Programming Languages

Qualifier Exam

July 10, 2002

Exercise 1

Programming languages such as C allow certain program statements (such as the assignment statement) to produce a value as a result; this allows statements to be used inside larger expressions. E.g., the following is a legal C statement

```
x = (y = z+3) * (w++);
```

Consider the simple imperative language with the following syntax specification:

```
Program ::= Comm
Comm    ::= Comm  Comm
        |   if (Exp) Comm else Comm endif
        |   Id = Exp
        |   while (Exp) Comm endwhile
        |   noop
Exp     ::= Id
        |   Nat
        |   Exp + Exp
```

Answer the following questions:

1. extend the syntax specification to allow arbitrary statements to be used in expressions;
2. describe intuitively the meaning of the different statements used in the context of expressions;
3. provide the denotational semantics for the above language extended with statements as expressions; model the intuitive meaning described in the previous question;
4. discuss those topics that should be addressed when proceeding with the implementation of this extended language (focusing on the management of the statements as expressions)

Exercise 2

Consider a simple imperative programming language with the following syntax specification:

```
Program ::= Comm
Comm    ::= Comm  Comm
        |   if (Exp) Comm endif
        |   Id = Exp
        |   while (Exp) Comm endwhile
Exp     ::= Id
        |   Nat
        |   Exp + Exp
```

We would like to extend this language by providing the ability to associate labels with statements (e.g., an identifier placed before the statement in the program) and adding a `goto` *`label`* instruction that transfers control to the statement that has the specified label.

Answer the following questions:

1. provide the syntax specification of the extended language
2. provide the denotational semantics or detailed pseudocode for the extended language
3. assume that we intend to provide a meaningful handling of the goto statement in an imperative language that includes subroutines (allowing jumps between subroutines). Discuss syntax and semantics issues that would have to be addressed in order to set the framework for this language extension.

Sketch of solutions

Exercise 1

The syntax will include:

```
Program ::=  Comm
Comm    ::= Comm  Comm
        |   if (Exp) Comm else Comm endif
        |   Id = Exp
        |   while (Exp) Comm endwhile
        |   noop
Exp     ::= Id
        |   Nat
        |   Exp + Exp
        |   Comm
        |   (Exp)
```

There are different meanings that can be given (any reasonable one will be accepted). An intuitive solution is

- for the assignment statement, the same solution as C can be used (the meaning of Id = Exp is the value of Exp)
- for the if-then-else, the meaning should be the meaning of the selected command
- for the while loop the meaning should be the meaning of the last iteration of the loop
- for the sequencing of commands, the meaning should be the value of the command on the right
- for the noop perhaps zero could be a valid results or some undefined

If we follow this scheme, then the denotational semantics should be modified according to the following guidelines (I am assuming that we have only variables of type Nat):

- the arity of the valuation function for the Commands should now be:

$$\mathcal{C} : Comm \to Store \to Store \times Nat$$

- the arity of the valuation function for the Expressions should be:

$$\mathcal{E} : Exp \to Store \to Store \times Nat$$

- then it should be easy to build the valuation function for the different cases; e.g.,

$$
\begin{aligned}
\mathcal{C}[Comm_1\, Comm_2] = \quad & \lambda s.let\,(s_1, v_1) = \mathcal{C}[Comm_1]s\, in \\
& \mathcal{C}[Comm_2]s_1 \\
\\
\mathcal{C}[if\ E\ C_1\ else\, C_2] = \quad & \lambda s.let\,(s_1, v_1) = \mathcal{E}[E]s\, in \\
& (v_1\ equal\, 0 \rightarrow \mathcal{C}[C_1]s_1 \,\Box\, \mathcal{C}[C_2]s_1) \\
\\
\mathcal{E}[Comm] = \quad & \lambda s.\mathcal{C}[Comm]s \\
\\
\mathcal{E}[E_1 + E_2] = \quad & \lambda s.let\,(s_1, v_1) = \mathcal{E}[E_1]s\, in \\
& let\,(s_2, v_2) = \mathcal{E}[E_2]s_1\, in \\
& (s_2, v_1 + v_2)
\end{aligned}
$$

Implementation just needs to worry about the order of evaluation since an operand of an expression now can have side-effects.

Exercise 2

The syntax specification can be as follows:

```
Program  ::= LComm
LComm    ::= Labeled LComm
          |   Labeled
Labeled  ::= Label Comm
          |   Comm
Comm     ::= if (Exp) Comm endif
          |   Id = Exp
          |   while (Exp) Comm endwhile
          |   goto Label
Exp      ::= Id
          |   Nat
          |   Exp + Exp
Label    ::= Id
```

The denotational semantics should account for the need to maintain the mapping of the labels to points in the program. The easiest way seems to be to create different functions for the different segments of the program, assuming labels for each command in the program (make up arbitrary labels for commands that do not have a label attached).

Let us add a semantic algebras for the labels:

$$Lab = Label \mapsto (Command \times Label)$$

We can start by creating a function that maps labels to program statements

$$\begin{aligned}
&\mathcal{L}[Labeled\,LComm] = let\,(\ell, map) = \mathcal{L}[LComm]\; in\\
&\qquad let\,(\ell_1, syntax) = \mathcal{L}\dashv\lfloor[Labeled]\; in\\
&\qquad\qquad (\ell_1, update\; map\; \ell_1\; (syntax, \ell))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{L}[Labeled] = let\,(\ell, syntax) = \mathcal{L}\dashv\lfloor[Labeled]\; in\\
&\qquad (\ell, update\; newmap\; \ell\; (syntax, nop))
\end{aligned}$$

The intuition is that each label is mapped to the syntax of the command associated to the label plus the label of the immediately following command.

The valuation function of the command now needs to take as input this mapping along with the store:

$$\mathcal{C} : Comm \to Label \to Lab \to Store \to Store \times Label$$

It can be defined as follows:

$$\begin{array}{ll}
\mathcal{C}[Id = Exp] = & \lambda\ell.\lambda map.\lambda s.(update\; s\; Id\; \mathcal{E}[Exp]s, (snd\; (map\; \ell)))\\
\mathcal{C}[goto\; label] = & \lambda\ell.\lambda map.\lambda s.(s, label)\\
\mathcal{C}[if(Exp)Commendif] = & \lambda\ell.\lambda map.\lambda s.(\mathcal{E}[Exp]s\; equal\; zero \to (s, (snd\; (map\; ell)))\; \Box\\
& \mathcal{C}[Comm](\ell, map, s)
\end{array}$$

and so on...

The last question is completely open and I would be looking just for something that has a minimal of sense; there are various issues that should be addressed:

- if I perform a goto from a function, what happens to the function we are leaving? Should be kept alive and part of the dynamic chain? Should it be removed from the dynamic chain? Should we remove other elements from the dynamic chain?
- we are jumping inside another function. If the function already has some activations in the current dynamic chain, should we try to reuse some of these activations (maybe the most recent one)? Or should we initiate a new activation? Should we add it to the end of the dynamic chain? How about the parameters and the local environment for the new activation?
- what happen when the function we jump into returns?