5CCS2PLD: Programming Language Design Paradigms
Maribel Fernández

Week 5 revision tasks
Selected questions from past exams

Exercise 1:

1. The following grammar describes the concrete syntax of the programming language RBT used to program a robot moving in a grid:

$$\begin{array}{lll} P & ::= & Decl \text{ ; } \texttt{begin } StatList \texttt{ end} \\ Decl & ::= & Num * Num \\ Num & ::= & Digit \mid Digit\, Num \\ Digit & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ StatList & ::= & Stat \mid StatList \text{ ; } StatList \\ Stat & ::= & \texttt{init} \mid \texttt{north}(Num) \mid \texttt{south}(Num) \mid \\ & & \texttt{east}(Num) \mid \texttt{west}(Num) \end{array}$$

   For example, the following is an RBT program; we will call it P_1:

```
100 * 100;
begin
   init; north(5); east(5)
end
```

   The user manual for RBT warns programmers about the ambiguity in this grammar.

   Using an example, show that the grammar given above is indeed ambiguous.

   An ambiguous grammar is a grammar where at least one word has more than one left-most derivation. The grammar above is ambiguous since any program with more than 2 statements has at least 2 alternative derivations. For example, the program above can be parsed as

$$\begin{array}{rl} P \rightarrow & Decl;\ \texttt{begin } StatList \texttt{ end} \rightarrow \\ & Decl;\ \texttt{begin } StatList; StatList \texttt{ end} \rightarrow \\ & Decl;\ \texttt{begin } Stat; StatList \texttt{ end} \rightarrow \\ & Decl;\ \texttt{begin } Stat; StatList; StatList \texttt{ end} \rightarrow \ldots \end{array}$$

   or

$$\begin{array}{rl} P \rightarrow & Decl;\ \texttt{begin } StatList \texttt{ end} \rightarrow \\ & Decl;\ \texttt{begin } StatList; StatList \texttt{ end} \rightarrow \\ & Decl;\ \texttt{begin } StatList; StatList; StatList \texttt{ end} \rightarrow \ldots \end{array}$$

2. The abstract syntax of the language RBT is defined by the grammar:

$$\begin{array}{lll} P & ::= & prog(m, n, C) \\ C & ::= & skip \mid Init \mid N(k) \mid S(k) \mid E(k) \mid W(k) \mid seq(C, C) \end{array}$$

   where m, n, k represent natural numbers ($m, n, k \geq 0$).

   A compiler for RBT has parsed the program P_1 defined in part 1 and produced an abstract syntax tree.

Give a possible abstract syntax tree for the example program P_1.

An abstract syntax for P_1 should have root $prog$ and three subtrees. The first two subtrees contain just one leaf each, labelled by the number 100, and the third subtree represents the commands.

In this case, there are two alternative representations for the commands, so we can have a subtree

$seq(Init, seq(N(5), E(5))$

or $seq(seq(Init, N(5)), E(5))$.

Both are valid (due to the ambiguous concrete syntax grammar, there is more than one abstract syntax tree for this program).

3. The following axioms and rules define the big-step semantics of some of the commands in RBT.

   Configurations have the form $\langle prog(m, n, C), (x, y)\rangle$, abbreviated as $\langle C, (x, y)\rangle_{m,n}$. Here $prog(m, n, C)$ represents a program, as specified by the grammar given in part 2, and (x, y) is the current position of the robot in the grid (x and y are natural numbers representing the coordinates of the robot in the grid).

$$\frac{}{\langle skip, (x, y)\rangle_{m,n} \Downarrow \langle skip, (x, y)\rangle_{m,n}}$$

$$\frac{}{\langle Init, (x, y)\rangle_{m,n} \Downarrow \langle skip, (1, 1)\rangle_{m,n}}$$

$$\frac{}{\langle N(k), (x, y)\rangle_{m,n} \Downarrow \langle skip, (x, y + k)\rangle_{m,n}} \; if\ y + k \leq m$$

$$\frac{}{\langle N(k), (x, y)\rangle_{m,n} \Downarrow \langle skip, (x, m)\rangle_{m,n}} \; if\ y + k > m$$

$$\frac{}{\langle E(k), (x, y)\rangle_{m,n} \Downarrow \langle skip, (x + k, y)\rangle_{m,n}} \; if\ x + k \leq n$$

$$\frac{}{\langle E(k), (x, y)\rangle_{m,n} \Downarrow \langle skip, (n, y)\rangle_{m,n}} \; if\ x + k > n$$

$$\frac{\langle C_1, (x, y)\rangle_{m,n} \Downarrow \langle skip, (x', y')\rangle_{m,n} \quad \langle C_2, (x', y')\rangle_{m,n} \Downarrow \langle skip, (x'', y'')\rangle_{m,n}}{\langle seq(C_1, C_2), (x, y)\rangle_{m,n} \Downarrow \langle skip, (x'', y'')\rangle_{m,n}}$$

   Using the big-step semantics, describe the execution of the program P_1 given in part 1 by a robot initially placed at position (5,5) in the grid. In particular, give the position of the robot at the end of the program execution.

The program P_1 consists of a declaration (grid size 100 $\times$ 100) and a sequence of commands: init, north(5), east(5). To execute P_1 we apply the rules given on its abstract syntax tree.

There are two alternative trees, but both produce the same result.

If we use $seq(Init, seq(N(5), E(5))$ then the initial configuration is

$\langle seq(Init, seq(N(5), E(5)), (5, 5)\rangle_{100,100}$.

According to the rule for seq, the $Init$ command is executed first, and positions the robot in the square (1,1).

The next command moves the robot to position (1,6) (that is, same x coordinate, $y + 5$), as specified by the rule for $N(k)$, here m = n = 100.

Finally, the robot moves right 5 squares to position (6,6) as specified by the rule for $E(k)$.

4. A PLD student has implemented a compiler for RBT, with the following "optimisation":

   > *Whenever the abstract syntax tree of an RBT program contains a subtree of the form* $seq(N(i), N(j))$*, replace this subtree with* $N(i + j)$.

   Is this optimisation correct? Either show that the optimised program is equivalent to the original one according to the given operational semantics (i.e., the optimisation is correct) or give an example showing that the optimised program and the original one are not equivalent.

   The optimisation is correct, because the two programs produce the same result.

   According to the operational semantics, for any (x, y):

   $\langle seq(N(i), N(j)), (x, y)\rangle_{m,n} \Downarrow \langle skip, (x, z)\rangle_{m,n}$

   and

   $\langle N(i + j), (x, y)\rangle_{m,n} \Downarrow \langle skip, (x, z)\rangle_{m,n}$

   where $z = min(y + i + j, m)$.

   Therefore the two commands are equivalent.

Exercise 2:

1. We add to the syntax of the language SIMP a post-test logically controlled loop:

   ```
   do C while B
   ```

   where the loop-body C will be repeated until the boolean expression B evaluates to False.

   Give a formal definition of the semantics of this command (you can either give additional transition rules for the abstract machine, or give rules to extend the small-step semantics or the big-step semantics of SIMP).

2. Show that this extension does not change the computational power of the language by giving a translation of `do C while B` in terms of the constructs that already exist in SIMP.

1. We can add to the big-step semantics the following rules:

$$\frac{\langle C, s\rangle \Downarrow \langle skip, s'\rangle \quad \langle B, s'\rangle \Downarrow \langle True, s''\rangle \quad \langle do\, C\ while\ B, s''\rangle \Downarrow \langle skip, s'''\rangle}{\langle do\, C\ while\ B, s\rangle \Downarrow \langle skip, s'''\rangle}$$

$$\frac{\langle C, s\rangle \Downarrow \langle skip, s'\rangle \quad \langle B, s'\rangle \Downarrow \langle False, s''\rangle}{\langle do\, C\ while\ B, s\rangle \Downarrow \langle skip, s''\rangle}$$

2. We can encode `do C while B as C;while B do C`, which has the same semantics according to the given rules. In other words, either both configurations are non-terminating, or they both reach the same final state, as can be seen by analysing the rules.

Exercise 3:

1. Recall the rules defining the semantics of the assignment and the if-then-else commands in SIMP (big-step semantics):

$$\frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle}{\langle x := E, s\rangle \Downarrow \langle skip, s'[x \mapsto n]\rangle}$$

$$\frac{\langle B, s\rangle \Downarrow \langle True, s'\rangle \quad \langle C_1, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s\rangle \Downarrow \langle skip, s''\rangle}$$

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s\rangle \Downarrow \langle skip, s''\rangle}$$

   Show that the SIMP program:

   ```
   if 2 < 0 then x := 0 else x := 1
   ```

   is equivalent to `x:= 1` (that is, has the same semantics as the program `x:= 1`).

2. We now replace the last rule for the if-then-else command given above by the following rule:

$$\frac{\langle B, s\rangle \Downarrow \langle False, s'\rangle}{\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s\rangle \Downarrow \langle skip, s'\rangle}$$

   (a) How is the semantics of the language affected by this change?

   (b) Using this new rule, what is the value of `x` in memory after the execution of the program:

   ```
   if 2 < 0 then x := 0 else x := 1
   ```

1. Using the big-step semantics of SIMP, we can show that both programs reach the same state $s[x \mapsto 1]$ when they are executed in a state s. More precisely, we prove that:

   $\langle \text{if } 2 < 0 \text{ then } x := 0 \text{ else } x := 1, s\rangle \Downarrow \langle skip, s'\rangle$ if and only if $\langle x := 1, s\rangle \Downarrow \langle skip, s'\rangle$

by applying the corresponding if-then-else rule as follows:

$$\dfrac{\dfrac{\dfrac{}{\langle 2, s\rangle \Downarrow \langle 2, s\rangle} \quad \dfrac{}{\langle 0, s\rangle \Downarrow \langle 0, s\rangle}}{\langle 2 < 0, s\rangle \Downarrow \langle False, s\rangle} \quad \dfrac{\dfrac{}{\langle 1, s\rangle \Downarrow \langle 1, s\rangle}}{\langle x := 1, s\rangle \Downarrow \langle skip, s[x \mapsto 1]\rangle}}{\langle \text{if } 2 < 0 \text{ then } x := 0 \text{ else } x := 1, s\rangle \Downarrow \langle skip, s[x \mapsto 1]\rangle}$$

and the assignment rule:

$$\dfrac{\dfrac{}{\langle 1, s\rangle \Downarrow \langle 1, s\rangle}}{\langle x := 1, s\rangle \Downarrow \langle skip, s[x \mapsto 1]\rangle}$$

2. The semantics changes, since now the if-then-else command behaves like an if-then, ignoring the else branch.

   The value of x after the execution of the program will be the value that was in the memory before starting the program.

Exercise 4:
Recall the syntax of integer expressions in SIMP:

$$E ::= !l \mid n \mid E \ op \ E$$

and the rules defining the semantics of assignment and sequential composition (big-step semantics):

$$\dfrac{\langle E, s\rangle \Downarrow \langle n, s'\rangle}{\langle l := E, s\rangle \Downarrow \langle skip, s'[l \mapsto n]\rangle}\ (:=) \qquad \dfrac{\langle C_1, s\rangle \Downarrow \langle skip, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle skip, s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle skip, s''\rangle}\ (\text{seq})$$

1. Show that, for any integer expression E, the SIMP program

   $$x := 1; y := E$$

   is equivalent to (that is, has the same semantics as) the program

   $$x := 1; y := E'$$

   if E' is the expression E where we replaced all the occurrences of $!x$ by 1.

   Using the big-step semantics of SIMP, we can show by induction on E that both programs reach the same state s' when they are executed in a state s. More precisely, we prove that:

   $\langle x := 1; y := E, s\rangle \Downarrow \langle skip, s'\rangle$ if and only if $\langle x := 1; y := E', s\rangle \Downarrow \langle skip, s'\rangle$

   There are two base cases: If E is a number then E' is the same as E and the two programs are identical. If E is $!l$, and l is not x, again the two programs are identical. When l is x, in the first program we have to execute $y :=!x$ in a memory $s[x \mapsto 1]$, which means we put in y the value 1. In the other program we have to execute $y := 1$ which has the same effect.

   Inductive step: If E is $E_1 op E_2$ the property holds directly by the induction hypotheses for E_1 and E_2.

2. Explain why the program $x := 1; C$ (where C may be any SIMP command) is not necessarily equivalent to the program $x := 1; C'$ where C' is the same as C except that all occurrences of $!x$ have been replaced by 1. Give an example of a command C for which the results are different.

   Although for expressions we can make the replacement, for commands we cannot since a new assignment might change the value of x. For instance, if C is: $x := 2; y :=!x$ then replacing $!x$ by 1 is not correct (the program will produce a different result).

Exercise 5:
We add to the language SIMP a new command $C_1||C_2$. More precisely, we extend the abstract syntax of SIMP, adding the following rule to the grammar:

$$C ::= C_1 \,||\, C_2$$

The small-step semantics of the operator $||$ is given by the axioms and rules:

$$\frac{}{\langle skip||skip, s\rangle \to \langle skip, s\rangle}$$

$$\frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1||C_2, s\rangle \to \langle C_1'||C_2, s'\rangle} \qquad \frac{\langle C_2, s\rangle \to \langle C_2', s'\rangle}{\langle C_1||C_2, s\rangle \to \langle C_1||C_2', s'\rangle}$$

1. Describe in your own words the behaviour of a program of the form $C_1||C_2$ (indicate the intended meaning of the operator $||$).

   The rules show that atomic execution steps of C_1 and C_2 can be interleaved. The intended behaviour is that C_1 and C_2 will be executed in parallel until they both terminate.

2. What are the possible results for the program

   $$(x := 1) \,||\, (if\ x = 0\ then\ y := !y + 1\ else\ skip)$$

   assuming the values of x and y in the memory, when the program starts, are 0 and 1 respectively.

   If we choose to evaluate first the assignment, then the result is

   $$x = 1, y = 1$$

   If the conditional is evaluated first, then we obtain

   $$x = 1, y = 2$$