CS2PLD: Programming Language Design Paradigms
Maribel Fernández

Exam May 2014

Question 1:

1. Describe informally the role of selection constructs in imperative programming languages. Give two examples of selection constructs available in modern imperative languages.

   —

   Selection constructs in imperative languages are control structures, used to choose between alternative flows of control. Two well-known examples are the if-then-else construct (a two-way selector) and the switch-case construct (a multiple selector).

   Selection constructs were discussed in the lectures and tutorials.

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

$$\begin{array}{lcl} C & ::= & skip \mid l := E \mid C;C \mid if\ B\ then\ C\ else\ C \mid while\ B\ do\ C \\ E & ::= & !l \mid n \mid E\ op\ E \\ op & ::= & + \mid - \mid * \mid / \\ B & ::= & True \mid False \mid E\ bop\ E \mid \neg B \mid B \wedge B \\ bop & ::= & > \mid < \mid = \end{array}$$

   (a) Draw the abstract syntax tree of the program:

$$while\ !x < 0\ do\ (x := !x + 1;\ y := !y - 1)$$

   —

   Here you need to draw a tree, the root should have a label denoting a while command, and it should have subtrees for the condition and the body of the loop. Similar exercises were discussed in the tutorials.

   (b) We add to the language SIMP a new arithmetic operator, with abstract syntax defined by the grammar rule:

$$E ::= \ **l$$

   where l represents a memory location, as usual. Its small-step semantics is defined by:

$$\frac{}{\langle **l, s\rangle \rightarrow \langle n * n, s[l \mapsto n * n]\rangle} \quad \text{if } s(l) = n$$

   Explain with your own words the behaviour of this operator.

   —

   The operator ** takes one argument, a memory location l. When an expression $**l$ is evaluated in a memory state s, the result is $n * n$, where n is the value stored at location l in s. Additionally, the evaluation of $**l$ causes a change in the memory state: the new value stored in s at location l is $n * n$.

(c) In the language resulting from the extension of SIMP with the operator $**$, the $+$ and $*$ operators (representing addition and multiplication) are not commutative. This means that programmers can write expressions of the form $E_1 + E_2$ such that the semantics of $E_1 + E_2$ and $E_2 + E_1$ are different.

Write an expression of the form $E_1 + E_2$ in the extended version of SIMP, such that the value of $E_1 + E_2$ computed in a given memory s is different from the value of $E_2 + E_1$ in the same memory s.

—

There are many possible examples. For instance, the expression $**l+!l$, in a memory state where l contains the value 2, produces the value 8 if evaluated from left to right and the value 6 if evaluated from right to left.

Question 2:

1. We add to the language SIMP, studied in the course, a new class of expressions that will be used to represent *lists*. The concrete syntax for lists uses brackets and commas; it is defined by the grammar below, where "[", "]", and "," are terminals and E is defined in the grammar for SIMP given in Question 1b.

$$\begin{array}{lcl} \mathit{List} & ::= & [\,] \mid [E\ \mathit{Rest} \\ \mathit{Rest} & ::= & ,\ E\ \mathit{Rest} \mid\ ] \end{array}$$

As the grammar above indicates, the elements of the lists are arithmetic expressions. For example, the list containing the numbers 1, 2 and 3 is written $[1, 2, 3]$. We can also write $[0 + 1, 2, 3 * 1]$, etc.

The *abstract syntax* of lists is defined by the grammar:

$$L ::= \texttt{nil} \mid \texttt{list}(E, L)$$

where `nil` denotes an empty list and $\texttt{list}(E, L)$ represents a list where the first element is the expression E and the rest of the list is given by L.

(a) Give the abstract syntax representation for the expression $[1, 2, 3]$ (that is, write the representation, in abstract syntax, of the list containing 1, 2, and 3).

—

The list containing the numbers 1, 2, and 3 is represented by the abstract syntax
`list`(1, `list`(2, `list`(3,`nil`))).

(b) Give the principle of structural induction that should be used to prove that some property $P(L)$ holds for all lists L in this extension of SIMP.

—

Similar expressions were defined in the lectures and tutorials, and a principle of induction for integer expressions and for Boolean expressions in SIMP was given. Structural induction on lists was also discussed. The principle of induction for this kind of lists has one base case and one inductive step. It is as follows:

Base case:

$P(\texttt{nil})$

Induction step:

$P(L) \Rightarrow P(\texttt{list}(E, L))$ for any E and L

2. To evaluate lists, the following axiom and rule have been added to the set of axioms and rules defining the big-step semantics of SIMP.

$$\frac{}{\langle nil, s\rangle \Downarrow \langle nil, s\rangle}$$

$$\frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle \quad \langle L, s'\rangle \Downarrow \langle v, s''\rangle}{\langle list(E, L), s\rangle \Downarrow \langle list(n, v), s''\rangle}$$

Explain, in plain English, the process of evaluation of lists in this extension of SIMP. In particular, indicate whether the evaluation starts at the front or at the back of the list.

—

The rules indicate that the empty list is a value, since it is not evaluated any further, and if a list is not empty, that is, it is represented by an expression $list(E, L)$, then the result is a list of numbers starting with the result of the evaluation of E followed by the value of L. The evaluation of the list starts at the front because E is evaluated in s, but since SIMP does not have side-effects in integer expressions E, this choice does not affect the result.

3. In addition to the abstract syntax for lists, we include in the extended version of SIMP an operator @ to perform concatenation of lists. For instance, $[1, 2, 3]@[4, 5]$ produces the list $[1, 2, 3, 4, 5]$. Give a formal definition of the semantics of this operator (you can give transition rules for an abstract machine, or rules to extend the small-step semantics or the big-step semantics of SIMP).

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

$$\frac{\langle L_1, s\rangle \Downarrow \langle nil, s'\rangle \quad \langle L_2, s'\rangle \Downarrow \langle v, s''\rangle}{\langle L_1 @ L_2, s\rangle \Downarrow \langle v, s''\rangle}$$

$$\frac{\langle L_1, s\rangle \Downarrow \langle list(v_1, L), s'\rangle \quad \langle L @ L_2, s'\rangle \Downarrow \langle v_2, s''\rangle}{\langle L_1 @ L_2, s\rangle \Downarrow \langle list(v_1, v_2), s''\rangle}$$

Question 3

1. Briefly describe the main differences between declarative and imperative programming languages.

—

Imperative programming is based on change of state, programs contain the instructions which should be followed to obtain the desired result. Imperative programs describe how the computation should be performed. Declarative programming does not have a primitive notion of state. The main declarative paradigms are functional and logic. Declarative programs specify what the computation should achieve. In functional programming, programs are expressions consisting of functions that are evaluated in a context which defines those functions. In logic programming, goals are solved with respect to a set of clauses that describes the problem.

2. Define and compare with respect to efficiency the call-by-value and call-by-name evaluation strategies for SFUN.

—

Both strategies were discussed in lectures and tutorials. The answer should mention that call by value evaluates arguments first, and call by name evaluates arguments only if needed. Call by value is more efficient if arguments are used more than once.

3. Consider the following definition of the function `iter`:

```
iter p f x = if (p x) then x else (iter p f (f x))
```

(a) Give a polymorphic type for this function.

—

To type the function we need to compute a type for each argument and a type for the result. The full type is:

`iter::` $(\alpha \to Bool) \to (\alpha \to \alpha) \to \alpha \to \alpha$

Similar exercises were discussed in the tutorials.

(b) Assume the function `is-prime` checks whether a given number is prime, that is, the expression

`is-prime n`

is true if `n` is prime and false otherwise.

The function `inc` is defined as follows.

```
inc x = x + 1
```

The expression `iter is-prime inc` denotes a function.
Explain in your own words the behaviour of the function
`iter is-prime inc`.

—

The expression `iter is-prime inc n` is evaluated by repeatedly increasing the number n until it becomes a prime number. The function `iter is-prime inc` takes a number and returns either the number (if it is prime) or the next number that is prime.

Question 4:

1. The *unification algorithm* checks whether there is a solution for a set of equations between terms. What is the purpose of the *occur-check* in the unification algorithm?

   —

   The occur-check is a test performed when a variable X is unified with a term t. If X occurs in t, the unification algorithm fails, to avoid self-referential bindings (such as $X \mapsto f(X)$).

   The unification algorithm was presented in the lectures and discussed in the tutorial.

2. Consider the following logic program:

   ```
   friend(max, josh).
   friend(max, luke).
   friend(josh, mark).
   ```

   and the goal:

   ```
   :- friend(max,U).
   ```

   Describe the answers that Prolog will produce for this goal.

   —

   The answers are U = josh and U = luke. They are obtained by unification of the goal with the first two facts in the program.

3. Consider the following logic program (where we have added a clause to the program given in the previous subquestion):

   ```
   friend(max, josh).
   friend(max, luke).
   friend(josh, mark).
   friend(X,Y) :- friend(X,Z), friend(Z,Y).
   ```

   and the goal:

   ```
   :- friend(max,U).
   ```

   Draw the SLD-resolution tree for this goal, using the computation rule that always selects the leftmost literal as the one resolved upon.

   Show all the answers that Prolog finds for this goal.

   —

   Here you should draw a diagram of the SLD-tree. The SLD tree for this goal has three branches. The leftmost branch produces the answer U = josh. The next branch produces U = luke. Finally, the third branch computes the resolvent

   ```
   friend(max,Z), friend(Z,U)
   ```

   which in turn produces the answer U = mark.

4. Consider the following logic program:

```
friend(X,Y) :- friend(X,Z), friend(Z,Y).
friend(max, josh).
friend(max, luke).
friend(josh, mark).
```

and the goal:

```
:- friend(max,U).
```

Explain the reasons why Prolog does not find any answers for this goal.

—

The SLD tree has a leftmost branch which is infinite. Since Prolog's strategy to build the SLD tree will first try to build the left most branch, Prolog cannot find the answers for this goal, which are in other branches of the SLD tree.