]\!]$),
iff $\langle a_1, \sigma \rangle \to n_1$ and $\langle a_2, \sigma \rangle \to n_2$, where n is the sum of n_1 and n_2 (by the evaluation of arithmetic expressions),
iff $[\![a_1]\!](\sigma) = n_1$ and $[\![a_2]\!](\sigma) = n_2$, where n is the sum of n_1 and n_2 (by definition of $[\![\cdot]\!]$),
iff $([\![a_1]\!] +_{\mathrm{p}} [\![a_2]\!])(\sigma) = n$ (by definition of $+_{\mathrm{p}}$).

■

Problem 2. Let $\mathrm{cond} : (\Sigma \to \mathbf{T}) \times (\Sigma \to \Sigma) \times (\Sigma \to \Sigma) \to (\Sigma \to \Sigma)$ be defined as follows:

$$\mathrm{cond}(m_b, m_{c_1}, m_{c_2})(\sigma) = \begin{cases} m_{c_1}(\sigma) & \text{if } m_b(\sigma) = \mathbf{true}, \\ m_{c_2}(\sigma) & \text{if } m_b(\sigma) = \mathbf{false}. \end{cases}$$

(a) Prove that

$$[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!] = \text{cond}([\![b]\!], [\![c_1]\!], [\![c_2]\!]).$$

Answer: Just as in Problem 1, we will show that for all states σ,

$$[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!](\sigma) = \text{cond}([\![b]\!], [\![c_1]\!], [\![c_2]\!])(\sigma).$$

By the definition of $[\![\cdot]\!]$, we have

$$[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!](\sigma) = \sigma' \text{ iff } \langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \sigma'.$$

By the definition of the execution of commands, we have

$$\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \sigma' \text{ iff } \begin{cases} \langle b, \sigma\rangle \to \textbf{true} \text{ and } \langle c_1, \sigma\rangle \to \sigma', \\ \text{or } \langle b, \sigma\rangle \to \textbf{false} \text{ and } \langle c_2, \sigma\rangle \to \sigma'. \end{cases}$$

And by the definition of $[\![\cdot]\!]$,

$$\begin{array}{rcl} \langle b, \sigma\rangle \to \textbf{true} & \text{iff} & [\![b]\!](\sigma) = \textbf{true}, \\ \langle c_1, \sigma\rangle \to \sigma' & \text{iff} & [\![c_1]\!](\sigma) = \sigma', \\ \langle b, \sigma\rangle \to \textbf{false} & \text{iff} & [\![b]\!](\sigma) = \textbf{false}, \\ \langle c_2, \sigma\rangle \to \sigma' & \text{iff} & [\![c_2]\!](\sigma) = \sigma'. \end{array}$$

Combining all of the above, we have

$$[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!](\sigma) = \sigma' \text{ iff } \begin{cases} [\![b]\!](\sigma) = \textbf{true} \text{ and } [\![c_1]\!](\sigma) = \sigma', \\ \text{or } [\![b]\!](\sigma) = \textbf{false} \text{ and } [\![c_2]\!](\sigma) = \sigma'. \end{cases}$$

Then by the definition of cond, we have

$$[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!](\sigma) = \sigma' \text{ iff } \text{cond}([\![b]\!], [\![c_1]\!], [\![c_2]\!])(\sigma) = \sigma',$$

as desired. ■

(b) We omitted the **if** case of the proof of compositionality for **IMP** (Theorem 3 from Handout 22). Using the result from (a), prove the missing **if** case (i.e., the case $C[\cdot] \equiv \textbf{if } b \textbf{ then } C_1[\cdot] \textbf{ else } C_2[\cdot]$).

Answer: We are assuming that $[\![c_1]\!] = [\![c_2]\!]$, and we want to show that for all command contexts $C[\cdot]$, we have $[\![C[c_1]]\!] = [\![C[c_2]]\!]$. The proof was by induction on the structure of $C[\cdot]$, and we only need to prove the case $C[\cdot] \equiv \textbf{if } b \textbf{ then } C_1[\cdot] \textbf{ else } C_2[\cdot]$.

Then we reason as follows:

$$\begin{array}{rcll} [\![C[c_1]]\!] & = & [\![\textbf{if } b \textbf{ then } C_1[c_1] \textbf{ else } C_2[c_1]]\!] & \text{by defn. “plugging in”}, \\ & = & \text{cond}([\![b]\!], [\![C_1[c_1]]\!], [\![C_2[c_1]]\!]) & \text{by part (a)}, \\ & = & \text{cond}([\![b]\!], [\![C_1[c_2]]\!], [\![C_2[c_2]]\!]) & \text{by induction on contexts}, \\ & = & [\![\textbf{if } b \textbf{ then } C_1[c_2] \textbf{ else } C_2[c_2]]\!] & \text{by part (a)}, \\ & = & [\![C[c_2]]\!] & \text{by defn. “plugging in”}. \end{array}$$

■

Problem 3. Prove Lemma 6.8 from Winskel's book.

Answer: We restate the lemma here:

Lemma 6.8 If $a, a_0 \in$ **Aexpv** and $X \in$ **Loc**, then for all interpretations I and states σ,

$$\mathcal{A}v[\![a_0[a/X]]\!]I\sigma = \mathcal{A}v[\![a_0]\!]I(\sigma[(\mathcal{A}v[\![a]\!]I\sigma)/X]).$$

Proof: By structural induction on a_0. Let $\sigma' = \sigma[(\mathcal{A}v[\![a]\!]I\sigma)/X]$; then we want to show

$$\mathcal{A}v[\![a_0[a/X]]\!]I\sigma = \mathcal{A}v[\![a_0]\!]I\sigma'.$$

- If X does not appear in a_0, then

  $$\mathcal{A}v[\![a_0[a/X]]\!]I\sigma = \mathcal{A}v[\![a_0]\!]I\sigma,$$

  by the definition of substitution. And since $\sigma(Y) = \sigma'(Y)$ for all $Y \not\equiv X$, and X does not appear in a_0, we have

  $$\mathcal{A}v[\![a_0]\!]I\sigma = \mathcal{A}v[\![a_0]\!]I\sigma'.$$

  Thus for the cases $a_0 \equiv n$, $a_0 \equiv i$, and $a_0 \equiv Y \not\equiv X$, we have

  $$\mathcal{A}v[\![a_0[a/X]]\!]I\sigma = \mathcal{A}v[\![a_0]\!]I\sigma',$$

  as desired.

- $a_0 \equiv X$. Then

  $$\begin{aligned} \mathcal{A}v[\![a_0[a/X]]\!]I\sigma &= \mathcal{A}v[\![X[a/X]]\!]I\sigma, \\ &= \mathcal{A}v[\![a]\!]I\sigma && \text{by defn. substitution.} \end{aligned}$$

  And

  $$\begin{aligned} \mathcal{A}v[\![a_0]\!]I\sigma' &= \mathcal{A}v[\![X]\!]I\sigma', \\ &= \sigma'(X) && \text{by defn. } \mathcal{A}v[\![\cdot]\!], \\ &= (\sigma[(\mathcal{A}v[\![a]\!]I\sigma)/X])(X), \\ &= \mathcal{A}v[\![a]\!]I\sigma && \text{by defn. “patching” of } \sigma. \end{aligned}$$

- $a_0 \equiv a_1 + a_2$. Then

  $$\begin{aligned} \mathcal{A}v[\![a_0[a/X]]\!]I\sigma &= \mathcal{A}v[\![(a_1 + a_2)[a/X]]\!]I\sigma, \\ &= \mathcal{A}v[\![a_1[a/X] + a_2[a/X]]\!]I\sigma && \text{by defn. substitution,} \\ &= \mathcal{A}v[\![a_1[a/X]]\!]I\sigma \\ &\quad + \mathcal{A}v[\![a_2[a/X]]\!]I\sigma && \text{by defn. } \mathcal{A}v[\![\cdot]\!], \\ &= \mathcal{A}v[\![a_1]\!]I\sigma' \\ &\quad + \mathcal{A}v[\![a_2]\!]I\sigma' && \text{by induction on } a_0, \\ &= \mathcal{A}v[\![a_1 + a_2]\!]I\sigma' && \text{by defn. } \mathcal{A}v[\![\cdot]\!], \\ &= \mathcal{A}v[\![a_0]\!]I\sigma'. \end{aligned}$$

- The cases $a_0 \equiv a_1 - a_2$ and $a_0 \equiv a_1 \times a_2$ are similar to the last case.

■

Problem 4.

(a) Using the definition of validity, prove that for all assertions A and integer variables j,

$$\text{if} \models A, \text{ then} \models \forall j.A.$$

Answer: We reason as follows:

$$\begin{aligned}
\models A \quad &\Rightarrow \quad \text{for all } \sigma \text{ and } I,\ \sigma \models^I A \\
&\qquad \text{(by the definition of validity)}, \\
&\Rightarrow \quad \text{for all } \sigma, I, \text{ and } n,\ \sigma \models^{I[n/j]} A, \\
&\Rightarrow \quad \text{for all } \sigma \text{ and } I,\ \sigma \models^I \forall j.A \\
&\qquad \text{(by the definition of satisfaction)}, \\
&\Rightarrow \quad \models \forall j.A \\
&\qquad \text{(by the definition of validity)}.
\end{aligned}$$

■

(b) Give an assertion A, state σ, and interpretation I such that

$$\sigma \models^I A,$$

but

$$\sigma \not\models^I \forall j.A.$$

Answer: Here is one example. Let A be the assertion $j = 8$, let I be any interpretation such that $I(j) = 8$, and let σ be arbitrary. Then clearly $\sigma \models^I A$.

However, $\sigma \not\models^I \forall j.A$, because (for example) $\sigma \not\models^{I[3/j]} A$. ■

(c) Conclude that for your A from part (b), $A \Rightarrow \forall j.A$ is not valid.

Answer: If $\models A \Rightarrow \forall j.A$, then for all σ and I,

$$\sigma \models^I A \Rightarrow \forall j.A,$$

by the definition of validity. And then by definition of satisfaction, we have for all σ and I, either (not $\sigma \models^I A$) or ($\sigma \models^I \forall j.A$).

Part (b) contradicts this, so it must be that $A \Rightarrow \forall j.A$ is not valid. ■

Expressing exponentiation in Assn

In this handout we will prove that assertions (**Assn**) are capable of expressing exponentiation. This is not as simple as it might seem; remember, **Assn** does not have exponentiation as a primitive, and while it does have addition and multiplication, it does not have recursive constructs like **while** that make the task easy in real-world programming languages.

Nevertheless, we will be able to "program" exponentiation in **Assn**. We will use a familiar programming strategy: we will break up the problem into sub-problems, solve the sub-problems, and combine those solutions into a solution for the overall problem.

Abbreviations

Each sub-problem will be solved by defining a predicate. We will be defining many predicates, and it will be helpful to have abbreviations for them. Instead of giving a formal definition for our abbreviations, we will illustrate their nuances with the following example:

We define a predicate $DIVIDES(i, j)$ which will hold if and only if i divides j:

$$DIVIDES(i, j) \equiv \exists k. i \times k = j.$$

We use the "(i, j)" in "$DIVIDES(i, j)$" to indicate that $DIVIDES$ is an assertion about the integer variables i and j. When we use $DIVIDES$, we will provide it with "arguments" in place of i and j. For example, when we write "$DIVIDES(z, 9)$", we mean the assertion

$$\exists k. z \times k = 9.$$

When we write "$DIVIDES(j, k)$" we mean the assertion

$$\exists k'. j \times k' = k.$$

Note that we have renamed the original k to k' to avoid capturing the free k.

We will be using some common arithmetic abbreviations, for example, "$x < y$" for "$\neg(y \leq x)$", and "$x > y$" for "$\neg(x \leq y)$".

Sequences

The language of assertions does not provide any of the sophisticated data structures found in modern programming languages. Our first task is to program up one such data structure, sequences of non-negative numbers, so that we can talk about, e.g., $\langle 2,3,4\rangle$ (the sequence of non-negative numbers consisting of 2 followed by 3 followed by 4) or $\langle\rangle$ (the empty sequence of non-negative numbers).

The trick is to observe that a sequence of numbers $\langle n_1, n_2 \ldots, n_m\rangle$ can be uniquely represented as the single number

$$(n_1 \times b^{m-1}) + (n_2 \times b^{m-2}) + \cdots + (n_m \times b^0),$$

provided b is greater than any n_i. That is, we will represent sequences as base b numbers. (For brevity, we omit mention of the "non-negative" condition here and subsequently.)

Offhand, our strategy seems to be circular: we want to express exponentiation by encoding sequences, but our encoding of sequences seems to rely on exponentiation. However, we can avoid this problem by using a prime number for the base b. Because of the special properties of prime numbers, exponentiation of primes is easier to talk about in **Assn** than exponentiation in general. We will be able to express the all of the properties of prime exponentiation that we will need from scratch.

First, we introduce some notation. For any sequence S, we write $\#_p S$ for the encoding of S as a base p number. For example,

$$\begin{aligned}\#_5\langle 4,0,2\rangle &= (4 \times 5^2) + (0 \times 5^1) + (2 \times 5^0),\\ &= 27,\end{aligned}$$

and $\#_{11}\langle\rangle = 0$.

Now, given any sequence $\langle n_1, n_2 \ldots, n_m\rangle$, we can easily express the predicate "$x = \#_p\langle n_1, n_2 \ldots, n_m\rangle$" in **Assn** by:

$$\begin{aligned}&(0 \le n_1) \wedge (n_1 < p)\\ &\quad \wedge \cdots \wedge (0 \le n_m) \wedge (n_m < p)\\ &\quad \wedge (x = (n_1 \times \underbrace{(p \times \cdots \times p)}_{m-1 \text{ times}}) + (n_2 \times \underbrace{(p \times \cdots \times p)}_{m-2 \text{ times}}) + \cdots + n_m).\end{aligned}$$

Note that we have *not* used exponentiation in this definition, nor have we used any special properties of primes. For example, "$x = \#_5\langle 4,0,2\rangle$" is expressed in **Assn** by

$$\begin{aligned}&(0 \le 4) \wedge (4 < 5) \wedge (0 \le 0) \wedge (0 < 5) \wedge (0 \le 2) \wedge (2 < 5)\\ &\quad \wedge (x = (4 \times (5 \times 5)) + (0 \times 5) + 2).\end{aligned}$$

This gives us one way of constructing sequences, but it requires that we have hold of every element of the sequence. We would also like to build up lists by concatenation.

If S_x is the sequence $\langle a_1, \ldots, a_n \rangle$ and S_y is the sequence $\langle b_1, \ldots, b_m \rangle$, then the concatenation of S_x and S_y, written $S_x @ S_y$, is the sequence

$$\langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle.$$

Then if $x = \#_p S_x$ and $y = \#_p S_y$,

$$\begin{aligned} x &= (a_1 \times p^{n-1}) + \cdots + (a_n \times p^0), \\ y &= (b_1 \times p^{m-1}) + \cdots + (b_m \times p^0). \end{aligned}$$

We would like to define an operator $@_p$ so that $x @_p y = \#_p(S_x @ S_y)$. Clearly, we want

$$\begin{aligned} x @_p y = {} & (a_1 \times p^{n-1+m}) + \cdots + (a_n \times p^{0+m}) \\ & + (b_1 \times p^{m-1}) + \cdots + (b_m \times p^0). \end{aligned}$$

That is, $x @_p y$ should be x shifted left by the length of y, plus y: $x \times p^m + y$. Call p^m the *shift-value.* We will use some special properties of primes to calculate the shift-value of a p-encoded sequence.

The predicate $PRIME(p)$ will hold if and only if p is a prime number. Recall that a prime number p is a number greater than 1 whose only positive factors are 1 and p:

$$PRIME(p) \quad \equiv \quad \neg(p \leq 1) \wedge (\forall i.(1 \leq i \wedge DIVIDES(i,p)) \Rightarrow (i = p \vee i = 1)).$$

The predicate $POWP(p, i)$ will hold if and only if p is prime and i is a power of p. Every number has a prime factorization; and the prime factorization of i is of the form $p \times p \times \cdots \times p$ iff i is a power of p. Thus every factor of i is a power of p iff i is a power of p. This leads us to the following definition:

$$\begin{aligned} POWP(p,i) \quad \equiv \quad & PRIME(p) \\ & \wedge (1 \leq i) \\ & \wedge (\forall j.(2 \leq j \wedge DIVIDES(j,i)) \Rightarrow DIVIDES(p,j)). \end{aligned}$$

The predicate $SHIFT(p, y, v)$ will hold if and only if p is prime and v is the shift-value of the p-encoded sequence y:

$$\begin{aligned} SHIFT(p,y,v) \quad \equiv \quad & POWP(p,v) \wedge v > y \\ & \wedge \forall w.(POWP(p,w) \wedge w > y) \Rightarrow v \leq x. \end{aligned}$$

More simply, v is the shift-value of y if it is the least power of p greater than y.

Now that we have $SHIFT$, many predicates involving concatenation are easy to encode. For example:

- The predicate "$w = x@_p y$" can be expressed as
$$\exists v.(SHIFT(p, y, v) \wedge w = x \times v + y).$$
- The predicate "$w = x@_p y@_p z$" can be expressed by
$$\exists q.(w = x@_p q) \wedge (q = y@_p z).$$
- For any sequence S, we can express "$w = \#_p S@_p x$" by
$$\exists q.(w = q@_p x) \wedge (q = \#_p S).$$

All of the predicates involving $@_p$ and $\#_p$ that we use subsequently are similarly defined.

Exponentiation

Our goal is to give an assertion $POW(i, k, n)$ that holds if and only if $i = k^n$. The top-level idea is that $i = k^n$ if and only if there is a sequence of the form
$$\langle 1, k, 0, 2, k^2 \ldots 0, m, k^m, 0\rangle,$$
and n equals the third-to-last element of the sequence, and i equals the second-to-last element of the sequence.

We will develop our assertion POW in a step by step fashion. First, let's call our sequence x, and see how to express that x is of the form above.

We require that x start with 1 followed by k followed by 0:
$$START(p, x, k) \quad \equiv \quad \exists y_1.x = \#_p\langle 1, k, 0\rangle @_p y_1.$$

Any subsequence $\langle a, b, 0\rangle$ of x is either at the end of x, or is immediately followed by a subsequence $\langle a + 1, b \times k, 0\rangle$:
$$\begin{aligned} PATTERN(p, x, k) \quad \equiv \quad & \forall y_2, y_3, a, b. \\ & \quad (x = y_2 @_p(\#_p\langle a, b, 0\rangle)@_p y_3) \Rightarrow \\ & \quad\quad (y_3 = \#_p\langle\rangle \\ & \quad\quad \vee \exists y_4.y_3 = \#_p\langle a + 1, b \times k, 0\rangle @_p y_4). \end{aligned}$$

The third-to-last digit of x must be n, and the second-to-last digit of x must be i:
$$FINISH(p, x, n, i) \quad \equiv \quad \exists y_5.x = y_5 @_p(\#_p\langle n, i, 0\rangle).$$

Putting all the pieces together, we have
$$\begin{aligned} POW(i, k, n) \quad \equiv \quad & \exists p, x.START(p, x, k) \\ & \wedge PATTERN(p, x, k) \\ & \wedge FINISH(p, x, n, i). \end{aligned}$$

Quiz 2

Instructions. This is a **closed book, closed note** exam. There are five problems, on pages 2–10 of this booklet. Write your solutions for all problems on this exam sheet in the spaces provided, including your *name on each sheet.* Don't accidentally skip a page. Ask for further blank sheets if you need them.

Several appendices contain definitions related to the language **IMP**. You have seen all of the material in the appendices before; it is included for your reference only.

GOOD LUCK!

NAME: ______________________________

Problem	Points	Score
1	15	
2	25	
3	25	
4	15	
5	20	
Total	100	

Problem 1 [15 points]. In this problem, we will extend the language **IMP** with a parallel operator, **timeshare**. Informally, $\mathbf{timeshare}(c_0, c_1)$ interleaves the execution of the commands c_0 and c_1. The execution of $\mathbf{timeshare}(c_0, c_1)$ proceeds by first executing c_0 one step, then executing c_1 by one step, then c_0, then c_1, etc. When one of the commands finishes executing, the other is allowed to run to completion.

The **IMP** commands, command contexts, and the "plugging in" operation, (given in Appendix A) are extended as follows:

- $c ::= \ldots \mid \mathbf{timeshare}(c_0, c_1)$,
- $C[\cdot] ::= \ldots \mid \mathbf{timeshare}(C_0[\cdot], C_1[\cdot])$,
- If $C[\cdot] \equiv \mathbf{timeshare}(C_0[\cdot], C_1[\cdot])$, then $C[c] \equiv \mathbf{timeshare}(C_0[c], C_1[c])$.

The one-step semantics $\rightarrow_1$ for **IMP** (given in Appendix B) is extended by the rules:

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \langle c_0', \sigma'\rangle}{\langle \mathbf{timeshare}(c_0, c_1), \sigma\rangle \rightarrow_1 \langle \mathbf{timeshare}(c_1, c_0'), \sigma'\rangle}$$

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \sigma'}{\langle \mathbf{timeshare}(c_0, c_1), \sigma\rangle \rightarrow_1 \langle c_1, \sigma'\rangle}$$

We call the extended language **IMP+timeshare**.

The meaning function for **IMP+timeshare** commands is defined exactly as for **IMP**:

$$[\![c]\!](\sigma) = \sigma' \quad \text{iff} \quad \langle c, \sigma\rangle \rightarrow_1^* \sigma',$$

where the relation $\rightarrow_1^*$ is the transitive closure of the relation $\rightarrow_1$, extended as above.

3(a) [8 points]. Write a definition for $FINISH(p, x, q, n)$, which means that the base-p representation of x ends properly.

The **Assn** $PATTERN(p, x)$ means that if i, j, 0, i', j', and k are consecutive digits in the base-p representation of x, then i, j, i', j', and k are related as indicated by $(\star)$ above:

$$
\begin{aligned}
PATTERN(p, x) \quad \equiv \quad & \forall i, j, k, i', j', x_1, x_2. \\
& (x = (x_1 \ @_p\ (\#_p\langle i, j, 0\rangle) \ @_p\ (\#_p\langle i', j', k\rangle) \ @_p\ x_2) \\
& \quad \Rightarrow (k = 0 \\
& \qquad \wedge\, i' = i + 1 \\
& \qquad \wedge\, A)),
\end{aligned}
$$

where $A \in$ **Assn** expresses the proper relation between j and j'.

3(b) [8 points]. Clearly state in English the proper relation between j and j'.

3(c) [9 points]. Write an $A \in$ **Assn** expressing this relation.

Problem 4 [15 points]. For each of the following forms of **Assn**'s, indicate whether it is valid for every **Assn** A. If it is not valid for every A, exhibit a specific $A \in$ **Assn** which makes the form into a false **Assn**.

4(a) [5 points]. $(\forall i.A) \Rightarrow (\exists i.A)$.

4(b) [5 points]. $(\exists i.A) \Rightarrow (\forall i.A)$.

4(c) [5 points]. $(\forall i.\exists j.A) \Rightarrow (\exists j.\forall i.A)$.

Problem 5 [20 points]. For any program $c \in \mathbf{Com}$ and assertion $A \in \mathbf{Assn}$, we would like to have an **Assn** "$[c]A$" whose meaning is, "if c is executed and it halts, then A holds in the resulting state." That is,

$$\sigma \models^I \text{“}[c]A\text{”} \quad \text{iff} \quad (\text{if } [\![c]\!]\sigma \text{ is defined, then } [\![c]\!]\sigma \models^I A).$$

For example:

- For every state σ and interpretation I,

$$\sigma \models^I \text{“}[X := 0](X = 0)\text{”}.$$

- For every state σ and interpretation I,

$$\sigma \not\models^I \text{“}[X := 0](X = 1)\text{”}.$$

- The assertion "$[X := X + 1](X = Y)$" is true in some states and interpretations, and false in some states and interpretations.

5(a) [5 points]. Describe a state σ_1 such that for all interpretations I,

$$\sigma_1 \models^I \text{“}[X := X + 1](X = Y)\text{”}.$$

5(b) [5 points]. Describe a state σ_2 such that for all interpretations I,

$$\sigma_2 \not\models^I \text{“}[X := X + 1](X = Y)\text{”}.$$

There will always be an **Assn** whose meaning is "$[c]A$". This follows from the expressibility by assertions of the input/output relation of any command.

In particular, for $c \in$ **Com** with $\mathrm{loc}(c) = X$, we know there is an assertion $IO_{c,X}(i,j) \in$ **Assn**, with free variables i and j, which means, "if $\sigma(X) = i$ and $[\![c]\!]\sigma$ is defined, then $([\![c]\!]\sigma)(X) = j$." So the assertion for "$[c]A$" can be written in the form:

$$\forall i,j.(X = i \wedge IO_{c,X}(i,j)) \Rightarrow A',$$

for some $A' \in$ **Assn**.

5(c) [10 points]. Complete the above definition by giving an appropriate $A' \in$ **Assn**.

Appendix A: Syntax of IMP

The arithmetic expressions **Aexp**:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

The boolean expressions **Bexp**:

$$b ::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

The commands **Com**:

$$c ::= \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

The command contexts:

$$\begin{aligned} C[\cdot] ::= & [\cdot] \\ \mid & c \\ \mid & C_0[\cdot]; C_1[\cdot] \\ \mid & \mathbf{if}\ b\ \mathbf{then}\ C_0[\cdot]\ \mathbf{else}\ C_1[\cdot] \\ \mid & \mathbf{while}\ b\ \mathbf{do}\ C'[\cdot] \end{aligned}$$

The "plugging-in" operation for commands and command contexts:

- If $C[\cdot] \equiv [\cdot]$, then $C[c] \equiv c$.
- If $C[\cdot] \equiv c'$, then $C[c] \equiv c'$.
- If $C[\cdot] \equiv C_0[\cdot]; C_1[\cdot]$, then $C[c] \equiv C_0[c]; C_1[c]$.
- If $C[\cdot] \equiv \mathbf{if}\ b\ \mathbf{then}\ C_0[\cdot]\ \mathbf{else}\ C_1[\cdot]$, then

$$C[c] \equiv \mathbf{if}\ b\ \mathbf{then}\ C_0[c]\ \mathbf{else}\ C_1[c].$$

- If $C[\cdot] \equiv \mathbf{while}\ b\ \mathbf{do}\ C'[\cdot]$, then $C[c] \equiv \mathbf{while}\ b\ \mathbf{do}\ C'[c]$.

Appendix B: One-step Semantics of IMP

In this appendix, we use **op** to range over syntactic operator symbols, and *op* to range over corresponding arithmetic or Boolean operations.

One-Step Rules for Arithmetic Expressions

$$\langle X, \sigma \rangle \to_1 \langle \sigma(X), \sigma \rangle$$

$$\frac{\langle a_0, \sigma \rangle \to_1 \langle a_0', \sigma \rangle}{\langle a_0 \ \mathbf{op}\ a_1, \sigma \rangle \to_1 \langle a_0' \ \mathbf{op}\ a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \to_1 \langle a_1', \sigma \rangle}{\langle n \ \mathbf{op}\ a_1, \sigma \rangle \to_1 \langle n \ \mathbf{op}\ a_1', \sigma \rangle}$$

$$\langle n \ \mathbf{op}\ m, \sigma \rangle \to_1 \langle n \ op\ m, \sigma \rangle$$

op	*op*
$+$	the sum function
$-$	the subtraction function
$\times$	the multiplication function

Notice that

$$\langle 5 + 7, \sigma \rangle \to_1 \langle 12, \sigma \rangle$$

is an instance of the rule $\langle n \ \mathbf{op}\ m, \sigma \rangle \to_1 \langle n \ op\ m, \sigma \rangle$, but that

$$\langle 5 + 7, \sigma \rangle \to_1 \langle 5 + 7, \sigma \rangle$$

is *not* derivable at all.

One-Step Rules for Boolean Expressions

$$\frac{\langle a_0, \sigma \rangle \to_1 \langle a_0', \sigma \rangle}{\langle a_0 \ \mathbf{op}\ a_1, \sigma \rangle \to_1 \langle a_0' \ \mathbf{op}\ a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \to_1 \langle a_1', \sigma \rangle}{\langle n \ \mathbf{op}\ a_1, \sigma \rangle \to_1 \langle n \ \mathbf{op}\ a_1', \sigma \rangle}$$

$$\langle n \ \mathbf{op}\ m, \sigma \rangle \to_1 \langle n \ op\ m, \sigma \rangle$$

op	*op*
$=$	the equality predicate
$\leq$	the less than or equal to predicate

We next have the rules for Boolean negation:

$$\frac{\langle b, \sigma\rangle \rightarrow_1 \langle b', \sigma\rangle}{\langle \neg b, \sigma\rangle \rightarrow_1 \langle \neg b', \sigma\rangle}$$

$$\langle \neg\, \mathbf{true}, \sigma\rangle \rightarrow_1 \langle \mathbf{false}, \sigma\rangle$$

$$\langle \neg\, \mathbf{false}, \sigma\rangle \rightarrow_1 \langle \mathbf{true}, \sigma\rangle$$

Finally we have the rules for binary Boolean operators. We use **op** and *op* to range over the symbols and functions in the chart following the rules. We let $t, t_0, t_1, \ldots$ range over the set $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$.

$$\frac{\langle b_0, \sigma\rangle \rightarrow_1 \langle b_0', \sigma\rangle}{\langle b_0 \ \mathbf{op}\ b_1, \sigma\rangle \rightarrow_1 \langle b_0' \ \mathbf{op}\ b_1, \sigma\rangle}$$

$$\frac{\langle b_1, \sigma\rangle \rightarrow_1 \langle b_1', \sigma\rangle}{\langle t_0 \ \mathbf{op}\ b_1, \sigma\rangle \rightarrow_1 \langle t_0 \ \mathbf{op}\ b_1', \sigma\rangle}$$

$$\langle t_0 \ \mathbf{op}\ t_1, \sigma\rangle \rightarrow_1 \langle t_0 \ op\ t_1, \sigma\rangle$$

op	*op*
$\wedge$	the conjunction operation (Boolean AND)
$\vee$	the disjunction operation (Boolean OR)

One-Step Rules for Commands

Atomic Commands:

$$\langle \mathbf{skip}, \sigma\rangle \rightarrow_1 \sigma$$

$$\frac{\langle a, \sigma\rangle \rightarrow_1 \langle a', \sigma\rangle}{\langle X := a, \sigma\rangle \rightarrow_1 \langle X := a', \sigma\rangle}$$

$$\langle X := n, \sigma\rangle \rightarrow_1 \sigma[n/X]$$

Sequencing:

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \langle c_0', \sigma'\rangle}{\langle (c_0; c_1), \sigma\rangle \rightarrow_1 \langle (c_0'; c_1), \sigma'\rangle}$$

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \sigma'}{\langle (c_0; c_1), \sigma\rangle \rightarrow_1 \langle c_1, \sigma'\rangle}$$

Conditionals:

$$\frac{\langle b, \sigma\rangle \rightarrow_1 \langle b', \sigma\rangle}{\langle \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \rightarrow_1 \langle \mathbf{if}\, b'\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle}$$

$$\langle \mathbf{if\ true\ then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \rightarrow_1 \langle c_0, \sigma\rangle$$

$$\langle \mathbf{if\ false\ then}\, c_0\, \mathbf{else}\, c_1, \sigma\rangle \rightarrow_1 \langle c_1, \sigma\rangle$$

While-loops:

$$\langle \mathbf{while}\, b\, \mathbf{do}\, c, \sigma\rangle \rightarrow_1 \langle \mathbf{if}\, b\, \mathbf{then} (c; \mathbf{while}\, b\, \mathbf{do}\, c)\, \mathbf{else\ skip}, \sigma\rangle$$

Appendix C: Rules of Hoare logic for IMP

$$\{A\}\mathbf{skip}\{\mathbf{A}\} \qquad \text{[skip]}$$

$$\{B[a/X]\}X := a\{B\} \qquad \text{[assignment]}$$

$$\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0; c_1\{B\}} \qquad \text{[sequencing]}$$

$$\frac{\{A \wedge b\}c_0\{B\} \quad \{A \wedge \neg b\}c_1\{B\}}{\{A\}\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\{B\}} \qquad \text{[conditional]}$$

$$\frac{\{A \wedge b\}c\{A\}}{\{A\}\mathbf{while}\ b\ \mathbf{do}\ c\{A \wedge \neg b\}} \qquad \text{[loop-invariant]}$$

$$\frac{\models (A \Rightarrow A') \quad \{A'\}c\{B'\} \quad \models (B' \Rightarrow B)}{\{A\}c\{B\}} \qquad \text{[consequence]}$$

Appendix D: The assertion language Assn

$$
\begin{aligned}
A ::= &\ \mathbf{true} \mid \mathbf{false} \\
&\mid a_0 = a_1 \mid a_0 \leq a_1 \\
&\mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid \neg A \mid A_0 \Rightarrow A_1 \\
&\mid \forall i.A \mid \exists i.A
\end{aligned}
$$

Quiz 2 Solution

(This was a closed book, closed note exam. There were five problems, worth 100 points total).

Problem 1 [15 points]. In this problem, we will extend the language **IMP** with a parallel operator, **timeshare**. Informally, **timeshare**(c_0, c_1) interleaves the execution of the commands c_0 and c_1. The execution of **timeshare**(c_0, c_1) proceeds by first executing c_0 one step, then executing c_1 by one step, then c_0, then c_1, etc. When one of the commands finishes executing, the other is allowed to run to completion.

The **IMP** commands, command contexts, and the "plugging in" operation, (given in Appendix A) are extended as follows:

- $c ::= \ldots \mid \textbf{timeshare}(c_0, c_1)$,
- $C[\cdot] ::= \ldots \mid \textbf{timeshare}(C_0[\cdot], C_1[\cdot])$,
- If $C[\cdot] \equiv \textbf{timeshare}(C_0[\cdot], C_1[\cdot])$, then $C[c] \equiv \textbf{timeshare}(C_0[c], C_1[c])$.

The one-step semantics $\rightarrow_1$ for **IMP** (given in Appendix B) is extended by the rules:

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \langle c_0', \sigma'\rangle}{\langle \textbf{timeshare}(c_0, c_1), \sigma\rangle \rightarrow_1 \langle \textbf{timeshare}(c_1, c_0'), \sigma'\rangle}$$

$$\frac{\langle c_0, \sigma\rangle \rightarrow_1 \sigma'}{\langle \textbf{timeshare}(c_0, c_1), \sigma\rangle \rightarrow_1 \langle c_1, \sigma'\rangle}$$

We call the extended language **IMP+timeshare**.

The meaning function for **IMP+timeshare** commands is defined exactly as for **IMP**:

$$[\![c]\!](\sigma) = \sigma' \quad \text{iff} \quad \langle c, \sigma\rangle \rightarrow_1^* \sigma',$$

where the relation $\rightarrow_1^*$ is the transitive closure of the relation $\rightarrow_1$, extended as above.

1(a) [7 points]. Exhibit a simple command, c, of **IMP** (without **timeshare**) such that

$$[\![\mathbf{timeshare}((X := 0; X := 1), c)]\!] \neq [\![\mathbf{timeshare}(X := 1, c)]\!].$$

Answer: One example is $c \equiv (X := 2)$; then for any σ, we have

$$[\![\mathbf{timeshare}((X := 0; X := 1), X := 2)]\!](\sigma)(X) = 1,$$
$$[\![\mathbf{timeshare}(X := 1, X := 2)]\!](\sigma)(X) = 2.$$

■

1(b) [8 points]. Explain why a correct answer to part (a) provides a counterexample to the claim, "the meaning function $[\![\cdot]\!]$ is compositional for the language **IMP+timeshare**."

[Hint: *Any* command c satisfying the inequality of part (a) provides a counterexample; your answer to part (b) should not depend on the particular c you chose for part (a).]

Answer: Let $C[\cdot] \equiv \mathbf{timeshare}([\cdot], c)$, $c_1 \equiv (X := 0; X := 1)$, and $c_2 \equiv X := 1$. Note that $[\![c_1]\!] = [\![c_2]\!]$, but part (a) shows that $[\![C[c_1]]\!] \neq [\![C[c_2]]\!]$, directly contradicting compositionality.

■

Problem 2 [25 points].

2(a) [10 points]. Exhibit a simple program HALVE $\in$ **Com** satisfying the partial correctness assertion

$$\{0 \leq Y \wedge (\exists j.(j = 0 \vee j = 1) \wedge ((2 \times i) + j = Y))\}\text{HALVE}\{i = Y\}.$$

Answer: (I use "$a < a'$" as an abbreviation for $\neg(a' \leq a)$.)

$$\text{HALVE} \equiv Z := Y; \mathbf{while}\ Z < 2 \times Y\ \mathbf{do}\ Y := Y - 1$$

■

Assume that HALVE is a program satisfying the partial correctness assertion from part (a), and let X be a location that is not used in HALVE. Define the program LOGTWO $\in$ **Com** as follows:

$$\text{LOGTWO} \equiv \mathbf{while}\ \neg(Y = 1)\ \mathbf{do}\ (\text{HALVE}; X := X + 1).$$

We claim that the predicate

"$Y \times 2^X = n$ and Y is a power of 2"

is a loop-invariant for LOGTWO.

2(b) [5 points]. To show that the above predicate is a loop-invariant, what partial correctness assertion must be shown to be valid? (The Hoare logic rules for **IMP** are included in Appendix C.)

Answer: Let A be the predicate above. In order to show that A is a loop-invariant, we must show

$$\{A \wedge \neg(Y = 1)\}(\text{HALVE}; X := X + 1)\{A\}.$$

■

2(c) [10 points]. Explain why the partial correctness assertion you gave in part (b) is indeed valid.

Answer: Any power of 2 is greater than or equal to 1. Thus $I \wedge \neg(Y = 1)$ implies $Y \geq 2$, and so Y is of the form $2 \times i$, where i is a power of 2. Then by part (a) we have

$$\{I \wedge \neg(Y = 1)\}\text{HALVE}\{\text{"}Y \times 2^{X+1} = n \text{ and } Y \text{ is a power of 2"}\}.$$

And we have

$$\{\text{"}Y \times 2^{X+1} = n \text{ and } Y \text{ is a power of 2"}\}X := X + 1\{A\}$$

by the rule for assignment. Then using the sequencing rule, we have the desired result.

■

Problem 3 [25 points]. We would like to express the assertion "q is the nth prime" in the language **Assn**. (**Assn** is defined in Appendix D.) We will take an approach similar to the one we used, in class and in a handout, to define "$i = k^n$" in **Assn**.

We have at our disposal the following **Assn**'s:

- $PRIME(p)$, which means that p is a prime number;
- "$x = \#_p\langle i_1, i_2, \ldots, i_m\rangle$", which means that the base-p representation of x is the sequence of digits $i_1, i_2, \ldots, i_m$;
- and "$z = x @_p y$", which means that the base-p representation of z is the concatenation of the base-p representations of x and y.

The top-level idea is that q is the nth prime if and only if there is a sequence of the form

$$\langle 1, p_1, 0, 2, p_2, 0, 3, p_3, 0, \ldots, m, p_m, 0\rangle, \qquad (\star)$$

where $n = m$, $q = p_m$, and p_1, p_2, p_3, p_4, ... are the primes in ascending order:

$$2, 3, 5, 7, \ldots.$$

For example, 11 is the 5th prime, and there is a sequence

$$\langle 1, 2, 0, 2, 3, 0, 3, 5, 0, 4, 7, 0, 5, 11, 0\rangle$$

of the above form.

Using this idea, we will define $NTHPRIME(q, n) \in$ **Assn** which means that q is the nth prime:

$$\begin{aligned} NTHPRIME(q,n) \quad \equiv \quad & \exists p, x.(START(p,x) \\ & \wedge PATTERN(p,x) \\ & \wedge FINISH(p,x,q,n)). \end{aligned}$$

The **Assn** $START(p, x)$ says that the base-p representation of x is a sequence that starts out properly:

$$\begin{aligned} START(p,x) \quad \equiv \quad & \exists x_1, x_2.(x = x_1 \mathbin{@_p} x_2 \\ & \wedge x_1 = \#_p\langle 1, 2, 0\rangle). \end{aligned}$$

Similarly, $FINISH(p, x, q, n)$ will say that x ends properly, and $PATTERN(p, x)$ will say that sub-sequences of x follow an appropriate pattern.

3(a) [8 points]. Write a definition for $FINISH(p, x, q, n)$, which means that the base-p representation of x ends properly.

Answer:

$$\begin{aligned} FINISH(p,x,q,n) \quad \equiv \quad & \exists x_1, x_2.(x = x_1 \mathbin{@_p} x_2 \\ & \wedge x_1 = \#_p\langle n, q, 0\rangle). \end{aligned}$$

■

The **Assn** $PATTERN(p, x)$ means that if i, j, 0, i', j', and k are consecutive digits in the base-p representation of x, then i, j, i', j', and k are related as indicated by $(\star)$ above:

$$\begin{aligned} PATTERN(p,x) \quad \equiv \quad & \forall i, j, k, i', j', x_1, x_2. \\ & (x = (x_1 \mathbin{@_p} (\#_p\langle i, j, 0\rangle) \mathbin{@_p} (\#_p\langle i', j', k\rangle) \mathbin{@_p} x_2) \\ & \quad \Rightarrow (k = 0 \\ & \qquad \wedge i' = i + 1 \\ & \qquad \wedge A)), \end{aligned}$$

where $A \in$ **Assn** expresses the proper relation between j and j'.

3(b) [8 points]. Clearly state in English the proper relation between j and j'.

Answer: The proper relation is:

"j' is the least prime greater than j."

It is not necessary to say that j is prime; it will follow by induction that all the j's are prime. However, it is not harmful, either.

■

3(c) [9 points]. Write an $A \in$ **Assn** expressing this relation.

Answer: (I use "$a < a'$" as an abbreviation for $\neg(a' \leq a)$.)

$$A \equiv PRIME(j') \land (j < j') \land \neg(\exists r.PRIME(r) \land (j < r) \land (r < j')).$$

■

Problem 4 [15 points]. For each of the following forms of **Assn**'s, indicate whether it is valid for every **Assn** A. If it is not valid for every A, exhibit a specific $A \in$ **Assn** which makes the form into a false **Assn**.

4(a) [5 points]. $(\forall i.A) \Rightarrow (\exists i.A)$.

Answer: Valid for all A.

■

4(b) [5 points]. $(\exists i.A) \Rightarrow (\forall i.A)$.

Answer: Not valid for all A. One counterexample is

$$A \equiv (i = 3).$$

■

4(c) [5 points]. $(\forall i.\exists j.A) \Rightarrow (\exists j.\forall i.A)$.

Answer: About half the class missed this one. The form is *not* valid for all A. One counterexample is

$$A \equiv (i = j).$$

■

Problem 5 [20 points]. For any program $c \in$ **Com** and assertion $A \in$ **Assn**, we would like to have an **Assn** "$[c]A$" whose meaning is, "if c is executed and it halts, then A holds in the resulting state." That is,

$$\sigma \models^I \text{"}[c]A\text{"} \quad \text{iff} \quad (\text{if } [\![c]\!]\sigma \text{ is defined, then } [\![c]\!]\sigma \models^I A).$$

For example:

- For every state σ and interpretation I,
$$\sigma \models^I \text{"}[X := 0](X = 0)\text{"}.$$
- For every state σ and interpretation I,
$$\sigma \not\models^I \text{"}[X := 0](X = 1)\text{"}.$$
- The assertion "$[X := X + 1](X = Y)$" is true in some states and interpretations, and false in some states and interpretations.

5(a) [5 points]. Describe a state σ_1 such that for all interpretations I,

$$\sigma_1 \models^I \text{"}[X := X + 1](X = Y)\text{"}.$$

Answer: Any σ_1 satisfying $\sigma_1(Y) = \sigma_1(X) + 1$. For example, a σ_1 where

$$\sigma_1(X) = 3,$$
$$\sigma_1(Y) = 4.$$

■

5(b) [5 points]. Describe a state σ_2 such that for all interpretations I,

$$\sigma_2 \not\models^I \text{"}[X := X + 1](X = Y)\text{"}.$$

Answer: Any σ_2 satisfying $\sigma_2(Y) \neq \sigma_2(X) + 1$. For example, a σ_2 where

$$\sigma_2(X) = 4,$$
$$\sigma_2(Y) = 4.$$

■

There will always be an **Assn** whose meaning is "$[c]A$". This follows from the expressibility by assertions of the input/output relation of any command.

In particular, for $c \in$ **Com** with $\text{loc}(c) = X$, we know there is an assertion $IO_{c,X}(i, j) \in$ **Assn**, with free variables i and j, which means, "if $\sigma(X) = i$ and $[\![c]\!]\sigma$ is defined, then $([\![c]\!]\sigma)(X) = j$." So the assertion for "$[c]A$" can be written in the form:

$$\forall i, j.\big(X = i \wedge IO_{c,X}(i, j)\big) \Rightarrow A',$$

for some $A' \in$ **Assn**.

5(c) [10 points]. Complete the above definition by giving an appropriate $A' \in$ **Assn**.

Answer: Remember that the truth of "$[c]A$" is being judged in a state *before* c is executed. The *IO* predicate allows us to look at the state *after* c is executed, through the variable j. Thus to see if A is true after c executes, we merely test A with the value of j in place of X:

$$A' \equiv A[j/X].$$

■

Appendix A: Syntax of IMP

The arithmetic expressions **Aexp**:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

The boolean expressions **Bexp**:

$$b ::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

The commands **Com**:

$$c ::= \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

The command contexts:

$$\begin{aligned} C[\cdot] ::= & \ [\cdot] \\ \mid & \ c \\ \mid & \ C_0[\cdot]; C_1[\cdot] \\ \mid & \ \mathbf{if}\ b\ \mathbf{then}\ C_0[\cdot]\ \mathbf{else}\ C_1[\cdot] \\ \mid & \ \mathbf{while}\ b\ \mathbf{do}\ C'[\cdot] \end{aligned}$$

The "plugging-in" operation for commands and command contexts:

- If $C[\cdot] \equiv [\cdot]$, then $C[c] \equiv c$.
- If $C[\cdot] \equiv c'$, then $C[c] \equiv c'$.
- If $C[\cdot] \equiv C_0[\cdot]; C_1[\cdot]$, then $C[c] \equiv C_0[c]; C_1[c]$.
- If $C[\cdot] \equiv \mathbf{if}\ b\ \mathbf{then}\ C_0[\cdot]\ \mathbf{else}\ C_1[\cdot]$, then

$$C[c] \equiv \mathbf{if}\ b\ \mathbf{then}\ C_0[c]\ \mathbf{else}\ C_1[c].$$

- If $C[\cdot] \equiv \mathbf{while}\ b\ \mathbf{do}\ C'[\cdot]$, then $C[c] \equiv \mathbf{while}\ b\ \mathbf{do}\ C'[c]$.

Appendix B: One-step Semantics of IMP

In this appendix, we use **op** to range over syntactic operator symbols, and *op* to range over corresponding arithmetic or Boolean operations.

One-Step Rules for Arithmetic Expressions

$$\langle X, \sigma \rangle \rightarrow_1 \langle \sigma(X), \sigma \rangle$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow_1 \langle a_0', \sigma \rangle}{\langle a_0 \ \mathbf{op} \ a_1, \sigma \rangle \rightarrow_1 \langle a_0' \ \mathbf{op} \ a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow_1 \langle a_1', \sigma \rangle}{\langle n \ \mathbf{op} \ a_1, \sigma \rangle \rightarrow_1 \langle n \ \mathbf{op} \ a_1', \sigma \rangle}$$

$$\langle n \ \mathbf{op} \ m, \sigma \rangle \rightarrow_1 \langle n \ op \ m, \sigma \rangle$$

op	*op*
$+$	the sum function
$-$	the subtraction function
$\times$	the multiplication function

Notice that

$$\langle 5 + 7, \sigma \rangle \rightarrow_1 \langle 12, \sigma \rangle$$

is an instance of the rule $\langle n \ \mathbf{op} \ m, \sigma \rangle \rightarrow_1 \langle n \ op \ m, \sigma \rangle$, but that

$$\langle 5 + 7, \sigma \rangle \rightarrow_1 \langle 5 + 7, \sigma \rangle$$

is *not* derivable at all.

One-Step Rules for Boolean Expressions

$$\frac{\langle a_0, \sigma \rangle \rightarrow_1 \langle a_0', \sigma \rangle}{\langle a_0 \ \mathbf{op} \ a_1, \sigma \rangle \rightarrow_1 \langle a_0' \ \mathbf{op} \ a_1, \sigma \rangle}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow_1 \langle a_1', \sigma \rangle}{\langle n \ \mathbf{op} \ a_1, \sigma \rangle \rightarrow_1 \langle n \ \mathbf{op} \ a_1', \sigma \rangle}$$

$$\langle n \ \mathbf{op} \ m, \sigma \rangle \rightarrow_1 \langle n \ op \ m, \sigma \rangle$$

op	*op*
$=$	the equality predicate
$\leq$	the less than or equal to predicate

We next have the rules for Boolean negation:

$$\frac{\langle b, \sigma\rangle \rightarrow_1 \langle b', \sigma\rangle}{\langle \neg b, \sigma\rangle \rightarrow_1 \langle \neg b', \sigma\rangle}$$

$$\langle \neg\, \mathbf{true}, \sigma\rangle \rightarrow_1 \langle \mathbf{false}, \sigma\rangle$$

$$\langle \neg\, \mathbf{false}, \sigma\rangle \rightarrow_1 \langle \mathbf{true}, \sigma\rangle$$

Finally we have the rules for binary Boolean operators. We use **op** and *op* to range over the symbols and functions in the chart following the rules. We let $t, t_0, t_1, \ldots$ range over the set $\mathbf{T} = \{\mathbf{true}, \mathbf{false}\}$.

$$\frac{\langle b_0, \sigma\rangle \rightarrow_1 \langle b_0', \sigma\rangle}{\langle b_0 \ \mathbf{op}\ b_1, \sigma\rangle \rightarrow_1 \langle b_0' \ \mathbf{op}\ b_1, \sigma\rangle}$$

$$\frac{\langle b_1, \sigma\rangle \rightarrow_1 \langle b_1', \sigma\rangle}{\langle t_0 \ \mathbf{op}\ b_1, \sigma\rangle \rightarrow_1 \langle t_0 \ \mathbf{op}\ b_1', \sigma\rangle}$$

$$\langle t_0 \ \mathbf{op}\ t_1, \sigma\rangle \rightarrow_1 \langle t_0 \ op\ t_1, \sigma\rangle$$

op	*op*
$\wedge$	the conjunction operation (Boolean AND)
$\vee$	the disjunction operation (Boolean OR)

One-Step Rules for Commands

Atomic Commands:

$$\langle \mathbf{skip}, \sigma\rangle \rightarrow_1 \sigma$$

$$\frac{\langle a, \sigma\rangle \rightarrow_1 \langle a', \sigma\rangle}{\langle X := a, \sigma\rangle \rightarrow_1 \langle X := a', \sigma\rangle}$$

$$\langle X := n, \sigma\rangle \rightarrow_1 \sigma[n/X]$$

Sequencing:

$$\frac{\langle c_0, \sigma\rangle \to_1 \langle c_0', \sigma'\rangle}{\langle (c_0; c_1), \sigma\rangle \to_1 \langle (c_0'; c_1), \sigma'\rangle}$$

$$\frac{\langle c_0, \sigma\rangle \to_1 \sigma'}{\langle (c_0; c_1), \sigma\rangle \to_1 \langle c_1, \sigma'\rangle}$$

Conditionals:

$$\frac{\langle b, \sigma\rangle \to_1 \langle b', \sigma\rangle}{\langle \textbf{if}\, b\, \textbf{then}\, c_0\, \textbf{else}\, c_1, \sigma\rangle \to_1 \langle \textbf{if}\, b'\, \textbf{then}\, c_0\, \textbf{else}\, c_1, \sigma\rangle}$$

$$\langle \textbf{if true then}\, c_0\, \textbf{else}\, c_1, \sigma\rangle \to_1 \langle c_0, \sigma\rangle$$

$$\langle \textbf{if false then}\, c_0\, \textbf{else}\, c_1, \sigma\rangle \to_1 \langle c_1, \sigma\rangle$$

While-loops:

$$\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to_1 \langle \textbf{if}\, b\, \textbf{then}(c; \textbf{while}\, b\, \textbf{do}\, c)\, \textbf{else}\, \textbf{skip}, \sigma\rangle$$

Appendix C: Rules of Hoare logic for IMP

$$\{A\}\textbf{skip}\{A\} \qquad \text{[skip]}$$

$$\{B[a/X]\}X := a\{B\} \qquad \text{[assignment]}$$

$$\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0; c_1\{B\}} \qquad \text{[sequencing]}$$

$$\frac{\{A \wedge b\}c_0\{B\} \quad \{A \wedge \neg b\}c_1\{B\}}{\{A\}\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1\{B\}} \qquad \text{[conditional]}$$

$$\frac{\{A \wedge b\}c\{A\}}{\{A\}\textbf{while}\ b\ \textbf{do}\ c\{A \wedge \neg b\}} \qquad \text{[loop-invariant]}$$

$$\frac{\models (A \Rightarrow A') \quad \{A'\}c\{B'\} \quad \models (B' \Rightarrow B)}{\{A\}c\{B\}} \qquad \text{[consequence]}$$

Appendix D: The assertion language Assn

$$
\begin{aligned}
A ::= \ & \mathbf{true} \mid \mathbf{false} \\
& \mid a_0 = a_1 \mid a_0 \leq a_1 \\
& \mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid \neg A \mid A_0 \Rightarrow A_1 \\
& \mid \forall i.A \mid \exists i.A
\end{aligned}
$$

Grade Statistics for Quiz 2

Number of quizzes taken: 11

Grade range: 68–100

Mean: 85

Histogram:

Range	Count
0–4:	
5–9:	
10–14:	
15–19:	
20–24:	
25–29:	
30–34:	
35–39:	
40–44:	
45–49:	
50–54:	
55–59:	
60–64:	
65–69:	*
70–74:	*
75–79:	*
80–84:	***
85–89:	*
90–94:	**
95–100:	**

Notes on Expressiveness

We define the set **DynAssn** of "dynamic assertions" by adding, for $c \in \mathbf{Com}$, a new form of assertion, $[c]D$, to the definition of **Assn**:

$$\begin{aligned} D \ ::= \quad & a_1 = a_2 \mid a_1 \leq a_2 \mid \neg D \mid D_1 \wedge D_2 \mid D_1 \vee D_2 \mid D_1 \Rightarrow D_2 \\ & \mid \exists j.D \mid \forall j.D \mid [c]D \end{aligned}$$

Informally, $[c]D$ means "after doing c, the property D will hold." Formally,

$$\sigma \models^I [c]D \quad \text{iff} \quad (\text{if } [\![c]\!]\sigma \text{ is defined, then } [\![c]\!]\sigma \models^I D).$$

The following important properties are immediate consequences of the definitions:

- $[c]D$ is a precondition sufficient to ensure that D holds after executing c:

$$\models \{[c]D\}c\{D\}.$$

- Any precondition D' sufficient to ensure that D holds after executing c is stronger than $[c]D$:

$$\models \{D'\}c\{D\} \quad \text{iff} \quad \models (D' \Rightarrow [c]D).$$

For this reason, any formula equivalent to $[c]D$ is called a *weakest precondition of D under c.*

We illustrate $[c]D$ with a few sample equivalences:

$$\begin{aligned} [\mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2]D \quad & \text{iff} \quad (b \Rightarrow [c_1]D) \wedge (\neg b \Rightarrow [c_2]D), \\ [(c_1; c_2)]D \quad & \text{iff} \quad [c_1]([c_2]D), \\ [X := a]A \quad & \text{iff} \quad A[a/X] \quad (\text{for } A \in \mathbf{Assn}). \end{aligned}$$

(Note that A in the last equivalence above must not be a **DynAssn** containing commands, since we have not defined substitution into such formulas. This is hard to do properly, because a location on the left-hand side of an assignment statement acts like a binding identifier.)

It might seem that adding weakest preconditions to the language of assertions will allow us to express more predicates. However, it can be shown that **Assn** already has the power to express weakest preconditions:

Theorem 1 (Expressiveness). For all $c \in \mathbf{Com}$, $A \in \mathbf{Assn}$, there is a formula $W(c, A) \in \mathbf{Assn}$ such that $W(c, A)$ is a weakest precondition of A under c.

Corollary 1. There is a translation mapping any $D \in$ **DynAssn** into an equivalent $\widehat{D} \in$ **Assn**.

Proof (of corollary).

$$\begin{array}{rcl} \widehat{a_1 = a_2} & \text{is} & a_1 = a_2 \\ \widehat{D_1 \vee D_2} & \text{is} & \widehat{D_1} \wedge \widehat{D_2} \\ \widehat{\exists j.D} & \text{is} & \exists j.\widehat{D} \\ \widehat{[c]D} & \text{is} & W(c, \widehat{D}) \end{array}$$

The other cases are similar. ■

Proof (of Expressiveness Theorem). By induction on c:

$$\begin{aligned} W(\textbf{skip}, A) &::= A, \\ W(X := a, A) &::= A[a/X], \\ W(\textbf{if}\, b\, \textbf{then}\, c_1\, \textbf{else}\, c_2, A) &::= (b \Rightarrow W(c_1, A)) \wedge (\neg b \Rightarrow W(c_2, A)), \\ W((c_1; c_2), A) &::= W(c_1, W(c_2, A)). \end{aligned}$$

That $W(c, A)$ is equivalent to $[c]A$ in each case above follows from the equivalences between **DynAssn**'s that we have already noted.

The remaining case, $W(\textbf{while}\, b\, \textbf{do}\, c, A)$, is more elaborate. We take the following approach: we show that there is an easy way to express a weakest precondition for a command if we can express its input/output relation (and vice-versa); and we define the input/output relation for while loops.

Let c' be an arbitrary command, and for notational convenience, say that $\text{loc}(c') \subseteq \{X_1, X_2\}$. We know that the evaluation of c' depends only on the values of X_1 and X_2. Let $\text{IO}_{c',X_1,X_2}(i_1, i_2, j_1, j_2)$ be a formula with free variables i_1, i_2, j_1, and j_2, which means

$$\mathcal{C}[\![c']\!]\sigma[i_1/X_1, i_2/X_2] = \sigma[j_1/X_1, j_2/X_2].$$

To minimize clutter, we'll omit the subscripts X_1, X_2 below.

If $\text{IO}_{c'}(i_1, i_2, j_1, j_2) \in$ **Assn**, then we can define $W(c', A) \in$ **Assn** to be

$$\forall i_1, i_2, j_1, j_2.(X_1 = i_1 \wedge X_2 = i_2 \wedge \text{IO}_{c'}(i_1, i_2, j_1, j_2)) \Rightarrow A[j_1/X_1, j_2/X_2].$$

Since we have defined weakest preconditions for **skip**, assignment, if-statements, and sequencing above, we have defined the corresponding cases of $\text{IO}_{c'}$.

Conversely, from **Assn**'s which are weakest preconditions for c' we can define $\text{IO}_{c'}(i_1, i_2, j_1, j_2) \in$ **Assn** to be

$$(\, W(c', j_1 = X_1 \wedge j_2 = X_2) \wedge \neg W(c', \textbf{false})\,)[i_1/X_1, i_2/X_2].$$

Note that $W(c', j_1 = X_1 \wedge j_2 = X_2)$ means that *if* c' terminates, it does so with final contents of X_1, X_2 equal to j_1, j_2. The other conjunct, $\neg W(c', \mathbf{false})$ is needed to assert that c' does indeed terminate.

It will also be convenient to have a one-to-one coding of pairs of numbers into positive numbers. One way to do this is to define the one-to-one function mkpair : $(\mathbf{N} \times \mathbf{N}) \rightarrow \mathbf{N}^+$ by

$$\text{mkpair}(n, m) ::= 2^{|n|} \cdot 3^{\text{sg}(n)} \cdot 5^{|m|} \cdot 7^{\text{sg}(n)}$$

where $|n|$ is the absolute value of n, $\text{sg}(n) = 1$ if $n > 0$, and $\text{sg}(n) = 0$ if $n \leq 0$.

It was shown in Handout 26 that "$i = k^n$" can be expressed in **Assn**, so it is easy to see that there is an **Assn** which means "$k = \text{mkpair}(i, j)$."

We can now define $\text{IO}_{\mathbf{while}\, b\, \mathbf{do}\, c}$ in much the same way that we defined the assertion "$i = k^n$". Let $\#_p^{-1}(k)$ denote the base p representation of k. We say that there is a sequence, $\#_p^{-1}(k)$ whose first digit is the code of $\langle i_1, i_2, \rangle$, whose last digit is the code of $\langle j_1, j_2 \rangle$, and every two consecutive digits code pairs in the input-output relation of the body, c, of the **while**. Also, every pair but the final one satisfies the guard, b, of the **while**.

Thus we can describe $\text{IO}_{\mathbf{while}\, b\, \mathbf{do}\, c}(i_1, i_2, j_1, j_2) \in$ **Assn** as follows:

$$\begin{aligned}
\exists k \exists p.\ & \text{“}\#_p^{-1}(k) \text{ starts with digit mkpair}(i_1, i_2)\text{”} \wedge \\
& \text{“}\#_p^{-1}(k) \text{ ends with digit mkpair}(j_1, j_2)\text{”} \wedge \\
& (\forall k_1, k_2, l_1, l_2. \\
& \qquad \text{“mkpair}(k_1, k_2) \text{ and mkpair}(l_1, l_2) \text{ are consecutive digits of } \#_p^{-1}(k)\text{”} \\
& \qquad\qquad \Rightarrow (b[k_1/X_1][k_2/X_2] \wedge \text{IO}_c(k_1, k_1, l_1, l_2))\,) \wedge \\
& \neg b[j_1/X_1][j_2/X_2]
\end{aligned}$$

■

Problem Set 7

Due: 24 November 1993.

Reading assignment. Winskel, Appendix; Handout 30

Problem 1. Prove that [**while** b **do** c]A is a loop-invariant for **while** b **do** c.

Now we extend the language **IMP** to a language $\mathbf{IMP}_F$ with a feature for calling "externally defined" partial functions on **N**.

Let **Funcvar** be a set $\{f, f_1, \ldots, g, g_1, \ldots\}$ whose elements are called *function variables.* For each $f \in$ **Funcvar** there is an associated nonnegative integer called its *arity,* written arity(f). $\mathbf{IMP}_F$ is defined by adding one further case to the grammar of **IMP** commands:

$$c ::= \ \ldots \ \mid X := f(Y_1, \ldots, Y_n)$$

where $f \in$ **Funcvar**, arity$(f) = n$, and $X, Y_1, \ldots, Y_n \in$ **Loc**.

A *interpretation,* ρ, of **Funcvar** is an "arity respecting" map from **Funcvar** to partial functions on **N**. That is, $\rho(f) : \mathbf{N}^n \rightharpoonup \mathbf{N}$ where arity$(f) = n$, for every function variable f.

The evaluation of $\mathbf{IMP}_F$ programs is defined relative to an interpretation, ρ, of **Funcvar**. We make the dependence on ρ explicit by a subscript in evaluation assertions for $\mathbf{Com}_F$. Now all clauses in the natural semantics definition of the evaluation relation for $\mathbf{Com}_F$ are the same as the corresponding ones for **Com**, but we add one further axiom for the new kind of assignment commands:

$$\langle X := f(Y_1, \ldots, Y_n), \sigma\rangle \rightarrow^{\rho} \sigma[m/X],$$

where $m = \rho(f)(\sigma(Y_1), \ldots, \sigma(Y_n))$.

Similarly, we define the meaning of $c \in \mathbf{Com}_F$ relative to ρ:

$$[\![c]\!]^{\rho}(\sigma) = \sigma' \quad \text{iff} \quad \langle c, \sigma\rangle \rightarrow^{\rho} \sigma'.$$

And as in the Appendix of Winskel, we define $\{c\}^{\rho}_{X_1,\ldots,X_n,Y}$, the function computed by c with input locations $X_1, \ldots, X_n$ and output location Y, as follows:

$$\{c\}^{\rho}_{X_1,\ldots,X_n,Y}(m_1, \ldots, m_n) \;=\; ([\![c]\!]^{\rho}\sigma)(Y),$$

where σ is the state $\sigma_0[m_1/X_1, \ldots, m_n/X_n]$. A partial function on **N** is said to be **IMP**-computable *relative to* ρ, iff it equals $\{c\}^{\rho}_{X_1,\ldots,X_n,Y}$ for some $c \in \mathbf{Com}_F$ and $X_1, \ldots, X_n, Y \in$ **Loc**.

Problem 2. An interpretation ρ is said to *expressible* iff $\rho(f)$ is expressible for every $f \in$ **Funcvar**. We will prove

Theorem: If ρ is an expressible interpretation of **Funcvar**, then every **IMP**-computable function relative to ρ is also expressible.

To begin, we observe

Lemma: If ρ is expressible, then there is an assertion IO_c^ρ expressing the input/output relation of any $c \in \mathbf{Com}_F$.

To prove the Lemma, we construct the assertions IO_c^ρ by induction on c exactly as in class and in Handout 30 for **IMP**, with one additional case for $\mathbf{IMP}_F$ commands of the form $X := f(Y_1, \ldots, Y_n)$. To simplify notation, we take $n = 2$ and assume X, Y_1, Y_2 are distinct locations.

2(a) Describe how to construct an **Assn**,

$$\mathrm{IO}^{\rho}_{c_0,X,Y_1,Y_2}(i_1, i_2, i_3, j_1, j_2, j_3),$$

that expresses the input/output relation of the command $c_0 \equiv X := f(Y_1, Y_2)$.

2(b) Taking the above Lemma as proved, complete the proof of the above Theorem.

2(c) For any total function $f : \mathbf{N} \to \mathbf{N}$, define its *iterate*, $f^* : \mathbf{N} \to \mathbf{N}$ as follows:

$$f^*(n) = \begin{cases} 0 & \text{if } n \leq 0, \\ f(f^*(n-1)) & \text{otherwise.} \end{cases}$$

Show that if f is expressible, then f^* is expressible.

Problem 3. Let H^ρ be the set of $\mathbf{IMP}_F$ commands that are "self-halting" under interpretation ρ:

$$H^\rho = \{c \mid \{c\}^\rho(\#(c)) \text{ is defined}\}$$

Show that H^ρ is not decidable relative to ρ, for any interpretation ρ. (That is, the characteristic function of H^ρ is not **IMP**-computable relative to ρ.)

Problem Set 6 Solution

Problem 1 [15 points]. Exercise A.6 from Winskel.

Describe how to transform a command c into one which meets the description "do c for S steps or until c halts (whichever happens first)."

Answer: First we must decide what to call a "step". The purpose of the exercise is to show that we can construct a decider for a set M if we have checkers for M and $\bar{M}$. Thus the important properties of a "step" are:

- for any command c and number S, "do c for S steps or until c halts (whichever happens first)" always halts; and
- if c is a command that halts, then there is some value of S such that c halts in less than S steps.

A convenient notion of step satisfying these criteria is the number of executions of while-loop bodies. (While-loops are the only source of non-termination in the language; any while-free **IMP**-command is guaranteed to halt.) So our transformation will consist of maintaining a counter of the number of steps, and modifying commands so that they execute only if the counter has not passed S.

Let C be a fresh location, which we will use for the step counter. Then let

$$\text{“do } c \text{ for } S \text{ steps or until } c \text{ halts (whichever happens first)”} \stackrel{\text{def}}{=} (C := 1; \hat{c}),$$

where $\hat{c}$ is defined by:

$$\begin{aligned}
\widehat{\mathbf{skip}} &\stackrel{\text{def}}{=} \mathbf{skip}, \\
\widehat{X := a} &\stackrel{\text{def}}{=} \mathbf{if}\, C \leq S \,\mathbf{then}\, X := a \,\mathbf{else\,skip}, \\
\widehat{c_0; c_1} &\stackrel{\text{def}}{=} \widehat{c_0}; \widehat{c_1}, \\
\widehat{\mathbf{if}\, b \,\mathbf{then}\, c_0 \,\mathbf{else}\, c_1} &\stackrel{\text{def}}{=} \mathbf{if}\, b \,\mathbf{then}\, \widehat{c_0} \,\mathbf{else}\, \widehat{c_1}, \\
\widehat{\mathbf{while}\, b \,\mathbf{do}\, c} &\stackrel{\text{def}}{=} \mathbf{while}\, (b \wedge C \leq S)\, \mathbf{do}\, (\widehat{c}; C := C + 1)\,.
\end{aligned}$$

Problem 2 [30 points]. Exercise A.8 from Winskel.

(i) Produce **IMP**-commands Mkpair, Left, Right satisfying

$$\{\text{Mkpair}\}(n, m) = \text{mkpair}(n, m),$$
$$\{\text{Left}\}(n) = \text{left}(n),$$
$$\{\text{Right}\}(n) = \text{right}(n),$$

for all $n, m \in \mathbf{N}$.

Answer: Let A, N, and M be fresh locations. Then Mkpair and Left can be defined as follows:

$$\begin{aligned}
\text{Mkpair} \equiv\; & A := 1; \\
& N := X_1; \\
& M := X_2; \\
& \mathbf{if}\,(N < 0)\ \mathbf{then}\ N := -1 \times N\ \mathbf{else}\ A := A \times 2\,; \\
& \mathbf{while}\,(N > 0)\ \mathbf{do}\,(A := A \times 3; N := N - 1)\,; \\
& \mathbf{if}\,(M < 0)\ \mathbf{then}\ M := -1 \times M\ \mathbf{else}\ A := A \times 5\,; \\
& \mathbf{while}\,(M > 0)\ \mathbf{do}\,(A := A \times 7; M := M - 1)\,; \\
& X_1 := A
\end{aligned}$$

$$\begin{aligned}
\text{Left} \equiv\; & A := -1; \\
& N := 0; \\
& M := 1; \\
& \mathbf{while}\ N = 0\ \mathbf{do} \\
& \quad A := A + 1; \\
& \quad N := X_1; \\
& \quad M := M \times 3; \\
& \quad \mathbf{while}\, M \leq N\, \mathbf{do}\, N := N - M \\
& N := 1; \\
& \mathbf{while}\,(2 \times N) < X_1\ \mathbf{do}\ N := N + 1\,; \\
& \mathbf{if}\ 2 \times N = X_1\ \mathbf{then\ skip\ else}\ A := -1 \times A\,; \\
& X_1 := A
\end{aligned}$$

We can define Right just like Left, except using 5 instead of 2, and 7 instead of 3.

(ii) Let c be a text which is of the form of an **IMP**-command, except that c contains assignment statements of the form "$X :=$ left(Y)." Describe how to construct an authentic **IMP**-command $\hat{c}$ which simulates c up to temporary locations.

Answer: Easy once we have Left from part (i). (We show how to handle "$X :=$ right(Y)" and "$X :=$ mkpair(Y, Z)" as well.)

$$\widehat{\mathbf{skip}} \stackrel{\text{def}}{=} \mathbf{skip},$$

$$
\begin{aligned}
\widehat{X := a} &\stackrel{\text{def}}{=} X := a,\\
\widehat{c_0; c_1} &\stackrel{\text{def}}{=} \widehat{c_0}; \widehat{c_1},\\
\widehat{\mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1} &\stackrel{\text{def}}{=} \mathbf{if}\, b\, \mathbf{then}\, \widehat{c_0}\, \mathbf{else}\, \widehat{c_1},\\
\widehat{\mathbf{while}\, b\, \mathbf{do}\, c} &\stackrel{\text{def}}{=} \mathbf{while}\, b\, \mathbf{do}\, \hat{c},\\
X := \widehat{\text{left}}(Y) &\stackrel{\text{def}}{=} (X_1 := Y; \text{Left}; X := X_1),\\
X := \widehat{\text{right}}(Y) &\stackrel{\text{def}}{=} (X_1 := Y; \text{Right}; X := X_1),\\
X := \widehat{\text{mkpair}}(Y, Z) &\stackrel{\text{def}}{=} (X_1 := Y; X_2 := Z; \text{Mkpair}; X := X_1).
\end{aligned}
$$

In each of the last three clauses, we are assuming that X_1, X_2, and the locations used in Left, Right, and Mkpair are distinct from all the locations used in c.

(iii) Suppose that the definition of **Aexp**, and hence of **IMP**, was modified to allow **Aexp**'s of the form "mkpair(a_1, a_2)", "left(a)", and "right(a)" for a, a_1, and a_2 themselves modified **Aexp**'s. Call the resulting language **IMP**$'$. Explain how to translate every $c' \in \mathbf{Com}'$ into a $c \in \mathbf{Com}$ such that c simulates c'.

Answer: For any **IMP**$'$-command c, we define the **IMP**-command $\bar{c}$ that simulates it by taking each sub-command of c of the form $X := a$ and replacing it with an **IMP**-command that simulates it:

$$
\begin{aligned}
\overline{X := a} &\stackrel{\text{def}}{=} (c; X := a'), \quad \text{where } \langle c, a' \rangle = \hat{a},\\
\overline{\mathbf{skip}} &\stackrel{\text{def}}{=} \mathbf{skip},\\
\overline{c_0; c_1} &\stackrel{\text{def}}{=} \overline{c_0}; \overline{c_1},\\
\overline{\mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1} &\stackrel{\text{def}}{=} (c; \mathbf{if}\, b'\, \mathbf{then}\, \overline{c_0}\, \mathbf{else}\, \overline{c_1}), \quad \text{where } \langle c, b' \rangle = \hat{b},\\
\overline{\mathbf{while}\, b\, \mathbf{do}\, c} &\stackrel{\text{def}}{=} (c'; \mathbf{while}\, b'\, \mathbf{do}\, \bar{c}), \quad \text{where } \langle c', b' \rangle = \hat{b}.
\end{aligned}
$$

For any modified **Aexp** a, $\hat{a}$ is a pair $\langle c, a' \rangle$ of an **IMP**-command and an **Aexp** such that the value of a can be calculated by first executing c, then using the value of a':

$$
\begin{aligned}
\hat{n} &\stackrel{\text{def}}{=} \langle \mathbf{skip}, n \rangle,\\
\hat{X} &\stackrel{\text{def}}{=} \langle \mathbf{skip}, X \rangle,\\
\widehat{a_1\, \mathbf{op}\, a_2} &\stackrel{\text{def}}{=} \langle (c_1; c_2), a_1'\ \mathbf{op}\ a_2' \rangle,\\
&\qquad \text{where } \langle c_1, a_1' \rangle = \hat{a_1} \text{ and } \langle c_2, a_2' \rangle = \hat{a_2},\\
&\qquad \text{and } \mathbf{op} \in \{+, -, \times\},\\
\widehat{\text{left}(a)} &\stackrel{\text{def}}{=} \langle (c; Y := a'; X := \text{left}(Y)), X \rangle,\\
&\qquad \text{where } \langle c, a' \rangle = \hat{a}, \text{ and } X \text{ and } Y \text{ are fresh locations,}
\end{aligned}
$$

$$\widehat{\text{right}(a)} \stackrel{\text{def}}{=} \langle (c; Y := a'; X := \text{right}(Y)), X \rangle,$$
where $\langle c, a' \rangle = \hat{a}$, and X and Y are fresh locations,

$$\widehat{\text{mkpair}(a_1, a_2)} \stackrel{\text{def}}{=} \langle (c; Y := a'_1; Z := a'_2; X := \text{mkpair}(Y, Z)), X \rangle,$$
where $\langle c_1, a'_1 \rangle = \hat{a_1}$, $\langle c_2, a'_2 \rangle = \hat{a_2}$,
and X, Y, and Z are fresh locations.

The definition of "$X := \text{left}(Y)$", "$X := \text{right}(Y)$", and "$X := \text{mkpair}(Y, Z)$" is as for part (ii).

Finally, we define $\hat{b}$ for a modified boolean expression b in a similar manner.

$$\widehat{\textbf{true}} \stackrel{\text{def}}{=} \langle \textbf{skip}, \textbf{true} \rangle,$$

$$\widehat{\textbf{false}} \stackrel{\text{def}}{=} \langle \textbf{skip}, \textbf{false} \rangle,$$

$$\widehat{\neg b} \stackrel{\text{def}}{=} \langle c, \neg b' \rangle, \quad \text{where } \langle c, b' \rangle = \hat{b},$$

$$\widehat{b_1 \,\textbf{op}\, b_2} \stackrel{\text{def}}{=} \langle (c_1; c_2), b'_1 \,\textbf{op}\, b'_2 \rangle,$$
where $\langle c_1, b'_1 \rangle = \hat{b_1}$ and $\langle c_2, b'_2 \rangle = \hat{b_2}$,
and $\textbf{op} \in \{\wedge, \vee\}$,

$$\widehat{a_1 \,\textbf{op}\, a_2} \stackrel{\text{def}}{=} \langle (c_1; c_2), a'_1 \,\textbf{op}\, a'_2 \rangle,$$
where $\langle c_1, a'_1 \rangle = \hat{a_1}$ and $\langle c_2, a'_2 \rangle = \hat{a_2}$,
and $\textbf{op} \in \{=, \leq\}$.

Expressibility, Checkability, and Decidability

These are supplementary notes covering lecture and problem set material not in Winskel, Appendix A.

For any $A \in \mathbf{Assn}$, let $\#A$ be its Gödel number (as defined in the Appendix to Winskel). Conversely, let A_n be the assertion, if any, whose Gödel number is n. It is convenient to have A_n be defined for all $n \in \mathbf{N}$, so if n is not the Gödel number of an assertion by the numbering described in Winskell, define A_n to be the assertion **false**.

Let $\mathbf{Validity} = \{ A \in \mathbf{Assn} \mid \models A \}$.

Theorem 1. **Validity** is not expressible.

Proof: For convenience, in this proof we will write "$A(n)$" for the **Assn** $A[n/i_0]$, where $A \in \mathbf{Assn}$, $n \in \mathbf{N}$, and i_0 is a fixed integer variable.

Suppose to the contrary that there was a $V \in \mathbf{Assn}$ which expressed **Validity**. That is, for all $n \in \mathbf{N}$,

$$\models V(n) \quad \text{iff} \quad \models A_n.$$

Using V we will construct $D \in \mathbf{Assn}$ such that

$$\models D(n) \quad \text{iff} \quad \not\models A_n(n) \tag{$\dagger$}$$

("D" is for "diagonal"). Since $D \in \mathbf{Assn}$, we know that $D = A_{n_0}$ for some $n_0 \geq 0$. So rewriting ($\dagger$), we have for all $n \in \mathbf{N}$

$$\models A_{n_0}(n) \quad \text{iff} \quad \not\models A_n(n).$$

Now let n be n_0 for an immediate contradiction.

To construct D, let $p(n,m) = \#A_n(m)$. We let the reader explain why the function p is computable. Now let D be

$$\exists j.(i_0 = j \wedge (\exists i_0.\text{``}i_0 = p(j,j)\text{''} \wedge \neg V)).$$

We know that $D \in \mathbf{Assn}$ because computability of p implies an **Assn** which means "$i_0 = p(j,j)$." Now

$$\begin{array}{llll} \models D(n) & \text{iff} & \models \neg V(p(n,n)) & \text{(by definition of } D\text{)},\\ & \text{iff} & \not\models A_{p(n,n)} & \text{(by definition of } V\text{)},\\ & \text{iff} & \not\models A_n(n) & \text{(by definition of } p\text{)}. \end{array}$$

■

Theorem 2. A set $C \subseteq N$ is checkable iff $C = f(D)$ for some decidable set $D \subseteq N$ and a total computable $f : \mathbf{N} \to \mathbf{N}$.

Proof: ($\Longleftarrow$). Suppose $d \in \mathbf{Com}$ is a decider for D according to the definition in Winskel, Appendix, i.e., in the notation of Problem Set 8, $\{d\}_{X_1,X_1} = \text{char}_D$, the characteristic function of D. We construct a checker for $f(D)$ by searching through $n = 0, 1, -1, 2, -2, \ldots$ for an n such that the input equals $f(n)$ and d returns 1 on input n. Namely, the following command is a checker for $f(D)$.

```
Y1 := X1;                                  % save the input in Y1
X1 := 0;                                   % X1 = 0 means keep searching
Y2 := 0;                                   % start the search at 0
while X1 = 0 do
   if Y1 ≠ F(Y2)                           % is the input = f(Y2)?
        then skip                          % if not, try another Y2
        else loc(d) := 0;                  % clear loc(d)
             X1 := Y2;                     % if Y2 is in D, then setting
             d;                            %    X1 to 1 will end the search
   if Y2 ≤ 0 then Y2 := 1 - Y2 else Y2 := -Y2   % try Y2 = the next number
```

Here we assume that Y_1, Y_2 are "fresh", namely, not in loc(d), and that F is a function variable whose interpretation is f (cf. Problem Set 8, problem 4).

($\Longrightarrow$) Suppose c is a checker for C. Let

$$D = \{ n \mid c \text{ halts on input left}(n) \text{ in right}(n) \text{ steps} \}.$$

Then clearly $C = \text{left}(D)$, D is decidable (cf. Winskel, Exercise A.6), and left : $\mathbf{N} \to \mathbf{N}$ is easily seen to be total computable. ■

Definition 1. A proof system, $\mathcal{P}$, consists of a countable set of objects d_0, d_1, d_2, ..., called "proofs" and an **IMP**-decidable "is a proof of" relation between proofs, d, and **Assn**'s, A, which we write "$d \vdash_{\mathcal{P}} A$". (Saying the relation is decidable means $\{\text{mkpair}(n, \#A) \mid d_n \vdash_{\mathcal{P}} A\}$ is **IMP**-decidable.) An assertion A is *provable in* $\mathcal{P}$ iff there is a proof of A. **Provable**$_{\mathcal{P}}$ is the set of provable assertions. A proof system is *sound* when every provable assertion is valid.

Corollary 1. For any proof system, **Provable** is a checkable set.

Proof: The set of (Gödel numbers of) the provable assertions equals right(D) where D is the **IMP**-decidable set $\{\text{mkpair}(n, \#A) \mid d_n \vdash_{\mathcal{P}} A\}$. ■

Theorem 3 (Incompleteness). For any proof system,

$$\textbf{Provable} \neq \textbf{Validity}.$$

In particular, if a proof system is sound, then there is valid assertion which is not provable.

Proof: **Provable** is checkable and therefore is expressible. **Validity** is not expressible, so **Provable** $\neq$ **Validity**. In a sound system **Provable** $\subseteq$ **Validity**, so we must have (**Validity** $-$ **Provable**) $\neq \emptyset$. ∎

(In class we proved Incompleteness simply using the fact that **Validity** is not checkable. Noncheckability of **Validity** followed by a many-one reduction ($\leq_m$) of the Halting problem to **Validity**, which we deduced from the checkability of the Halting Problem and expressiveness of **Assn**'s. The proof above does not use checkability of the Halting Problem.)

It is important to realize that it doesn't make sense to ask for a true sentence which can never be proved *somehow*. For *each* sound proof system, $\mathcal{P}$, there is a valid assertion unprovable in $\mathcal{P}$, but for *any* given valid assertion, there is certainly a sound proof system which proves it, namely a proof system which has the given sentence as an axiom. In sorting this out, it may help to consider an analogue: think of "assertion" as "integer," "valid assertion" as "even number," "provable assertions" as "finite *or* cofinite set of numbers." We have that no finite or cofinite set of integers equals the even numbers. A "sound" set of "provable assertions" would correspond to a finite or cofinite set of even numbers–which can only be a finite set of even numbers. So every "provable" set in a sound system is missing some even number; on the other hand, there is no even number which is not a member of *some* finite set of even numbers.

Theorem 2 can be usefully strengthened:

Theorem 4. The following are equivalent for $C \subseteq N$:

(i) C is checkable,

(ii) C is empty or $C = \text{range}(f)$ for some total computable $f : \mathbf{N} \rightarrow \mathbf{N}$,

(iii) $C = g(C')$ for some partial computable $g : \mathbf{N} \rightharpoonup \mathbf{N}$ and checkable set C'.

Proof: By Theorem 2, we have that (ii) implies (i) which in turn implies (iii), so we need only show that (iii) implies (ii). If C is empty we are done, so we may assume there is a number $m_0 \in C$. Let $c' \in \textbf{Com}$ be a checker for C' and $c_g \in \textbf{Com}$ be a command computing g. Let mkquadruple be a coding for

quadruples of integers with decoding functions first, second, third and fourth. Define

$$f(n) = \begin{cases} \text{first}(n) & \text{if } c' \text{ on input second}(n) \text{ halts in third}(n) \text{ steps, and} \\ & c_g \text{ on input second}(n) \text{ halts in fourth}(n) \\ & \text{steps with output first}(n), \\ m_0 & \text{otherwise.} \end{cases}$$

It is easy to see that f is total computable and range$(f) = C$. ∎

Note that Theorem 4, part (iii) implies that, had we relaxed the condition that proofs be decidable to be merely checkable, the provable assertions would still be checkable and Incompleteness would still hold.

Definition 2. For $S_1, S_2 \subseteq \mathbf{N}$, we say S_1 is *many-one reducible* to S_2, in symbols $S_1 \leq_m S_2$, iff there is a total computable $f : \mathbf{N} \to \mathbf{N}$ such that for all $n \in \mathbf{N}$

$$n \in S_1 \text{ iff } f(n) \in S_2.$$

Such a function, f, is called a many-one *reduction* from S_1 to S_2.

Lemma 1. The following are equivalent:

1. $S_1 \leq_m S_2$
2. $S_1 = f^{-1}(S_2)$ for some total computable function $f : \mathbf{N} \to \mathbf{N}$,
3. $\text{char}_{S_1} = \text{char}_{S_2} \circ f$ for some total computable $f : \mathbf{N} \to \mathbf{N}$.
4. $\overline{S_1} \leq_m \overline{S_2}$

Proof: From the definition of $\leq_m$. ∎

Lemma 2. Many-one reducibility is transitive.

Proof: Suppose f is a many-one reduction from S_1 to S_2, and g is a many-one reduction from S_2 to S_3. Then $g \circ f$ is a many-one reduction from S_1 to S_3. ∎

Definition 3. A property, P, of sets *inherits downward* under $\leq_m$ iff

$$[S_1 \leq_m S_2 \text{ and } P(S_2)] \text{ implies } P(S_1)$$

Lemma 3. Expressibility, checkability, and decidability all inherit downward under $\leq_m$.

Proof: Suppose f is many-one reduction from S_1 to S_2, and c is a decider (respectively, a checker) for S_2. Then $X_1 := F(X_1)$; c is a decider (resp., checker) for S_1 where F is a function variable denoting f. So decidability (resp. checkability) inherit down.

Suppose $A \in \mathbf{Assn}$ means "$i_0 \in S_2$." Then $\exists i_1$."$i_1 = f(i_0)$" $\wedge A[i_1/i_0]$ means "$i_0 \in S_1$," so expressibility inherits down. ■

Remark. It follows immediately that *non*expressibility, *un*checkability, and *un*decidability inherit *upward* under $\leq_m$.

A property of sets is a *nontrivial property of checkable sets* iff there is some checkable set which has the property and also some checkable set which does not have the property. For any property, P, of sets let

$$\mathbf{Com}_P = \{c \in \mathbf{Com} \mid c \text{ is a checker for some set } S \text{ with property } P\}$$

Example: The Zero-Halting Problem is $\mathbf{Com}_P$ where P is the "contains zero" property, namely, $P(S)$ iff $0 \in S$.

Theorem 5 (Rice). Let P be any nontrivial property of checkable sets such that the empty set does not have property P. Then $S \leq_m \mathbf{Com}_P$ for every checkable set S.

Proof: Let c_S be a checker for S, *i.e.*, $S = \text{domain}(\{c_S\}_{X_1,X_1})$. Since P is non-trivial, there is some non-empty, checkable set S_0 with property P; let c_{S_0} be a checker for S_0.

Now let Y be a fresh location ($Y \notin \text{loc}(c_S) \cup \text{loc}(c_{S_0})$). For $n \in \mathbf{N}$ define $d_n \in \mathbf{Com}$ to be

$$Y := X_1;\ X_1 := n;\ c_S;\ \text{loc}(c_{S_0}) := 0;\ X_1 := Y;\ c_{S_0}\,.$$

If $n \notin S$, then d_n diverges in all states, and so is a checker for the empty set. On the other hand, if $n \in S$, then d_n acts like c_{S_0} in all states such that all locations except X_1 have contents equal to zero, and so d_n is a checker for S_0. Thus,

$$n \in S \text{ iff } d_n \in \mathbf{Com}_P.$$

But the function $f(n) = \#d_n$ is another composition of the mkpair function with itself with some substituted constants, and so is clearly total and computable. Thus, $S \leq_m \mathbf{Com}_P$. ■

Corollary 2. If P is a nontrivial property of checkable sets, then $\mathbf{Com}_P$ is not decidable.

Proof: If not $P(\emptyset)$, Rice's Theorem immediately yields

$$H \leq_m \mathbf{Com}_P$$

because the Halting Problem, H, is checkable. Since H is undecidable and undecidablity inherits up $\leq_m$, $\mathbf{Com}_P$ is undecidable also.

If $P(\emptyset)$, then let P' be $\neg P$. By the previous case, $\mathbf{Com}_{P'}$ is undecidable. But note that $\mathbf{Com}_{P'} = \mathbf{Com} \cap \overline{\mathbf{Com}_P}$, so $\mathbf{Com}_P$ must be undecidable (cf. the next exercise).

■

Exercise. (a) Show that if $S_1 = D \cap S_2$ for sets $S_1, D, S_2 \subseteq \mathbf{N}$ such that $S_1 \neq \emptyset, D$ is decidable, and $\overline{S_2} \neq \emptyset$, then $S_1 \leq_m S_2$.

(b) Describe S_1, S_2 such that $S_1 \leq_m S_2$ but $S_1 \neq D \cap S_2$ for any set D.

Some Examples: The valid equations between **Aexp**'s is an interesting example of a decidable set. The Halting Problem, H, is a checkable set which is not decidable. $\overline{H}$ is an expressible set which is not checkable. Let $H^{(1)} = H$ and $H^{(n+1)}$ be the Halting Problem relative to $H^{(n)}$; then for $n > 1$, the set $H^{(n)}$ is expressible but neither it nor its complement is checkable. **Validity** is not expressible.

6.044J/18.423J: Computability, Programming, and Logic — Handout 34
Massachusetts Institute of Technology — 1 December 1993

Problem Set 8

Due: 3 December 1993.

[This problem set builds on material introduced in Problem Set 7 (Handout 31).]

Let f_0 be some designated function variable of arity 1. For any set S, let ρ_S be the interpretation that maps f_0 to the characteristic function of S, and maps any other function variable f_i to the "always 0" function of arity(f_i). The purpose of ρ_S is to add the ability to decide membership in S to the language $\mathbf{IMP}_F$.

For sets S_1, S_2 we say that S_1 is *Turing-reducible to* S_2, or S_1 is *decidable relative to* S_2, iff S_1 is **IMP**-computable relative to ρ_{S_2}. We write $S_1 \leq_T S_2$ iff S_1 is Turing-reducible to S_2.

Problem 1.

(a) Show that for any S, we have $S \leq_T \bar{S}$.

(b) Show that $\leq_T$ is transitive. That is, for any sets S_1, S_2, and S_3, show that $S_1 \leq_T S_2$ and $S_2 \leq_T S_3$ implies $S_1 \leq_T S_3$.

[Hint: consider "inlining" of code.]

Problem 2. For any set S, let S' be the set H^{ρ_S}.

(a) Show that $S \leq_m S'$.

[Hint: for any $n \in \mathbf{N}$, consider the $\mathbf{IMP}_F$ program c_n defined as follows:

$$c_n \stackrel{\text{def}}{=} (X_1 := n;\ X_1 := f_0(X_1);\ \mathbf{if}\ X_1 = 1\ \mathbf{then\,skip\,else}\ \Omega),$$

where Ω is some diverging ("infinite loop") program.]

(b) We write $S_1 <_T S_2$ iff $S_1 \leq_T S_2$ and $S_2 \not\leq_T S_1$. Show that $S <_T S'$ for all S.

[Hint: consider problem 3 from Problem Set 7.]

Diophantine sets

A set S of numbers is *diophantine* iff for some polynomial $p(x_0, \ldots, x_k)$ with integer coefficients,

$$S = \{ n \mid p(n, m_1, \ldots, m_k) = 0 \text{ for some } m_1, \ldots, m_k \}. \qquad (1)$$

We say a set S' of numbers is NNR (for non-negative range) iff for some polynomial $p'(x_0, \ldots, x_k)$ with integer coefficients,

$$S' = \{ n \mid n \geq 0 \text{ and } n = p'(m_0, \ldots, m_k) \text{ for some } m_0, \ldots, m_k \}. \qquad (2)$$

We show that for any set S of non-negative numbers, S is diophantine iff S is NNR.

First, suppose S is diophantine. Then there is some polynomial $p(x_0, \ldots, x_k)$ satisfying the equation (1) above.

Define the polynomial $p'(x_0, \ldots, x_k)$ as follows:

$$p'(x_0, \ldots, x_k) \stackrel{\text{def}}{=} x_0 - (1 + x_0^2) \times (p(x_0, \ldots, x_k))^2.$$

[Technically, the right-hand side of the above equation is not a polynomial; it is an arithmetic expression (**Aexp**). However, any arithmetic expression can be transformed into an equivalent polynomial expression, as indicated in the Appendix of Winskel. We mean in the above definition that p' is a polynomial equivalent to the right-hand side.]

This defines for us a set S' satisfying (2). We show $S = S'$.

- If $n \in S$, then by (1) there are $m_1, \ldots, m_k$ such that

  $$p(n, m_1, \ldots, m_k) = 0.$$

  But then

  $$\begin{aligned} p'(n, m_1, \ldots, m_k) &= n - (1 + n^2) \times 0 \\ &= n. \end{aligned}$$

  Since S contained only non-negative numbers, $n \geq 0$ and so $n \in S'$.

- Now suppose $n \in S'$. Then $n \geq 0$ and for some $m_0, \ldots, m_k$ we have

  $$\begin{aligned} n &= p'(m_0, \ldots, m_k) \\ &= m_0 - (1 + m_0^2) \times (p(m_0, \ldots, m_k))^2. \end{aligned}$$

  Since every square is non-negative, we have

  $$(p(m_0, \ldots, m_k))^2 \geq 0.$$

  Consider the two following possibilities:

1. If $p(m_0, \ldots, m_k) = 0$, then $n = m_0$. And then
$$p(n, m_1, \ldots, m_k) = 0,$$
so clearly $n \in S$.
2. Else $(p(m_0, \ldots, m_k))^2 > 0$. Then since $(1 + m_0^2) > m_0$, we have
$$(1 + m_0^2) \times (p(m_0, \ldots, m_k))^2 > m_0,$$
and thus
$$\begin{aligned} n &= m_0 - (1 + m_0^2) \times ((p(m_0, \ldots, m_k))^2) \\ &< 0. \end{aligned}$$
Since we assumed $n \geq 0$, clearly this case cannot hold.

Thus we have shown $n \in S$ iff $n \in S'$, i.e. $S = S'$.

Now suppose S' is NNR. Then there is some polynomial $p'(x_0, \ldots, x_k)$ satisfying the equation (2) above.

By definition, $n \in S'$ iff for some $m_0, \ldots, m_k$, we have both
$$\begin{aligned} n &= p'(m_0, \ldots, m_k), \\ n &\geq 0. \end{aligned}$$

We will use the following number theoretic fact:

Theorem 1 (Four squares). Any non-negative number can be expressed as the sum of four squares. That is,
$$n \geq 0 \quad \text{iff} \quad n = a^2 + b^2 + c^2 + d^2 \text{ for some } a, b, c, d.$$

By the Four Squares Theorem, $n \in S'$ iff for some $m_0, \ldots, m_k$, a, b, c, d, we have
$$\begin{aligned} n - p'(m_0, \ldots, m_k) &= 0, \\ n - (a^2 + b^2 + c^2 + d^2) &= 0. \end{aligned}$$

Furthermore, for any integers n_1 and n_2, we know
$$(n_1 = 0 \text{ and } n_2 = 0) \quad \text{iff} \quad n_1^2 + n_2^2 = 0.$$

So define the polynomial
$$\begin{aligned} p(z, x_0, \ldots, x_k, y_1, y_2, y_3, y_4) \stackrel{\text{def}}{=} & (z - p'(x_0, \ldots, x_k))^2 \\ & + (z - (y_1^2 + y_2^2 + y_3^2 + y_4^2))^2. \end{aligned}$$

This defines for us a set S by equation (1). And $n \in S'$ iff for some $m_0, \ldots, m_k$, a, b, c, d,
$$p(n, m_0, \ldots, m_k, a, b, c, d) = 0.$$

Thus $n \in S'$ iff $n \in S$, i.e. $S = S'$.

Problem Set 7 Solution

Problem 1 [10 points]. Prove that $[\textbf{while}\, b\, \textbf{do}\, c]A$ is a loop-invariant for $\textbf{while}\, b\, \textbf{do}\, c$.

Answer: We want to show that

$$\models \{[\textbf{while}\, b\, \textbf{do}\, c]A \wedge b\}c\{[\textbf{while}\, b\, \textbf{do}\, c]A\},$$

or in other words, we want to show that for all states σ and interpretations I, if $\sigma \models^I [\textbf{while}\, b\, \textbf{do}\, c]A \wedge b$ and $[\![c]\!]\sigma$ is defined, then $[\![c]\!]\sigma \models^I [\textbf{while}\, b\, \textbf{do}\, c]A$.

So suppose that

$$\sigma \models^I [\textbf{while}\, b\, \textbf{do}\, c]A \wedge b, \tag{1}$$

and $[\![c]\!]\sigma$ is defined. Then for some state σ'',

$$[\![c]\!]\sigma = \sigma''. \tag{2}$$

We must show that $\sigma'' \models^I [\textbf{while}\, b\, \textbf{do}\, c]A$. By definition of weakest precondition, this is equivalent to showing that if $[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\sigma''$ is defined, then $[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\sigma'' \models^I A$. So assume

$$[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\sigma'' = \sigma'. \tag{3}$$

By (1) we know $\langle b, \sigma\rangle \to \textbf{true}$, and then from (2), (3), and the following rule for while-loops,

$$\frac{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma''\rangle \to \sigma', \quad \langle c, \sigma\rangle \to \sigma'', \quad \langle b, \sigma\rangle \to \textbf{true}}{\langle \textbf{while}\, b\, \textbf{do}\, c, \sigma\rangle \to \sigma'},$$

we know $[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\sigma = \sigma'$. Then by (1) and the definition of weakest precondition, we have $[\![\textbf{while}\, b\, \textbf{do}\, c]\!]\sigma \models^I A$. Thus $\sigma' \models^I A$ as desired. ■

Now we extend the language **IMP** to a language $\textbf{IMP}_F$ with a feature for calling "externally defined" partial functions on $\mathbf{N}$.

Let **Funcvar** be a set $\{f, f_1, \ldots, g, g_1, \ldots\}$ whose elements are called *function variables.* For each $f \in \textbf{Funcvar}$ there is an associated nonegative integer called its *arity*, written $\text{arity}(f)$. $\textbf{IMP}_F$ is defined by adding one further case to the grammar of **IMP** commands:

$$c ::= \ \ldots \ \mid X := f(Y_1, \ldots, Y_n)$$

where $f \in \textbf{Funcvar}$, $\text{arity}(f) = n$, and $X, Y_1, \ldots, Y_n \in \textbf{Loc}$.

A *interpretation*, ρ, of **Funcvar** is an "arity respecting" map from **Funcvar** to partial functions on **N**. That is, $\rho(f) : \mathbf{N}^n \rightharpoonup \mathbf{N}$ where arity$(f) = n$, for every function variable f.

The evaluation of $\mathbf{IMP}_F$ programs is defined relative to an interpretation, ρ, of **Funcvar**. We make the dependence on ρ explicit by a subscript in evaluation assertions for $\mathbf{Com}_F$. Now all clauses in the natural semantics definition of the evaluation relation for $\mathbf{Com}_F$ are the same as the corresponding ones for **Com**, but we add one further axiom for the new kind of assignment commands:

$$\langle X := f(Y_1, \ldots, Y_n), \sigma\rangle \rightarrow^{\rho} \sigma[m/X],$$

where $m = \rho(f)(\sigma(Y_1), \ldots, \sigma(Y_n))$.

Similarly, we define the meaning of $c \in \mathbf{Com}_F$ relative to ρ:

$$[\![c]\!]^{\rho}(\sigma) = \sigma' \quad \text{iff} \quad \langle c, \sigma\rangle \rightarrow^{\rho} \sigma'.$$

And as in the Appendix of Winskel, we define $\{c\}^{\rho}_{X_1,\ldots,X_n,Y}$, the function computed by c with input locations $X_1, \ldots, X_n$ and output location Y, as follows:

$$\{c\}^{\rho}_{X_1,\ldots,X_n,Y}(m_1, \ldots, m_n) \quad = \quad ([\![c]\!]^{\rho}\sigma)(Y),$$

where σ is the state $\sigma_0[m_1/X_1, \ldots, m_n/X_n]$. A partial function on **N** is said to be **IMP**-computable *relative to* ρ, iff it equals $\{c\}^{\rho}_{X_1,\ldots,X_n,Y}$ for some $c \in \mathbf{Com}_F$ and $X_1, \ldots, X_n, Y \in \mathbf{Loc}$.

Problem 2 [15 points]. An interpretation ρ is said to *expressible* iff $\rho(f)$ is expressible for every $f \in \mathbf{Funcvar}$. We will prove

Theorem: If ρ is an expressible interpretation of **Funcvar**, then every **IMP**-computable function relative to ρ is also expressible.

To begin, we observe

Lemma: If ρ is expressible, then there is an assertion IO^{ρ}_c expressing the input/output relation of any $c \in \mathbf{Com}_F$.

To prove the Lemma, we construct the assertions IO^{ρ}_c by induction on c exactly as in class and in Handout 30 for **IMP**, with one additional case for $\mathbf{IMP}_F$ commands of the form $X := f(Y_1, \ldots, Y_n)$. To simplify notation, we take $n = 2$ and assume X, Y_1, Y_2 are distinct locations.

2(a) Describe how to construct an **Assn**,

$$\mathrm{IO}^{\rho}_{c_0,X,Y_1,Y_2}(i_1, i_2, i_3, j_1, j_2, j_3),$$

that expresses the input/output relation of the command $c_0 \equiv X := f(Y_1, Y_2)$.

Answer: ρ is expressible, so $\rho(f)$ is expressible. Therefore there is an assertion that means "$i = \rho(f)(i', i'')$", for any integer variables i, i', i''. We will use this to ensure that the output value of X is the value of $\rho(f)$ given the input values of Y_1 and Y_2. Also, the values of Y_1 and Y_2 should not be changed by the command. Thus our **Assn** is:

$$\text{“}j_1 = \rho(f)(i_2, i_3)\text{”} \wedge j_2 = i_2 \wedge j_3 = i_3.$$

■

2(b) Taking the above Lemma as proved, complete the proof of the above Theorem.

Answer: Suppose f is an **IMP**-computable function relative to ρ, and ρ is expressible. Then we must show that f is expressible. That is, we seek an **Assn** A such that

$$f(n) = m \quad \text{iff} \quad \models A[n/i, m/j].$$

Since f is **IMP**-computable relative to ρ, by definition there must be some **IMP**$_F$-command c such that

$$\begin{aligned} f(n) = m \quad & \text{iff} \quad \{c\}_{X_1,Y}(n) = m \\ & \text{iff} \quad \mathrm{IO}^{\rho}_{c,X_1,Y}(n, m) \\ & \text{iff} \quad \mathrm{IO}^{\rho}_{c,X_1,Y}[n/i, m/j]. \end{aligned}$$

So just let $A \stackrel{\text{def}}{=} \mathrm{IO}^{\rho}_{c,X_1,Y}$, which is an **Assn** by the Lemma above. ■

2(c) For any total function $f : \mathbf{N} \to \mathbf{N}$, define its *iterate*, $f^* : \mathbf{N} \to \mathbf{N}$ as follows:

$$f^*(n) = \begin{cases} 0 & \text{if } n \leq 0, \\ f(f^*(n-1)) & \text{otherwise.} \end{cases}$$

Show that if f is expressible, then f^* is expressible.

Answer: Let ρ be an interpretation such that $\rho(f_0) = f$, for some designated function symbol f_0 of arity 1, and $\rho(f_i)$ is the "always 0" function of arity(f_i) if f_i is any other function symbol. Since f is expressible, and the "always 0" function is expressible, we know ρ is expressible.

Then the following **IMP**$_F$-program computes f^*:

$$\begin{aligned} c \stackrel{\text{def}}{=} \quad & Y := 0; \\ & \textbf{while } 1 \leq X_1 \textbf{ do} \\ & \quad Y := f_0(Y); \\ & \quad X_1 := X_1 - 1 \end{aligned}$$

That is, $\{c\}_{X_1,Y}(n,m)$ iff $f^*(n) = m$. Therefore by the Theorem proved in 2(b), f^* is expressible. ■

Problem 3 [10 points]. Let H^ρ be the set of $\mathbf{IMP}_F$ commands that are "self-halting" under interpretation ρ:

$$H^\rho = \{c \mid \{c\}^\rho(\#(c)) \text{ is defined}\}$$

Show that H^ρ is not decidable relative to ρ, for any interpretation ρ. (That is, the characteristic function of H^ρ is not **IMP**-computable relative to ρ.)

Answer: There seemed to be some common misunderstandings about this problem, which I will try to clear up here.

Some people assumed that results which were proved about decidability also held for decidability relative to ρ. For example, it was proved in the book that for any set S of numbers,

$$S \text{ is decidable} \quad \text{iff} \quad S \text{ and } \bar{S} \text{ are checkable.}$$

It does *not* immediately follow that

$$S \text{ is decidable relative to } \rho \quad \text{iff} \quad S \text{ and } \bar{S} \text{ are checkable relative to } \rho.$$

The result does indeed hold, but it is not something that you can assume; it must be proved. In fact, the proof for this property, and many other simple properties, follow those in the book for decidability and checkability. I wouldn't expect you to write out the proofs again, but you should indicate that you are aware of the issue.

One incorrect proof attempt was as follows: since $H \subseteq H^\rho$, and H is not decidable, it must be that H^ρ is not decidable.

There are two things wrong with this attempt. First, note that if $S_1 \subseteq S_2$ and S_1 is not decidable, we do not necessarily have S_2 undecidable. For example, take $S_1 = H$ and $S_2 = \mathbf{N}$.

Second, we are not asking you to show H^ρ undecidable; we are asking you to show H^ρ undecidable *relative to* ρ. These are two very different things; the ρ can add a tremendous amount of power to the language. For example, there is an interpretation ρ_0 such that H is decidable relative to ρ_0! Just take $\rho_0(f_0)$ to be the characteristic function of H.

Why doesn't this argument show that we can find a ρ_1 such that H^{ρ_1} is decidable relative to ρ_1? Let's try to define ρ_1 in the same way as we defined ρ_0. We want to define ρ_1 so that $\rho_1(f_0)$ is the characteristic function of H^{ρ_1}. In other words,

$$\rho_1(f_0)(n) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if for some } c,\ n = \#(c) \text{ and } \{c\}^{\rho_1}(\#(c)) \text{ is defined,} \\ 0 & \text{otherwise.} \end{cases}$$

However, it is not immediately clear whether this is a good definition. The problem is that we are trying to define ρ_1 *in terms of itself.* The value of ρ_1 on f_0 is to be determined by the evaluation of a command which might use the value of $\rho_1(f_0)$ in the course of its execution. Sometimes such recursive "definitions" have a solution, and sometimes not. For an example of a non-sensical recursive "definition," just consider

$$x \stackrel{\text{def}}{=} x + 1.$$

In fact, the "definition" of ρ_1 is non-sensical. This can be seen as a corollary of the result we asked you to prove, and which we will prove here. The proof just follows the proof in the book for the undecidability of H.

It is easy to see that if a set S is decidable relative to ρ, then its complement $\bar{S}$ is decidable relative to ρ, and in fact $\bar{S}$ must be checkable relative to ρ (the proofs are just as in the book).

So if H^ρ is decidable relative to ρ, then $\bar{H}^\rho$ is checkable relative to ρ. We reach a contradiction by showing that $\bar{H}^\rho$ is not checkable relative to ρ.

By definition of $\bar{H}^\rho$, we have for all $c \in \mathbf{Com}_F$,

$$c \in \bar{H}^\rho \quad \text{iff} \quad \{c\}^\rho(\#(c)) \text{ is not defined.} \tag{4}$$

Now suppose $\bar{H}^\rho$ is checkable relative to ρ. Then by definition of checkable relative to ρ, there is some $c_0 \in \mathbf{Com}_F$ such that for all $c \in \mathbf{Com}_F$,

$$c \in \bar{H}^\rho \quad \text{iff} \quad \{c_0\}^\rho(\#(c)) \text{ is defined.} \tag{5}$$

Combining (4) and (5), we have for all $c \in \mathbf{Com}_F$,

$$\{c\}^\rho(\#(c)) \text{ is not defined} \quad \text{iff} \quad \{c_0\}^\rho(\#(c)) \text{ is defined.}$$

But $c_0 \in \mathbf{Com}_F$, so we must have

$$\{c_0\}^\rho(\#(c_0)) \text{ is not defined} \quad \text{iff} \quad \{c_0\}^\rho(\#(c_0)) \text{ is defined,}$$

a contradiction. Thus $\bar{H}^\rho$ is not checkable relative to ρ, so H^ρ is not decidable relative to ρ. ■

Problem Set 8 Solution

[This problem set builds on material introduced in Problem Set 7 (Handout 31).]

Let f_0 be some designated function variable of arity 1. For any set S, let ρ_S be the interpretation that maps f_0 to the characteristic function of S, and maps any other function variable f_i to the "always 0" function of arity(f_i). The purpose of ρ_S is to add the ability to decide membership in S to the language $\mathbf{IMP}_F$.

For sets S_1, S_2 we say that S_1 is *Turing-reducible to* S_2, or S_1 is *decidable relative to* S_2, iff char_{S_1} is **IMP**-computable relative to ρ_{S_2}. We write $S_1 \leq_T S_2$ iff S_1 is Turing-reducible to S_2.

Problem 1.

(a) Show that for any S, we have $S \leq_T \bar{S}$.

Answer: We must show that char_S is **IMP**-computable relative to $\rho_{\bar{S}}$. That is, we must find a $c_0 \in \mathbf{Com}_F$ such that

$$\text{char}_S = \{c_0\}^{\rho_{\bar{S}}}_{X,Y}. \tag{1}$$

Define

$$c_0 \stackrel{\text{def}}{=} \begin{array}{l} Y := f_0(X); \\ \mathbf{if}(Y = 0)\,\mathbf{then}\,Y := 1\,\mathbf{else}\,Y := 0. \end{array}$$

Then for all $n \in \mathbf{N}$,

$$\{c_0\}^{\rho_{\bar{S}}}_{X,Y}(n) = \begin{cases} 0 & \text{if } n \notin S, \\ 1 & \text{if } n \in S, \end{cases}$$

so c_0 satisfies (1) as desired. ∎

(b) Show that $\leq_T$ is transitive. That is, for any sets S_1, S_2, and S_3, show that $S_1 \leq_T S_2$ and $S_2 \leq_T S_3$ implies $S_1 \leq_T S_3$.

[Hint: consider "inlining" of code.]

Answer: Suppose $S_1 \leq_T S_2$ and $S_2 \leq_T S_3$. Then there are commands $c_{12}, c_{23} \in \mathbf{Com}_F$ such that c_{12} computes char_{S_1} relative to ρ_{S_2}, and c_{23} computes char_{S_2} relative to ρ_{S_3}. To show that $S_1 \leq_T S_3$, we must find $c_{13} \in \mathbf{Com}_F$ that computes char_{S_1} relative to ρ_{S_3}.

The command c_{12} is almost what we want: it computes char_{S_1}, but relative to ρ_{S_2} instead of ρ_{S_3}. That is, it computes char_{S_1} by asking questions

about membership in S_2. We want to eliminate questions about membership in S_2, and allow questions about membership in S_3 instead. We can do this by "inlining" the program c_{23} (that computes membership in S_2 relative to ρ_{S_3}) for any S_2 query in c_{12}.

In more detail, suppose w.l.o.g. that the locations of c_{12} and the locations of c_{23} are disjoint. And assume that the input and output locations of c_{23} are X_1 and Y_1 respectively. Then the command c_{13}, obtained from c_{12} by replacing each sub-command of the form

$$X := f_0(Y)$$

by the command

$$X_1 := X; c_{23}; Y := Y_1,$$

computes char_{S_1} relative to ρ_{S_3}. ∎

Problem 2. For any set S, let S' be the set H^{ρ_S}.

(a) Show that $S \leq_m S'$.

[Hint: for any $n \in \mathbf{N}$, consider the $\mathbf{IMP}_F$ program c_n defined as follows:

$$c_n \stackrel{\text{def}}{=} (X_1 := n;\ X_1 := f_0(X_1);\ \mathbf{if}\, X_1 = 1\, \mathbf{then\, skip\, else}\, \Omega),$$

where Ω is some diverging ("infinite loop") program.]

Answer: It is not hard to see that

$$\begin{aligned} n \in S \quad & \text{iff} \quad c_n \text{ halts under } \rightarrow_{\rho_S} \\ & \text{iff} \quad c_n \in S'. \end{aligned}$$

And the function f mapping n to c_n is total computable (this is a straightforward application of the Gödel numbering of commands as in the Appendix of Winskel).

Thus we have found a total computable function f such that $n \in S$ iff $f(n) \in S'$, and therefore $S \leq_m S'$. ∎

(b) We write $S_1 <_T S_2$ iff $S_1 \leq_T S_2$ and $S_2 \not\leq_T S_1$. Show that $S <_T S'$ for all S.

[Hint: consider problem 3 from Problem Set 7.]

Answer: First, $S \leq_m S'$ implies $S \leq_T S'$. (Write a program that computes the function f from part (a) and then uses $\rho_{S'}(f_0)$ to test membership in S' on the output of f.)

And $S' \not\leq_T S$ iff $H^{\rho_S} \not\leq_T S$
iff $\text{char}_{H^{\rho_S}}$ is not **IMP**-computable relative to ρ_S.

Problem 3 from Problem Set 7 proved that for *any* ρ, char_{H^ρ} is not **IMP**-computable relative to ρ, so we have $S' \not\leq_T S$. ∎

Quiz 4 and Solutions from 1991

(This was a **closed book, closed notes** exam. There were four (4) problems.)

Problem 1 [17 points]. For any sets S, T, let $S - T$ be the set of all elements of S which are not elements of T.

1(a) [10 points]. Show that if S and T are *decidable* subsets of $\mathbf{N}$, then $S - T$ is decidable.

Solution A: There are two reasonable solutions to this problem. The first solution uses the fact that the set of decidable languages is closed under intersection and complement.

We observe that $S - T = S \cap \overline{T}$. In class we were told that the set of decidable languages is closed under intersection, so if we can show that S and $\overline{T}$ are decidable then we are done. By the premise we have S decidable. It is then a simple task to show that if T is decidable then so is $\overline{T}$. Specifically, if d is a decider for T, then

$$d; \mathbf{if}\, X_1 = 0\, \mathbf{then}\, X_1 := 1\, \mathbf{else}\, X_1 := 0$$

is clearly an **IMP** command which decides $\overline{T}$.

Solution B: Let d_1 be a decider for S and d_2 be a decider for T, and let T_0 be a fresh location. Then the following **IMP** command is a decider for $S - T$.

$T_0 := X_1$
d_1;
if $X_1 = 0$
  then $T_0 := 0$
  else $X_1 := T_0$;
    d_2;
    $T_0 := 0$;
    if $X_1 = 0$ **then** $X_1 := 1$ **else** $X_1 := 0$

We then verbally argue that this does the job....

1(b) [7 points]. Give an example of two *checkable* (r.e.) subsets S and T of $\mathbf{N}$ such that $S - T$ is not checkable. No explanation is required.

Solution: Let $S = \mathbf{N}$, and T be any set which is checkable, but not decidable, for example $T = H$. Then $\mathbf{N} - T$ is simply $\overline{T} = \overline{H}$ which is not checkable.

Problem 2 [20 points]. Let

$$\textbf{Divergent} \stackrel{\text{def}}{=} \{n \geq 0 \mid [\![com_n]\!]s(k) = \bot \text{ for all } k\}.$$

Prove that **Divergent** is not checkable. (Here com_n is the command with Gödel number n, and $s(k)$ is the state with k in location X_1 and 0 in all other locations.)

Hint: $[\![com_n]\!]s(n) = \bot$ iff $[\![X := n; com_n]\!]s(k) = \bot$ for all k.

Solution: Assume $c \in$ **Com** is a checker for **Divergent**. Then the command:

$$\text{“}X_1 := \text{mkseq}(\text{mkassign}(\text{mkloc}(1), \text{mknum}(n)), n)\text{”}; c$$

will, by the hint, check NOT-SELF-HALT—a contradiction (since the NOT-SELF-SET is not checkable). Thus our assumption that **Divergent** was checkable is incorrect, and so **Divergent** is not checkable.

Problem 3 [28 points]. An assertion A is **satisfiable** iff there exists a state σ and interpretation I such that $\sigma \models^I A$.

3(a) [10 points]. Let **SAT** $\stackrel{\text{def}}{=} \{\#(A) \mid A \text{ is satisfiable}\}$. (Here $\#(A) \in \mathbf{N}$ is the Gödel number of $A \in$ **Assn**. The assertion A need not necessarily be closed.) Prove that **SAT** is not checkable.

Hint: Consider closed, location-free assertions.

The valid, closed, location-free assertions are not checkable. But the subset S, of Gödel-numbers of assertions which are Gödel numbers of closed, location-free assertions is a decidable set. As a closed, location-free assertion is valid iff it is satisfiable then **SAT** $\cap S =$ the valid, closed, location-free assertions.

Suppose **SAT** were decidable. As S is obviously decidable, and decidable sets are closed under intersection, then **SAT** $\cap S$ would be decidable—which it is not. Thus **SAT** is not decidable.

Let **BSAT** $= \{\#(b) \mid b \in$ **Bexp** and b is satisfiable$\}$.

3(b) [10 points]. Explain why **BSAT** *is* checkable.

Hint: We don't expect you to write an **IMP** program. Just describe in high-level terms an algorithm to decide whether or not a **Bexp** is satisfiable.

Solution: We can evaluate B:

Just check all possible assignments of numbers to $X_1, X_2, \ldots, X_k \in \text{loc}(b)$ (there must be a finite number of locations, wlog assume these are them). If B is satisfiable, one will yield true and the algorithm stops. Note: it is possible to canonically order the assignments of the locations.

When checking a particular assignment, plug the values of the X_i's into b. (This is easy to do). Then replace all **Aexp**'s in b by their value (as the **Aexp**'s no longer have locations or integer variables, this is easy to do). We can then replace all the equalities and inequalities by their appropriate truth values (again this is easy as they are of the form $n_1 \leq n_2$ or $n_1 = n_2$. Finally, we simply have a boolean combination of **true** and **false** which is also easy to evaluate. If the result is **true** then B was satisfiable, if it was **false**, we go on to the next assignment. This process of checking an assignment will always terminate, and give the right answer.

3(c) [8 points]. Prove that **BSAT** is *not* decidable.

Hint: Hilbert's 10th Problem.

Solution: Suppose **BSAT** were decidable. Let d be a decider for **BSET**. As satisfiability of polynomial equalities is a special case of **BSAT**, the following command would be a decider for Hilbert's 10th Problem.

$$\text{“}X_1 := \text{mkeq}(X_1, \text{mknum}(0))\text{”}; d$$

Since there can be no decider for Hilbert's 10th Problem, we have a contradiction, and so **BSAT** is not decidable.

Problem 4 [35 points]. We consider axioms for symmetries (rigid, "in place" transformations) of an equilateral triangle. For example, given the triangle with vertices labeled as in Figure 1, we can apply

Transformation "r": $r$otate 120° clockwise, obtaining the triangle in Figure 2;

Transformation "f": $f$lip about the vertical axis, obtaining the triangle in Figure 3;

Transformation "l": $l$eave unchanged, obtaining the triangle in Figure 3 again.

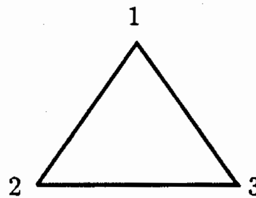


Figure 1: The original triangle.

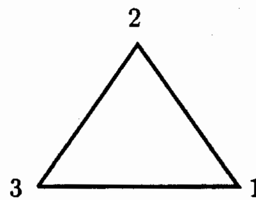


Figure 2: The original triangle after performing transformation r, a 120° clockwise rotation.

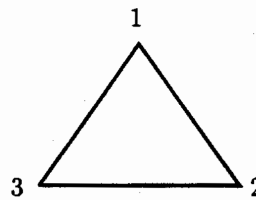


Figure 3: The original triangle after performing transformation f, a flip about the vertical axis.

Let W be the set of finite sequences (of length at least 1) of the letters r, f, and l. Elements of W are called *words* over the alphabet $\{r, f, l\}$.

By interpreting concatenation of letters as composition of permutations, we can associate with any word, w, a permutation, $[\![w]\!]$, of $\{1,2,3\}$ indicating the movement of vertices of a triangle. So the basic permutations defined by r and f are:

$$\begin{array}{rclrclrcl} [\![r]\!](1) & = & 2, & [\![r]\!](2) & = & 3, & [\![r]\!](3) & = & 1. \\ [\![f]\!](1) & = & 1, & [\![f]\!](2) & = & 3, & [\![f]\!](3) & = & 2. \end{array}$$

Note that $[\![l]\!]$ is simply the identity function. Inductively, let $[\![aw]\!] = [\![a]\!] \circ [\![w]\!]$ for $a \in \{f, r, l\}$. For example, $[\![rfrl]\!](x) = r(f(r(l(x))))$, so

$$[\![rfrl]\!](1) = 1, \quad [\![rfrl]\!](2) = 3, \quad [\![rfrl]\!](3) = 2.$$

Define "truth", $\models$, of a "triangle" word equation as follows:

$$\models (w_1 = w_2) \quad \text{iff} \quad [\![w_1]\!] = [\![w_2]\!].$$

For example, $\models rfrl = f$.

4(a) [3 points]. Exhibit w_1 and w_2, such that

$$\not\models w_1 w_2 = w_2 w_1.$$

Solution: For example, $w_1 = r$ and $w_2 = f$.

4(b) [7 points]. The "standard" rules for equality are reflexivity, symmetry, transitivity, and congruence. State these rules for the case of word equations.

Solution:

$$\vdash w = w \qquad \text{(reflexivity)}$$

$$\frac{\vdash w_1 = w_2}{w_2 = w_1} \qquad \text{(symmetry)}$$

$$\frac{\vdash w_1 = w_2 \quad \vdash w_2 = w_3}{\vdash w_1 = w_3} \qquad \text{(transitivity)}$$

$$\frac{\vdash w_1 = w_2}{\vdash aw_1 = aw_2} \qquad \text{(left congruence)}$$

where $a \in \{r, f, l\}$

$$\frac{\vdash w_1 = w_2}{\vdash w_1 a = w_2 a} \qquad \text{(right congruence)}$$

where $a \in \{r, f, l\}$

4(c) [10 points]. Show that if a sound axiom system is strong enough to prove any word equal to one of the six "canonical" forms below, then we can obtain a sound and complete axiom system by adding the standard rules for equality. The six canonical forms are:

$$l, \quad r, \quad rr, \quad f, \quad rf, \quad rrf.$$

Solution: Suppose $\models w_1 = w_2$, *i.e.*, $[\![w_1]\!] = [\![w_2]\!]$. By the presumption, there are canonical forms $\hat{w}_1$ and $\hat{w}_2$ such that $\vdash w_1 = \hat{w}_1$, and $\vdash w_2 = \hat{w}_2$. Since the system is sound, $\models w_i = \hat{w}_i$. $[\![\hat{w}_1]\!] = [\![w_1]\!] = [\![w_2]\!] = [\![\hat{w}_2]\!]$.

In addition, each of the six "canonical" forms have different meanings. So, we have

$$\vdash w_1 = \hat{w}_1 \quad \text{and} \quad \vdash w_2 = \hat{w}_2$$

and by symmetry and transitivity, we conclude $\vdash w_1 = w_2$.

4(d) [15 points]. Consider the complete proof system for triangle word equations whose rules are just the standard rules for equality plus the axioms:

$$rrr = ff = ll = l \qquad \text{(unit)}$$

$$rl = lr = r, \quad fl = lf = f \qquad \text{(identity)}$$

$$fr = rrf \qquad \text{(swap)}$$

Briefly explain why this proof system is sound and complete. *Hint*: Show how to prove that an arbitrary word equals one of the six canonical forms of problem 4(c).

Solution Assuming the result of Problem 4(c), it should be clear that all we need to do is show, using the above axoims and the rules for equality, that it is possible to prove that any triangle world is equal to one of the six canonical forms.

The following process will halt and reduce an arbitrary word to a canonical form.

Step 1 Erase all l's (unless $w \equiv l$, in which case we are done). This follows from the identity axioms, plus the rules for equality.

Step 2 Move all f's to the right. This is possible from the rules for equality and the swap axiom. So now we have a word containing only r's and f's with all f's on the right.

Step 3 Replace rrr (if it occurs) by l. This is possible from the rules for equality and the unit axiom.

Step 4 Erase all l's (unless $w \equiv l$, in which case we are done)

Step 5 If there is still rrr left in w go to Step 2.

Step 6 Replace ff (if it occurs) by l. This is possible from the rules for equality and the unit axiom.

Step 7 Erase all l's (unless $w \equiv l$, in which case we are done)

Step 8 If there is still ff left in w go to Step 5.

Clearly this will halt, as we are always making the word shorter.

Clearly if it halts it will have f's to the right of r's, if there are any l's left then the result is l. If there are r's left, they all must be adjacent on the left, thus by Steps 3 to 5, there can be no more than two r's. If there are f's left they all must be adjacent on the right, thus by steps 6–8, there can be no more than one f. This paragraph now precisely characterizes the canonical forms.

Quiz 3 and Solutions from 1992

[This is a copy of Quiz 3 from 1992, with handwritten solutions.]

Problem 1.

1(a) Exhibit a while-loop invariant suitable for a Hoare logic proof of

$$\{X = i \wedge Y = j\}\,\textbf{while}\,X \neq Y\ \textbf{do}\ Y := Y + 1\,\{i \leq j\}.$$

The simplest we found was $(j \leq Y \wedge X = i)$. Clearly, this is implied by the precondition, and the conjunction of it and $\neg(X \neq Y)$ implies the postcondition.

1(b) Give a formal proof in Hoare logic of

$$\{X = 0\}\textbf{while true do}\,(Y := Y + 1;\ X := X - 1)\{X = 3\}.$$

Hint: false is a loop invariant.

Since $true[X-1/X] \equiv true$ and $true[Y+1/Y] \equiv true$

we have

$\{true\}\ Y := Y+1\ \{true\} \qquad \{true\}\ X := X-1\ \{true\}$ — by assignment

---

$\{true\}\ Y := Y+1;\ X := X+1\ \{true\}$ — by sequencing

$\{true\}$ while true do $Y := Y+1;\ X := X-1\ \{true \wedge \neg true\}$ — by consequence $(\{true \wedge true\} \Rightarrow \{true\})$ and while-loop

---

$\{true\}$ while . . . . . $\{false\}$ — by consequence

$\{X=0\}$ while - - - - $\{X=3\}$ — by consequence $(\{X=0\} \Rightarrow \{true\} \wedge \{false\} \Rightarrow \{X=3\})$

Problem 2. Show that **Bexp** is expressive for **while**-free commands. That is, if $c \in$ **Com** contains no **while**-loops and $b \in$ **Bexp**, then the weakest precondition $\{\textbf{true}\}c\{b\}$ is equivalent to some $b' \in$ **Bexp**.

Hint: Induction on c.

As shown in the expressiveness proof:

$$w(\text{skip}, b) \equiv b$$

$$w(X := a, b) \equiv b[a/x]$$ (which is a Bexp since Bexps are closed under substitution)

Then, by induction,

if $w(c_0, b)$ is representable as a Bexp for all b, and $w(c_1, b')$ similarly, then by induction

$$w(c_0; c_1, b) \equiv w(c_0, w(c_2, b)) \equiv w(c_0, b')$$

for some Bexp b', which is then representable as a Bexp.

...

and similarly

$$w(\text{if } b' \text{ then } c_0 \text{ else } c_1, b)$$

$$\equiv (b' \Rightarrow w(c_0, b)) \wedge (\neg b' \Rightarrow w(c_1, b))$$

$$\equiv (\neg b' \vee w(c_0, b)) \wedge (b' \vee w(c_1, b))$$

is representable as a Bexp by induction.

Problem 3. Sketch how to transform any **Assn** into an equivalent **Assn** of the form

$$(Q_1 i_1) \ldots (Q_n i_n)[a = 0]$$

where each Q_i is either $\forall$ or $\exists$ and $a \in$ **Aexpv**.

0) Convert $A \Rightarrow B$ to $\neg A \vee B$ everywhere.

1) Push all $\neg$ inward as far as possible using

$\neg(A \wedge B) \equiv \neg A \vee \neg B$ $\quad$ $\neg(A \vee B) \equiv \neg A \wedge \neg B$

$\neg \exists j. A \equiv \forall j. \neg A$ $\quad$ $\neg \forall j. A \equiv \exists j. \neg A$

$\neg\neg A = A$

2) Replace $\neg(a_0 \leq a_1)$ with $(a_1 \leq a_0 \wedge \neg(a_1 = a_0))$

3) Replace $\neg(a_0 = a_1)$ with

$$\exists i, j, k, l.\ (a_0 - a_1)^2 = i^2 + j^2 + k^2 + l^2 + 1$$

4) Replace $a_0 \leq a_1$ with

$$\exists i, j, k, l.\ a_0 + i^2 + j^2 + k^2 + l^2 = a_1$$

5) Replace $a_0 = a_1$ with $a_0 - a_1 = 0$

6) Move all quantifiers out, renaming variables as necessary

e.g. $A \wedge \exists j. B \equiv \exists k. A \wedge B[k/j]$

where k is fresh.

7) Eliminate $\wedge$ and $\vee$ with

$a_0 = 0 \wedge a_1 = 0 \equiv a_0^2 + a_1^2 = 0$

$a_0 = 0 \vee a_1 = 0 \equiv a_0 \times a_1 = 0$

Problem 3. Sketch how to transform any **Assn** into an equivalent **Assn** of the form

$$(Q_1 i_1)\dots(Q_n i_n)[a = 0]$$

where each Q_i is either $\forall$ or $\exists$ and $a \in$ **Aexpv**.

0) Convert $A \Rightarrow B$ to $\neg A \vee B$

1) Push all $\neg$ inward as far as possible using

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$
$$\neg \exists j.A \equiv \forall j.\neg A$$
$$\neg \forall j.A \equiv \exists j.\neg A$$
$$\neg\neg A \equiv A$$

2) replace $\neg(a_0 \le a_1)$ with $(a_1 \le a_0 \wedge \neg(a_1 = a_0))$

3) replace all $\neg(a_0 = a_1)$ with

$$\exists i,j,k,l.\ (a_0-a_1)\times(a_0-a_1) + i^2 - j^2 - k^2 - l^2 - 1 = 0$$

4) replace all $a_0 \le a_1$ with

$$\exists i,j,k,l.\ a_1 - a_0 - i^2 - j^2 - k^2 - l^2 = 0$$

5) replace all $a_0 = a_1$ with $a_0 - a_1 = 0$

6) Move all quantifiers out, renaming variables as necessary.

e.g. $A \wedge \exists j.B \equiv \exists k.A \wedge B[k/j]$ where k is fresh.

7) Eliminate $\wedge$ and $\vee$ with

$$a_0 = 0 \wedge a_1 = 0 \equiv a_0^2 + a_1^2 = 0$$
$$a_0 = 0 \vee a_1 = 0 \equiv a_0 \times a_1 = 0$$

Problem 4. The grammar for *exponential constant expressions*, **Ecexp**, is

$$e ::= 1 \mid e + e \mid e \times e \mid e^e$$

Meaning is defined as for arithmetic expressions, with superscript denoting exponentiation, *e.g.*, the meaning of $(1+(1+1))^{(1+1)\times((1+1)+(1+1))}$ is $3^{2\cdot 4}$, namely, 6,561. An expression **Ecexp** is said to be a *canonical form* if it is a sum of 1's (parenthesized to the left).

4(a) Write down a simple set of sound axioms for equations between **Ecexp**'s, which, together with the usual inference rules for equations (reflexivity, symmetry, transitivity, congruence), allow one to prove that any $e \in$ **Ecexp** equals a canonical form canon(e). Also, briefly explain how to use your axioms to prove that $e =$ canon(e).

The simplest set of rules we found was

1) $e_0^{e_1+1} = e_0 \times e_0^{e_1}$

2) $e^1 = e$

3) $e_0 \times (e_1+1) = (e_0 \times e_1) + e_0$

4) $e \times 1 = e$

5) $e_0 + (e_1+1) = (e_0+e_1)+1$

~~converting expressions that match the left~~

As an algorithm: starting at the innermost, uppermost expressions, apply applicable rules from left-hand-side to right-hand-side. This will eventually convert any Ecexp to canonical form.

4(b) Prove that your axiom system is complete.

Given that the rules are sound and sufficient to prove $\vdash e = \text{canon}(e)$ for any e, we have:

if $\models e_0 = e_1$ then ~~[illegible]~~ $\vdash e_0 = \text{canon}(e_0)$ and (*)
$\vdash e_1 = \text{canon}(e_1)$ (**)

by soundness, then, $\models e_0 = \text{canon}(e_0)$ and
$\models e_1 = \text{canon}(e_1)$,

so by ~~[illegible]~~ symmetry & transitivity
$\models \text{canon}(e_0) = \text{canon}(e_1)$

Since canonical forms can only have the same value if they are identical (simple proof by induction on the length), canon(e_0) and canon(e_1) are the same expression. Then, by symmetry from (**)

$\vdash \text{canon}(e_1) = e_1$ (***)

and by transitivity from (*) and (***)

$\vdash e_0 = e_1$

A Hoare Logic

Axiom for **skip**:

$$\{A\}\,\mathbf{skip}\,\{A\}$$

Axiom for assignments:

$$\{B[a/X]\}X := a\{B\}$$

Rule for sequencing:

$$\frac{\{A\}c_0\{C\},\ \{C\}c_1\{B\}}{\{A\}(c_0;c_1)\{B\}}$$

Rule for conditionals:

$$\frac{\{A \wedge b\}c_0\{B\},\ \{A \wedge \neg b\}c_1\{B\}}{\{A\}\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\{B\}}$$

Rule for while loops:

$$\frac{\{A \wedge b\}c\{A\}}{\{A\}\mathbf{while}\ b\ \mathbf{do}\ c\{A \wedge \neg b\}}$$

Rule of consequence:

$$\frac{\{A'\}c\{B'\}}{\{A\}c\{B\}} \qquad (\text{providing} \models (A \Rightarrow A') \wedge (B' \Rightarrow B)\ )$$

Quiz 4 and Solutions from 1992

(This was a **closed book, closed notes** exam. There were five (5) problems of roughly equal weight.)

Problem 1 [20 points]. **For each of the following problems indicate which properties it has ($\surd$) or does not have ($\times$):**

	is decidable	is checkable	has checkable complement	is expressible
the self-halting problem $H = \{\#c \mid c \in \mathbf{Com}$ and c halts on input $\#c\}$	$\times$	$\surd$	$\times$	$\surd$
the valid equations between **Aexp**'s (arithmetic expressions)	$\surd$	$\surd$	$\surd$	$\surd$
the valid **Bexp**'s (Boolean expressions)	$\times$	$\times$	$\surd$	$\surd$
the unsatisfiable **Assn**'s (first-order arithmetic formulas)	$\times$	$\times$	$\times$	$\times$
$\{\#c \mid c \in \mathbf{Com}$ and c halts on some input$\}$	$\times$	$\surd$	$\times$	$\surd$
$\{\#c \mid c \in \mathbf{Com}$ and c is **while-free** $\}$	$\surd$	$\surd$	$\surd$	$\surd$
$\{\#c \mid c \in \mathbf{Com}$ and c halts on input 0 in at most 1000 steps$\}$	$\surd$	$\surd$	$\surd$	$\surd$
$\{n \in \mathbf{N} \mid n \geq$ some element of the self-halting problem $H\}$	$\surd$	$\surd$	$\surd$	$\surd$

Explanation:

Recall that a set is decidable iff it is both checkable and co-checkable, and if a set is checkable or co-checkable then it is expressible.

- The self-halting problem was shown in class to be checkable but not decidable.
- The appendix to Winskel gives a proof that the valid equations between **Aexp**'s are a decidable set.
- If we could check the validity of **Bexp**'s, then we could check the validity of inequations of the form $\neg(a = 0)$ where $a \in \mathbf{Aexp}$, which was shown to

be uncheckable in the appendix. However, if a **Bexp**, b, is not valid, then there is a substitution of numbers for its locations such that b is false. So, a checker for non-validity of **Bexp**'s only needs to run through all possible substitutions of numbers for variables in loc(b) until it finds one for which b evaluates to **false**.

- If the unsatisfiable **Assn**'s were expressible then so would the satisfiable ones be. A closed **Assn** is true iff it is satisfiable, so this would allow us to construct an expression for Truth, which has been shown to be inexpressible.
- This can be checked by gradually checking each machine against each input to see if that machine halts in n steps, gradually increasing n. If this set were decidable, though, then we could construct a decider for the "halts on 0" problem, H_0.
- It requires only a bounded, syntactic check to detect the presence of **while**-statements in a command, so this is decidable.
- This can be decided by running c on 0 for 1000 steps, and checking if c is done. This procedure will always terminate.
- Since all codes of machines are nonnegative, all elements of H are as well. Thus, there is a least $\#c \in H$. This set can be decided by comparing any given n against $\#c$.

Problem 2 [20 points]. **Outline a proof that if a set D and its complement $\overline{D}$ are checkable, then D is decidable.**

If D and $\overline{D}$ are checkable, then they both have checkers, c_D and $c_{\overline{D}}$. We can construct a decider for D by generating a program which saves its input, picks with some positive value S, and then alternately runs c_D and on that input for S steps and $c_{\overline{D}}$ on the same input for S steps, repeating and gradually increasing S until one of c_D or $c_{\overline{D}}$ halts. Since any given $n \in \mathbf{N}$ is either in D or $\overline{D}$, exactly one of the checkers must eventually halt. If c_D halts then return the value 1, and if $c_{\overline{D}}$ halts then return the value 0.

Problem 3 [15 points]. **For each of the following classes of sets indicate which closure properties it has (√) or does not have (×):**

	intersection	complement	mapping a set S to $f(S)$	mapping a set S to $H^{(S)}$
decidable sets	√	√	×	×
checkable sets	√	×	√	×
expressible sets	√	√	√	√
finite sets	√	×	√	×

where f is any total computable function and $H^{(S)}$ is the self-halting problem relative to S (*i.e.*, $H^{(S)} = \left\{c \in \mathbf{Com}_{fv} \mid \{c\}_{\rho_S, X_1, X_1}(\#c) \text{ halts}\right\}$).

Explanation:

- 1. The intersection of two decidable sets can be decided by running the two deciders in sequence and returning 1 if either returned 1, and 0 otherwise.
  2. The complement can be decided by running a decider and returning 1 if it returns 0, and 0 if it returns 1.
  3. The right projection of a decidable set can be undecidable, so the decidable sets are not closed under arbitrary computable functions.
  4. Even taking the trivially decidable set $\emptyset$, $H^{(\emptyset)} = H$, which is undecidable.
- 1. The intersection of two checkable sets can be checked by interleaving steps from the two checkers, and halting if either halts.
  2. The complement of a checkable set may not be checkable; *e.g.*, $\overline{H}$.
  3. Handout 33, Theorem 4 [**from 1993**] states that $f(C)$ is checkable if f is computable and C is checkable.
  4. In the solutions to Problem Set 9 [**from 1992**] we demonstrated that $S \leq_m S'$ implies $S \leq_T S'$ and that $H^{(S)} \not\leq_T S$ for any S. Thus, in particular, $H^{(H)} \not\leq_m H$; but all checkable sets are $\leq_m H$, so $H^{(H)}$ is not checkable, even though H is.
- We've already seen how to express most of these. The only tricky one is constructing $H^{(S)}$ for an expressible S, but we'll leave this one as an exercise for the reader.

- 1. The intersection of two finite sets is finite.
  2. The complement of a finite set is always infinite.
  3. If we map a finite set through any function, we'll get only a finite set of results.
  4. See above under "decidability."

Problem 4 [20 points]. **For $S \subseteq \mathbf{N}$, let $2S = \{2n \mid n \in S\}$ and likewise $S + 1 = \{n + 1 \mid n \in S\}$. For $S_1, S_2 \subseteq \mathbf{N}$, prove that $(2S_1) \cup (2S_2 + 1)$ is a least upper bound of S_1 and S_2 under many-one reducibility, $\leq_m$.**

The proof is virtually identical to that for the definition of "join" given in Problem Set 9 [**from 1992**]. Here the reductions from S_1 and S_2 to $(2S_1) \cup (2S_2 + 1)$ are $f(n) = 2n$ and $g(n) = 2n + 1$ respectively. $(2S_1) \cup (2S_2 + 1)$ is a least upper bound, because if $S_1 \leq_m S$ and $S_2 \leq_m S$ with reductions f' and g', then $(2S_1) \cup (2S_2 + 1) \leq_m S$ with reduction

$$h(n) = \begin{cases} f'(n/2) & \text{if } n \text{ is even} \\ g'((n-1)/2) & \text{if } n \text{ is odd} \end{cases}$$

Problem 5 [25 points].

5(a) [5 points]. **Explain why if a set S is many-one reducible to H (in symbols, $S \leq_m H$), then S is checkable. You may cite without proof any relevant properties of H and $\leq_m$ established in class or notes.**

H is checkable, and checkability inherits downwards.

5(b) [20 points]. **Prove conversely that every checkable set is $\leq_m H$. *Hint*: Similar to the proof that $H \leq_m H_0$ or the proof of Rice's Theorem. But Rice's theorem does *not* apply.**

Any checkable set S with checker c_S can be reduced to H under the function

$$f(n) = \#(X_1 := n; c_S).$$

This program code returned for n represents a function that will halt on its own number (or any input) iff c_S halts on input n. Thus, $f(n) \in H$ iff $n \in S$, so $S \leq_m H$.

Final Exam

Instructions. This is a **closed book, closed note** exam. There are six problems, on pages 2–14 of this booklet. Write your solutions for all problems on this exam sheet in the spaces provided, including your *name on each sheet.* Don't accidentally skip a page. Ask for further blank sheets if you need them.

At the end of the exam we have included an Appendix listing the syntax of **IMP**, and a Glossary of notation. You have seen all of the material in the Appendix and Glossary before; it is included for your reference only.

GOOD LUCK!

NAME: ______________________

Problem	Points	Score
1	30	
2	20	
3	20	
4	20	
5	20	
6	20	
Total	130	

Problem 1 [30 points].

1(a) [10 points]. Carefully state Gödel's Incompleteness Theorem.

1(b) [10 points]. Carefully state Rice's Theorem.

1(c) [10 points]. Let $\mathbf{Validity}_1 = \{ b \in \mathbf{Bexp} \mid \ \models b \}$.

Show that $\mathbf{Validity}_1$ is not checkable.

Problem 2 [20 points]. Give an example for each of the following.

2(a) [4 points]. A nonexpressible set.

2(b) [4 points]. A decidable set containing a nonexpressible set.

2(c) [4 points]. A decidable subset of **N** that is infinite and whose complement is infinite.

2(d) [4 points]. A set whose every subset is decidable.

2(e) [4 points]. Sets S_1 and S_2 such that $S_1 \leq_T S_2$ but $S_1 \not\leq_m S_2$.

Problem 3 [20 points]. Define the set S_{23} by

$$S_{23} ::= \{ c \in \mathbf{Com} \mid \{c\}(2)\downarrow \text{ and } \{c\}(3)\uparrow \}.$$

3(a) [7 points]. Show that S_{23} is not checkable.

3(b) [6 points]. Show that $\overline{S_{23}}$ is not checkable.

3(c) [7 points]. Explain why S_{23} is expressible.

Problem 4 [20 points]. For $i = 0, 1$, define the set S_i of commands that "self-halt" at i:

$$S_i = \{ c \in \mathbf{Com} \mid \{c\}(\#c) = i \}.$$

Let D be a set such that $S_0 \subseteq D$ and $S_1 \subseteq \overline{D}$.

4(a) [10 points]. Suppose that $\{c\}$ is a total, 0–1 valued function. Prove that

$$\text{char}_D(\#c) = 1 \quad \text{iff} \quad \{c\}(\#c) \neq 1,$$

where char_D is the characteristic function of D.

4(b) [10 points]. Conclude that D is not decidable.

Problem 5 [20 points]. We write $\mathbf{Bexp}_0$ for the set of *arithmetic-operation-free* boolean expressions. If $b \in \mathbf{Bexp}_0$ and $a_1 = a_2$ or $a_1 \leq a_2$ appears in b, then a_1 and a_2 can only be locations or numbers.

A *signature* $\mathcal{S}$ is a finite set of locations and numerical constants. For example, the set

$$\{X, Y, Z, -3, 7\}$$

is a signature.

If σ is a state and $n \in \mathbf{N}$, we define $\sigma(n) \stackrel{\text{def}}{=} n$. This will allow us to apply states to any element of a signature.

For any state σ and signature $\mathcal{S}$, we define $R^{\mathcal{S}}_{\sigma}$, a binary relation on $\mathcal{S}$, by the following rule:

$$s_1 \; R^{\mathcal{S}}_{\sigma} \; s_2 \quad \text{iff} \quad \sigma(s_1) \leq \sigma(s_2).$$

5(a) [7 points]. Describe a computational procedure that takes as input an expression $b \in \mathbf{Bexp}_0$ and a $k \times k$ table describing a relation $R^{\mathcal{S}}_{\sigma}$, and outputs the truth value $[\![b]\!]\sigma$, where $\mathcal{S}$ contains all the locations and numbers in b, and $k = |\mathcal{S}|$.

5(b) [6 points]. Describe a computational procedure that takes as input a signature S and a table describing a binary relation R on S, and outputs **true** if there is some state σ such that $R = R^S_\sigma$, and **false** otherwise.

5(c) [7 points]. Let $\mathbf{Validity}_0 = \{\, b \in \mathbf{Bexp}_0 \mid\ \models b \,\}$.

Show that $\mathbf{Validity}_0$ is decidable.

Problem 6 [20 points]. In this problem we consider the subset $\mathbf{Com}_0$ of commands which are *arithmetic-operation-free*. That is, all assignments must be of the form $X := Y$ or $X := n$, and all Boolean expressions are in $\mathbf{Bexp}_0$ from Problem 5.

Let Ω be an abbreviation for the "infinite loop" command **while true do skip**. We say a command is *simple* iff it contains no **while**-commands other than Ω.

6(a) [12 points]. Prove that $\mathbf{Bexp}_0$ is expressive for the simple $\mathbf{Com}_0$'s.

That is, show that for any simple $c \in \mathbf{Com}_0$, and any $b \in \mathbf{Bexp}_0$, there is some $W(c, b) \in \mathbf{Bexp}_0$ expressing the weakest precondition for b under c.

Actually $\mathbf{Bexp}_0$ is expressive for *all* $\mathbf{Com}_0$'s because

Fact 1. For every $c \in \mathbf{Com}_0$ there is a simple $c' \in \mathbf{Com}_0$ such that $[\![c]\!] = [\![c']\!]$.

We will not prove this fact, but ask you to work out an example:

6(b) [8 points]. Give a simple $\mathbf{Com}_0$ equivalent to

$$\mathbf{while}\, X_1 = X_2\, \mathbf{do}\, (X_1 := X_2;\ X_2 := X_3;\ X_3 := 4).$$

Appendix: Syntax of IMP

The arithmetic expressions **Aexp**:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

The boolean expressions **Bexp**:

$$b ::= \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

The commands **Com**:

$$c ::= \mathbf{skip} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c$$

Glossary

$\{c\}$	The partial function on $\mathbf{N}$ computed by command c with input register X_1 and output register X_1.
$f(n)\downarrow$	The partial function f is defined on argument n.
$f(n)\uparrow$	The partial function f is not defined on argument n.
$S_1 \leq_m S_2$	The set S_1 is many-one reducible to S_2, that is, there is a total computable function f such that for all n, $n \in S_1$ iff $f(n) \in S_2$.
Σ	The set of states.
$[\![b]\!] : \Sigma \to \mathbf{T}$	The meaning of the boolean expression b.
$[\![c]\!] : \Sigma \to \Sigma$	The meaning of the command c.
$S_1 \leq_T S_2$	The set S_1 is Turing-reducible to S_2, that is, the characteristic function of S_1 can be computed by a program that has the characteristic function of S_2 available as a built-in primitive.

Final Exam Solution

(This was a closed book, closed note exam. There were six problems, worth 130 points total).

Problem 1 [30 points].

1(a) [10 points]. Carefully state Gödel's Incompleteness Theorem.

Answer: For every sound proof system $\mathcal{P}$, $\mathbf{Provable}_{\mathcal{P}} \subsetneq \mathbf{Validity}$.

1(b) [10 points]. Carefully state Rice's Theorem.

Answer: Let P be any nontrivial property of checkable sets such that the empty set does not have property P. Let

$$\mathbf{Com}_P \quad = \quad \{ c \in \mathbf{Com} \mid c \text{ is a checker for a set with property } P \}.$$

Then $S \leq_m \mathbf{Com}_P$ for every checkable set S.

1(c) [10 points]. Let $\mathbf{Validity}_1 = \{ b \in \mathbf{Bexp} \mid\ \models b \}$.

Show that $\mathbf{Validity}_1$ is not checkable.

Answer: We will show that $\overline{H_{10}} \leq_m \mathbf{Validity}_1$. Since $\overline{H_{10}}$ is not checkable, and non-checkability inherits up, this implies that $\mathbf{Validity}_1$ is not checkable.

Recall the definition of $\overline{H_{10}}$:

$$\overline{H_{10}} = \{ a \in \mathbf{Aexp} \mid\ \models \neg(a = 0) \}.$$

Note that for any $a \in \mathbf{Aexp}$,

$$\begin{aligned} a \in \overline{H_{10}} \quad & \text{iff} \quad \models \neg(a = 0) \\ & \text{iff} \quad \neg(a = 0) \in \mathbf{Validity}_1. \end{aligned}$$

Furthermore, there is a total computable function $f : \mathbf{N} \to \mathbf{N}$ satisfying

$$f(\#a) \quad = \quad \#(\neg(a = 0))$$

for all $a \in \mathbf{Aexp}$. Thus $\overline{H_{10}} \leq_m \mathbf{Validity}_1$.

Problem 2 [20 points]. Give an example for each of the following.

2(a) [4 points]. A nonexpressible set.

Answer: The set **Validity** $= \{ A \in \textbf{Assn} \mid \models A \}$.

2(b) [4 points]. A decidable set containing a nonexpressible set.

Answer: The set **N**(or any other infinite decidable set).

2(c) [4 points]. A decidable subset of **N** that is infinite and whose complement is infinite.

Answer: The set of even integers, the set of odd integers, the set of primes, etc.

2(d) [4 points]. A set whose every subset is decidable.

Answer: The empty set, or any other finite set.

2(e) [4 points]. Sets S_1 and S_2 such that $S_1 \leq_T S_2$ but $S_1 \not\leq_m S_2$.

Answer: Take $S_1 = H$ and $S_2 = \overline{S_1}$, or replace H by any checkable set which is not decidable.

Problem 3 [20 points]. Define the set S_{23} by

$$S_{23} \quad = \quad \{ c \in \textbf{Com} \mid \{c\}(2)\downarrow \text{ and } \{c\}(3)\uparrow \}.$$

3(a) [7 points]. Show that S_{23} is not checkable.

Answer: We show $\overline{H} \leq_m S_{23}$. $\overline{H}$ is not checkable, and non-checkability inherits up, so this implies that S_{23} not checkable.

Recall the definition of $\overline{H}$:

$$\overline{H} \quad = \quad \{ c \in \textbf{Com} \mid \{c\}(\#c)\uparrow \}.$$

For any $c \in \textbf{Com}$, let $\hat{c} \in \textbf{Com}$ be defined by

$$\hat{c} \quad \equiv \quad \textbf{if } X_1 = 2 \textbf{ then skip else } (X_1 := (\#c); c).$$

Then for any $c \in \textbf{Com}$,

$$\{\hat{c}\}(2)\downarrow, \quad \text{and} \quad \{\hat{c}\}(3)\uparrow \text{ iff } \{c\}(\#c)\uparrow.$$

Thus $c \in \overline{H}$ iff $\hat{c} \in S_{23}$. Furthermore, there is a total computable function $f_1 : \mathbf{N} \rightarrow \mathbf{N}$ satisfying

$$f_1(\#c) \quad = \quad \#(\hat{c}).$$

Therefore $\overline{H} \leq_m S_{23}$, and hence S_{23} is not checkable.

3(b) [6 points]. Show that $\overline{S_{23}}$ is not checkable.

Answer: Note that

$$\overline{S_{23}} = \{ c \in \mathbf{Com} \mid \{c\}(2)\uparrow \text{ or } \{c\}(3)\downarrow \}.$$

For any $c \in \mathbf{Com}$, let $\check{c} \in \mathbf{Com}$ be defined by

$$\check{c} \equiv \mathbf{if}\, X_1 = 3\, \mathbf{then}\, \Omega\, \mathbf{else}\, (X_1 := (\#c); c),$$

where Ω is any "infinite loop". Then for any $c \in \mathbf{Com}$,

$$\{\check{c}\}(2)\uparrow \text{ iff } \{c\}(\#c)\uparrow, \quad \text{and} \quad \{\check{c}\}(3)\uparrow.$$

Thus $c \in \overline{H}$ iff $\check{c} \in \overline{S_{23}}$. Furthermore, there is a total computable function $f_2 : \mathbf{N} \to \mathbf{N}$ satisfying

$$f_2(\#c) = \#(\check{c}).$$

Therefore $\overline{H} \leq_m \overline{S_{23}}$, and hence $\overline{S_{23}}$ is not checkable.

3(c) [7 points]. Explain why S_{23} is expressible.

Answer: For $i = 2, 3$, define the sets

$$S_i = \{ c \in \mathbf{Com} \mid \{c\}(i)\downarrow \}.$$

The sets are checkable (by a program that simulates the execution of its argument c on input 2 or 3), and therefore expressible. Furthermore,

$$S_{23} = S_2 \cap \overline{S_3},$$

and expressibility is closed under complement and intersection. Thus S_{23} is expressible.

Problem 4 [20 points]. For $i = 0, 1$, define the set S_i of commands that "self-halt" at i:

$$S_i = \{ c \in \mathbf{Com} \mid \{c\}(\#c) = i \}.$$

Let D be a set such that $S_0 \subseteq D$ and $S_1 \subseteq \overline{D}$.

4(a) [10 points]. Suppose that $\{c\}$ is a total, 0–1 valued function. Prove that

$$\text{char}_D(\#c) = 1 \quad \text{iff} \quad \{c\}(\#c) \neq 1,$$

where char_D is the characteristic function of D.

Answer: Because $\{c\}$ is a total, 0–1 valued function, either $\{c\}(\#c) = 0$ or $\{c\}(\#c) = 1$. So we have

$$c \in S_0 \quad \text{iff} \quad c \notin S_1.$$

Then we reason as follows:

$$\begin{aligned} \text{char}_D(\#c) = 1 \quad & \text{iff} \quad c \in D \\ & \text{iff} \quad c \in S_0 \\ & \text{iff} \quad c \notin S_1 \\ & \text{iff} \quad \{c\}(\#c) \neq 1. \end{aligned}$$

4(b) [10 points]. Conclude that D is not decidable.

Answer: Suppose by way of contradiction that D is decidable, that is, there is a $d \in$ **Com** for D such that $\{d\} = \text{char}_D$. So by part (a),

$$\{d\}(\#c) = 1 \quad \text{iff} \quad \{c\}(\#c) \neq 1,$$

for all $c \in$ **Com** such that $\{c\}$ is a total, 0–1 valued function. But $\{d\}$ is a total, 0–1 valued function, so we can choose c to be d in the above "iff" to get an immediate contradiction.

Problem 5 [20 points]. We write $\mathbf{Bexp}_0$ for the set of *arithmetic-operation-free* boolean expressions. If $b \in \mathbf{Bexp}_0$ and $a_1 = a_2$ or $a_1 \leq a_2$ appears in b, then a_1 and a_2 can only be locations or numbers.

A *signature* $\mathcal{S}$ is a finite set of locations and numerical constants. For example, the set

$$\{X, Y, Z, -3, 7\}$$

is a signature.

If σ is a state and $n \in \mathbf{N}$, we define $\sigma(n) \stackrel{\text{def}}{=} n$. This will allow us to apply states to any element of a signature.

For any state σ and signature $\mathcal{S}$, we define $R_\sigma^{\mathcal{S}}$, a binary relation on $\mathcal{S}$, by the following rule:

$$s_1 \; R_\sigma^{\mathcal{S}} \; s_2 \quad \text{iff} \quad \sigma(s_1) \leq \sigma(s_2).$$

5(a) [7 points]. Describe a computational procedure that takes as input an expression $b \in \mathbf{Bexp}_0$ and a $k \times k$ table describing a relation $R_\sigma^{\mathcal{S}}$, and outputs the truth value $[\![b]\!]\sigma$, where $\mathcal{S}$ contains all the locations and numbers in b, and $k = |\mathcal{S}|$.

Answer: We define a procedure by induction on b.

- If $b \equiv s_0 \leq s_1$, then check the table to see if $s_0 \; R^S_\sigma \; s_1$.
  If $s_0 \; R^S_\sigma \; s_1$ holds then answer **true**, otherwise answer **false**.
- If $b \equiv s_0 = s_1$, then check the table to see if $s_0 \; R^S_\sigma \; s_1$ and $s_1 \; R^S_\sigma \; s_0$.
  If both hold then answer **true**, otherwise answer **false**.
- If $b \equiv$ **true** then answer **true**.
- If $b \equiv$ **false** then answer **false**.
- If $b \equiv \neg(b_0)$, then by induction, we can compute $[\![b_0]\!]\sigma$. If $[\![b_0]\!]\sigma =$ **true** answer **false**, and if $[\![b_0]\!]\sigma =$ **false** answer **true**.
- If $b \equiv (b_0 \wedge b_1)$, then by induction we can compute $[\![b_0]\!]\sigma$ and $[\![b_1]\!]\sigma$. If $[\![b_0]\!]\sigma =$ **true** and $[\![b_1]\!]\sigma =$ **true** answer **true**; otherwise answer **false**.
- If $b \equiv (b_0 \vee b_1)$, then by induction we can compute $[\![b_0]\!]\sigma$ and $[\![b_1]\!]\sigma$. If $[\![b_0]\!]\sigma =$ **true** or $[\![b_1]\!]\sigma =$ **true** answer **true**; otherwise answer **false**.

5(b) [6 points]. Describe a computational procedure that takes as input a signature $\mathcal{S}$ and a table describing a binary relation R on $\mathcal{S}$, and outputs **true** if there is some state σ such that $R = R^S_\sigma$, and **false** otherwise.

Answer: We need to check that

- For every $s \in \mathcal{S}$, we have $s \; R \; s$.
- For every $s_1, s_2 \in \mathcal{S}$, we have $s_1 \; R \; s_2$ or $s_2 \; R \; s_1$ (or both).
- For every $s_1, s_2, s_3 \in \mathcal{S}$, if $s_1 \; R \; s_2$ and $s_2 \; R \; s_3$, we have $s_1 \; R \; s_3$.
- For all numbers $n_1, n_2 \in \mathcal{S}$, we have $n_1 \leq n_2$ iff $n_1 \; R \; n_2$.

Since $\mathcal{S}$ is finite, we can list all the elements of $\mathcal{S}$ and use the table to perform the first test; then list all pairs of elements of $\mathcal{S}$ and perform the second and fourth tests; and list all triple of elements of $\mathcal{S}$ and perform the third test.

If R passes all the tests, we answer **true**, otherwise we answer **false**.

5(c) [7 points]. Let $\textbf{Validity}_0 = \{ b \in \textbf{Bexp}_0 \mid \models b \}$.

Show that $\textbf{Validity}_0$ is decidable.

Answer: To decide if $b \in \textbf{Bexp}_0$ is valid, let $\mathcal{S}$ be the set of locations and numbers that appear in b and let $k = |\mathcal{S}|$. For each possible $k \times k$ table of truth values representing a binary relation on $\mathcal{S}$, use the procedure of part (b) to check whether the table represents an R^S_σ for some σ; if so, use the procedure of part (a) to compute $[\![b]\!]\sigma$. Then b is valid iff the procedure returns **true** for every one of the (at most 2^{k^2}) tables.

Problem 6 [20 points]. In this problem we consider the subset $\mathbf{Com}_0$ of commands which are *arithmetic-operation-free*. That is, all assignments must be of the form $X := Y$ or $X := n$, and all Boolean expressions are in $\mathbf{Bexp}_0$ from Problem 5.

Let Ω be an abbreviation for the "infinite loop" command **while true do skip**. We say a command is *simple* iff it contains no **while**-commands other than Ω.

6(a) [12 points]. Prove that $\mathbf{Bexp}_0$ is expressive for the simple $\mathbf{Com}_0$'s.

That is, show that for any simple $c \in \mathbf{Com}_0$, and any $b \in \mathbf{Bexp}_0$, there is some $W(c, b) \in \mathbf{Bexp}_0$ expressing the weakest precondition for b under c.

Answer: We define $W(c, b)$ by induction on c.

$$\begin{aligned} W(\Omega, b) &= \mathbf{true}, \\ W(\mathbf{skip}, b) &= b, \\ W(X := Y, b) &= b[Y/X], \\ W(X := n, b) &= b[n/X], \\ W((c_1; c_2), b) &= W(c_1, W(c_2, b)), \\ W(\mathbf{if}\ b_0\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, b) &= (b_0 \Rightarrow W(c_1, b)) \wedge (\neg b_0 \Rightarrow W(c_2, b)). \end{aligned}$$

Clearly if $b \in \mathbf{Bexp}_0$ and $c_1, c_2 \in \mathbf{Com}_0$, then every right-hand side above is in $\mathbf{Bexp}_0$.

End answer.

Actually $\mathbf{Bexp}_0$ is expressive for *all* $\mathbf{Com}_0$'s because

Fact 1. For every $c \in \mathbf{Com}_0$ there is a simple $c' \in \mathbf{Com}_0$ such that $[\![c]\!] = [\![c']\!]$.

We will not prove this fact, but ask you to work out an example:

6(b) [8 points]. Give a simple $\mathbf{Com}_0$ equivalent to

$$\mathbf{while}\ X_1 = X_2\ \mathbf{do}\ (X_1 := X_2;\ X_2 := X_3;\ X_3 := 4).$$

Answer:

$\mathbf{if}\ \neg(X_1 = X_2)\ \mathbf{then\ skip}$
$\mathbf{else\ if}\ (X_1 = X_2) \wedge \neg(X_2 = X_3)\ \mathbf{then}\ (X_2 := X_3; X_3 := 4)$
$\mathbf{else\ if}\ (X_1 = X_2) \wedge (X_2 = X_3) \wedge \neg(X_3 = 4)\ \mathbf{then}\ (X_2 := 4; X_3 := 4)$
$\mathbf{else\ if}\ (X_1 = X_2) \wedge (X_2 = X_3) \wedge (X_3 = 4)\ \mathbf{then}\ \Omega.$

Appendix: Syntax of IMP

The arithmetic expressions **Aexp**:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

The boolean expressions **Bexp**:

$$b ::= \textbf{true} \mid \textbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

The commands **Com**:

$$c ::= \textbf{skip} \mid X := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c$$

Glossary

$\{c\}$	The partial function on $\mathbf{N}$ computed by command c with input register X_1 and output register X_1.
$f(n)\downarrow$	The partial function f is defined on argument n.
$f(n)\uparrow$	The partial function f is not defined on argument n.
$S_1 \leq_m S_2$	The set S_1 is many-one reducible to S_2, that is, there is a total computable function f such that for all n, $n \in S_1$ iff $f(n) \in S_2$.
Σ	The set of states.
$[\![b]\!] : \Sigma \to \mathbf{T}$	The meaning of the boolean expression b.
$[\![c]\!] : \Sigma \to \Sigma$	The meaning of the command c.
$S_1 \leq_T S_2$	The set S_1 is Turing-reducible to S_2, that is, the characteristic function of S_1 can be computed by a program that has the characteristic function of S_2 available as a built-in primitive.