Formal Semantics

Due Tuesday, April 12, at 11:59PM.

This assignment focuses on formal semantics and its application to the construction of interpreters and compilers.

Denotational Semantics

Problems 1–3 involve modifying the denotational semantics given in the lecture notes. For each problem, give only those parts of the semantics that have changed.

1. Modify the "Locations and Environments" semantics on page 14 of the printed version of the lecture notes so that variables are automatically initialized to zero at the time they are declared.
   [7 points]
2. Modify the "Gotos and Labels" semantics on page 18 of the printed version of the lecture notes so that *A* (answers) are the same as *V* (integers), and the answer given by a program is the value of the variable `Answer` at the end of termination. *Hint: it is sufficient to give a definition of the `done` function.*
   [7 points]
3. Again, start with the "Gotos and Labels" semantics, but modify the language to add a print statement. Add a grammar rule

   ```
   stm ::= PRINT exp
   ```

   Let *A*, the type of answers, be the same as *V list* (list of integer), and assume you are given the constructors `nil` and `cons` on lists. (Simply add `nil : V list` and `cons : V * V list -> V list` to the list of predefined functions.) Finally change the semantics so that a program like this one:

   ```
   A := -5;
   LOOP: ;
   PRINT A+5;
   A := A+1 ;
   IF A GOTO LOOP
   ```

   yields the answer *cons(0, cons(1, cons(2, cons(3, cons(4, nil)))))*. That is, the answer yielded by a program should be a list, in order, of everything printed.
   Hint: just fiddling with the "done" function is not enough. Think about the semantics of the `PRINT` statement.
   Warning: There is no variable called `Answer`, so don't try to use one.
   [11 points]

Each of these problems can be solved with a one– or two–line change to the denotational definitions given in class. For those of you who may wish to use TeX for your answers, the LaTeX source for the lecture notes is available online.

Applying semantics

In the remaining problems, you will apply ideas from ***both*** operational and denotational semantics to a tiny Java−like system: ``nano−Java.'' The nano−Java language does not really justify its name: it is like Impcore, but without even function definitions. But the *implementation* of nano−Java *does* justify the name; it is based on the Java Virtual Machine.

Before you tackle the remaining problems, please

- Read and understand the definition of the nano−Java Virtual Machine, or nano−JVM for short. We give you a definition in informal English and some code in ML.
- Read and understand the definition of the nano−Java source language . We give you some informal English and some code in ML.

Problems 4−7 involve nano−Java.

4. Write, using the notation of formal operational semantics, a definition of the nano−Java Virtual Machine. Be sure to say what the machine configuration is, and what are acceptable final states, as well as giving transition rules. To avoid repetition, give transition rules for only the following instructions:

   ```
   istore iload iconst iadd goto if_icmpeq label return
   ```

   To save bookkeeping, your semantics need not account for the behavior of `print`. If you want to describe the behavior of the `iprint` instruction, I'll take it for extra credit (**PRINT**). [17 points]
5. Using the ideas of continuation semantics, write a definitional interpreter for the nano−Java source language. The code for nano−Java is part of the nano−Java compiler source code Your code should be closely based on the definitional interpreter using expression continuations in the readings.

   To get your interpreter started, you'll want to use the following types and values.

   <nano−Java interpreter types>= **[D−>]**
   ```
   type V = int
   datatype A = ANSWER of V (* answer is the value returned by a program *)

   type Ide = string
   type L   = int           (* location *)
   type Env = Ide -> L      (* location of a variable *)
   type S   = L -> V        (* state == values in locations *)
   val upd  : S * L * V -> S
            = fn (sigma, n, v) => fn n' => if n = n' then v else sigma n'
   type C   = S -> A        (* continuation *)
   type K   = V -> C        (* expression continuation *)
   ```

   The unusual declaration of `A` makes `A` generative: you will have to apply `ANSWER` to a value to produce an answer. This little trick will enable the type checker to help you; it will keep you from confusing values and answers.

   The following three types define the meaning of a *statement*, the meaning of an *expression in a value context*, and the meaning of an *expression in a control−flow context*. An expression in a control−flow context should be a Boolean condition, and it takes *two* continuations: one to go to if the condition is true, and one to go to if the condition is false.

<nano–Java interpreter types>+= **[<–D]**
```
type Stm = Env -> C -> C
type Exp = Env -> K -> C
type Condition = Env -> C -> C -> C
```

Your interpreter should rely on five mutually recursive functions.

`P`	The meaning of a program: an answer `A`
`S`	The meaning of a statement: `Stm`
`Ss`	The meaning of a sequence of statements: `Stm`
`E`	The meaning of an expression in a value context: `Exp`
`B`	The meaning of an expression in a control–flow context: `Condition`

Please check the types of your functions by including the following lines after their definitions:

<type assertions for nano–Java interpreter>=
```
val _ = P  : prog      -> A
val _ = S  : stmt      -> Env -> C -> C
val _ = Ss : stmt list -> Env -> C -> C
val _ = E  : exp       -> Env -> K -> C
val _ = B  : exp       -> Env -> C -> C -> C
```

Notes:

- Choose carefully which expressions to pass to `E` and which you pass to `B`. When the only important thing about a value is whether it is zero, use `B`. When the value itself is important, use `E`. In the past, other students have had difficulty with this part of the problem. *If you implement Booleans by first converting to integers, then testing the integers, you will lose points.*
- While loops are tricky. To help you, here is one possible denotational rule (there are others):
  $\mathcal{S}[\![\mathtt{WHILE\ exp\ DO\ stm}]\!]\rho\theta = Y(\lambda f.\lambda\theta.\mathcal{E}[\![\mathtt{exp}]\!]\rho(\lambda\nu.\mathrm{ifnz}(\nu, \mathcal{S}[\![\mathtt{stm}]\!]\rho(f\theta), \theta)))\theta$
  This definition uses the `E` function; after you get your code working, you should *change over to the `B` function*.

  Because the Y combinator is not typeable in ML, you will not be able to use it. Use the function `fix` from the lecture notes on the lambda calculus. Also, you will need to beware of applicative–order versus normal–order evaluation. Remember, in ML you can delay something's evaluation by placing it under a lambda. Eta–expansion is a good way to do this.
- Not every nano–Java program is meaningful. In particular
  - A program that falls off the end without using `RETURN` is not meaningful.
  - A program that tries to use an uninitialized variable is not meaningful.

  Your definitional interpreter should raise an exception if it is given a program that is not meaningful.
- Your interpreter for `PRINT` should have an actual side effect of printing. Use this function:

  <printing function>=
  ```
  fun printVal n = app print ["  ", Int.toString n, "\n"]
  ```

  The timing is tricky–––you want to delay the call to `printVal` until you are actually given a state. Once you have a state, print the value, and then apply your continuation to the state.

- Don't be surprised if `RETURN` ignores its continuation. Of course it ignores its continuation! That's what continuations are for.
- Testing will be necessary. For your convenience, use the `primes` program from the nano–Java compiler source code.

Final note: *this problem involves much more thought than programming;* you can write a good solution by adding less than 40 lines of code to what is already provided. The hard part is mastering continuations.
[38 points]

6. One of the advantages of continuation semantics is the ease with which we can work with different kinds of answers. In this problem, you will make a very minor change to your interpreter that will enable you to answer questions about what values a program prints. You will use the modified interpreter to answer one such question.

   Change type of answer (`A`) to be

   ```
   type A = {printed : V list, returned : V}
   ```

   Change your interpreter so that *it no longer prints anything*. Instead, your interpreter should produce an answer of the new type `A`, which tells you what the nano–Java program *would have printed* had it been executed on the original interpreter. You will need to change your semantic equations (the definitions of your four functions) to work with the new `A` type.
   Hints:

   - Only two semantic equations should change———everything else should stay as is.
   - The answer to problem 3 above is relevant.

   Now use your new interpreter, and the `primes` program that is part of the nano–Java compiler source code, to write

   ```
   val earlyPrime : int -> bool = ...
   ```

   which determines whether `k` is in the first 20 primes by running the program and looking to see if `k` is printed.

   Note that you can get full credit for `earlyPrime` and for correct semantic equations even if you do not get your whole interpreter working.
   [11 points]
7. Add to nano–Java, and to your definitional interpreter, the short–circuit Boolean operators `AND` and `OR`, as well as Boolean `NOT`.
   Keep in mind the following algebraic laws:

   ```
   if p &q then s1 else s2  ===  if p then if q then s1 else s2 else s2
   if p || q then s1 else s2  ===  if p then s1 else if q then s1 else s2
   ```

   You should be able to discover a similar law for `NOT`, and by thinking about the laws, you should be able to discover very simple semantic equations for the new operators.
   Again, you can get full credit for handling the new constructs correctly even if your full interpreter does not work.
   [9 points]

Extra credit:

- **INPUT**. Add a `READ` expression to nano–Java, and update the semantics and interpreter accordingly.
  - ♦ Assume that you have a type `inputstream` with the following operations.

    ```
    <support for reading>= [D->]
    type inputstream
    val read : inputstream -> V * inputstream
    ```

    For your implementation, you can use `type inputstream = int list`.
  - ♦ The easy way to solve this problem is to let the state of your program include an `inputstream`. A somewhat slicker way (which is really equivalent) is to make a continuation take both a state and a stream:

    ```
    <support for reading>+= [<-D]
    type C = S -> inputstream -> A
    ```

    I actually recommend this latter approach. If you do it right, your nano–Java programs can have denotations of type `inputstream -> { printed : V list, returned : V }`.
  - ♦ Finally, rewrite the nano–Java `primes` program to use `READ`, and define a function `isPrime n` that works by enumerating the first `n` primes to see if `n` is among them.
- **BREAK**. Add break and continue to nano–Java, and update your definitional interpreter to handle them. (*Hint: pass 2 additional continuations to every statement.*)
- **COMPILE**. Add short–circuit Booleans to the nano–Java compiler. Change the primes program to use these operators, and measure the improvement in code size both before and after running the nano–JVM optimizer.

What to submit

For each problem above, submit

1. the changed parts of the semantics, either on paper or as part of the PostScript file `sem.ps`.
2. the changed parts of the semantics, either on paper or as part of the PostScript file `sem.ps`.
3. the changed parts of the semantics, either on paper or as part of the PostScript file `sem.ps`.
4. your operational semantics, either on paper or as part of the PostScript file `sem.ps`.
5. an interpreter in `ujint.sml`. This file should be complete and self–contained, including the definition of nano–Java and whatever else you need for it to work.
6. a self–contained interpreter in `ujintp.sml`
7. a self–contained interpreter in `ujintb.sml`

Please also submit a `README` file telling us *what you have completed* and *how much time you spent*.

The submit script for this homework is `~cs152/bin/submit-sem`.

- *<nano–Java interpreter types>*: D1, D2
- *<printing function>*: D1
- *<support for reading>*: D1, D2
- *<type assertions for nano–Java interpreter>*: D1