Programming Languages, Fall 2008

Assignment 3: SPROL

Contents

List of Figures

1 Aims of this assignment

The aim of this assignment is to become more familiar with the scoping and parameter passing methods used in imperative programming languages, but also to train reading descriptions of programming languages and implementations provided by someone else. For this, you are given an almost complete implementation of a simple programming language SPROL. What is missing are the choice and implementation of

- a scoping model,
- a parameter passing model, and
- a primitive data type for which a unary operator '~' and a binary operator '.' must be provided.

By making these choices and completing the implementation of the SPROL interpreter (written in Java) accordingly, one obtains a primitive, but nevertheless working programming language. You are asked to make *two such implementations*. One of the implementations should use static scoping and the other should use dynamic scoping. Furthermore, one of the implementations should implement pass-by-value, and the other pass-by-reference (you may choose yourself which combinations you implement). Finally, the two implementations should use two different primitive datatypes. Produce two different directories `lab3a` and `lab3b`, one for each implementation.

In addition, your choices and the way in which these were implemented shall be described in a short report. The report should discuss all interesting and important or noteworthy aspects of your work, but omit routine details. In particular, you should point out the differences between the two implementations. You should also discuss test programs which are chosen carefully enough to illustrate how the implemented mechanisms work.

> There are a number of restrictions on SPROL (mentioned below) which are intended to make the implementor's life (= your life) easier. You are, of course, highly welcome to drop or weaken some of these restrictions if you think it makes things more interesting!

In the following section, the programming language SPROL is described in detail. Afterwards, the already existing parts of the implementation (which in particular includes a parser) are described. The implementation of these Java classes should be completed in an appropriate manner (the classes have been designed in such a way that the assignment can be completed without

developing any further classes). Proposed steps in order to arrive at a proper solution are listed and explained in the last section of this specification.

2 The language SPROL

SPROL (Simple PROgramming Language) is an imperative programming language which provides the following constructs and concepts:

- nested procedure definitions including parameters and local variables (nested means that procedures can themselves contain local procedure definitions which are not visible outside);
- access to nonlocal variables;
- expressions of a single primitive type, in which constants, variables, and the operations '~' (unary) and '.' (binary) may occur;
- assignment statements of the form `X :=` E, where `X` is a variable and E is an expression;
- procedure calls of the form

  `P(`E_1`,...,`E_n`) if U <> V`

  where `U`, `V` are variables and the if-part is optional; if it is present, the procedure call is executed only if the values of `U` and `V` differ.

  > *Restriction:* The arguments $E_1, \ldots, E_n$ of a procedure call must always be variables.

Identifiers are alphanumeric strings starting with an uppercase letter. Statements are separated by semicolons, identifier lists by commas. The following keywords are used:

program introduces the main program,

proc introduces a procedure definition,

var precedes the list of local variables (if present),

begin $\cdots$ **end** enclose the block of statements in a procedure definition,

if is used for conditional procedure calls (see above).

Comments have the form "`//` ⟨*text*⟩" (as in C, C++, Java).

Here is a sample SPROL program. It is a program which I used to test my different SPROL versions. The constants in expressions have been replaced

with the placeholder ⟨*const*⟩ in order to hide the primitive type I used.[1]

```
program Test
  var X, Y, Z
  proc Sub1(A, B, C)
    var X
    proc Sub2()
      var X
    begin
      X := ⟨const⟩;
      Sub3()
    end
    proc Sub3()
    begin
      X := (⟨const⟩ .  ((~X) . ⟨const⟩));
      A := (⟨const⟩ .  A);
      Sub3() if A <> B
    end
  begin
    X := ⟨const⟩;
    Sub2();
    Z := ⟨const⟩;
    C := X;
    Z := (Z . (⟨const⟩ .  C))
  end
begin
  X := ⟨const⟩;
  Y := ⟨const⟩;
  Z := ⟨const⟩;
  Sub1(X,Y,Z)
end
```

The main program declares three variables `X`, `Y`, `Z` and one procedure `Sub1`. `Sub1` declares a local variable `X` (thus, the `X` in `Test` is not visible in `Sub1`) and two local procedures `Sub2`, `Sub3`. `Sub1` has formal parameters `A`, `B`, `C`. Note that `Sub2` calls `Sub3`, which is not local to `Sub2`. You may try to figure out how the program behaves under different scoping and parameter passing models.

[1] information hiding of a different sort `:-)`

For simplicity, SPROL programs do not take any input. In order to test a program, you should assign the desired "input" values to its global variables using assignment statements in the program itself (as is done above with `X` and `Y` before `Sub1` is called in `Main`). In order to test the program for different input values, you have to change these expressions accordingly and re-run the program. (Again: Please feel free to extend the syntax and semantics of SPROL in some way if you want to get rid of this restriction and have an idea how to accomplish this. For example, a third kind of statement `read(X)` which reads a value from the terminal could be implemented and added to the language rather easily.)

The precise syntax of SPROL is given by the following productions in EBNF (*Extended Backus-Naur-Form*, you remember?) notation:

⟨***program***⟩ → **program** ⟨***ID***⟩ ⟨***body***⟩

> A program definition consists of the keyword `program`, followed by an identifier (the name of the program), followed by its body. Identifiers belong to the lexical syntax of SPROL and are, as mentioned above, strings consisting of letters and digits, starting with an uppercase letter.

⟨***procedure***⟩ → **proc** ⟨***ID***⟩ **(** ⟨***identifier_list***⟩ **)** ⟨***body***⟩

> A procedure definition consists of the keyword `proc`, followed by the procedure's name, the list of formal parameters in parentheses, and the body. Note that a simple list of identifiers is sufficient since there is only one data type and only one parameter passing model.

⟨***identifier_list***⟩ → [⟨***ID***⟩ **{ ,** ⟨***ID***⟩ **}**]

> An identifier list consists of an arbitrary number (including zero) of identifiers with commas in between (recall that [···] denotes something optional and {···} denotes an arbitrary number of repetitions).

⟨***body***⟩ → [**var** ⟨***identifier_list***⟩]
[⟨***subproc_decls***⟩]
begin
[⟨***statement_seq***⟩]
end

> The body of a program or procedure consists of a declaration of local variables, the definition of local procedures, and a statement sequence which is enclosed in begin···end. Each of the three nonterminal parts is optional.

⟨***subproc_decls***⟩ → ⟨***procedure***⟩ **{** ⟨***procedure***⟩ **}**

The definition of local procedures is a sequence of procedure definitions (without commas, semicolons, or the like in between).

⟨***statement_seq***⟩ → ⟨***statement***⟩ **{ ;** ⟨***statement***⟩ **}**

A statement sequence is a sequence of statements with semicolons in between (but not after the last one!).

⟨***statement***⟩ → ⟨***assignment***⟩ | ⟨***call***⟩

A statement is either an assignment or a procedure call (recall that '|' separates alternatives).

⟨***assignment***⟩ → ⟨***ID***⟩ **:=** ⟨***expr***⟩

A assignment consists of an identifier (the variable name) followed by the symbol := and an expression (the one whose value is to be assigned to the named variable).

⟨***call***⟩ → ⟨***ID***⟩**(** ⟨***identifier_list***⟩ **)** [**if** ⟨***ID***⟩ **<>** ⟨***ID***⟩]

A procedure call consists of an identifier (the procedure name) followed by a parenthesis containing the list of parameters (which are variable names). Optionally, a condition can be added using the keyword 'if', which is followed by two variable names with the symbol <> in between. The intended semantics is that the call is only executed if the two variables have different values (or the optional part is left out).

⟨***expr***⟩ → ⟨***const***⟩
| ⟨***ID***⟩
| **(** ~ ⟨***expr***⟩ **)**
| **(** ⟨***expr***⟩ **.** ⟨***expr***⟩ **)**

An expression is either a constant (of the type you decide to implement) or an identifier (the name of a variable), or it has one of the two forms (~ E) and (E . E'), where E, E' are expressions. Semantically, the expression (~ E) is meant to denote the application of the unary operator '~' to (the value of) E. Similarly, (E . E') denotes the application of the binary operator '.' to E and E'. As mentioned above, you can (and should) give these operators any meaning you think is appropriate.

3 Existing parts of the implementation and what they do for you

3.1 The parser Sprol.jj

The most important part of the implementation is the SPROL parser. This parser is defined in a file called `Sprol.jj`, which mainly describes the context-free syntax of SPROL by rules similar to those above. The file serves as input to a rather comfortable parser generator called JavaCC (*Java Compiler Compiler*). If the parser generator is applied to this file ("`javacc Sprol.jj`") it generates the source code for a few Java classes, the main one being the actual parser `Sprol.java`.

After compilation ("`javac Sprol.java`"), the resulting parser can be used to parse and execute a SPROL program. You simply call "`java Sprol` ⟨*file*⟩, where ⟨*file*⟩ is the file name of the SPROL program. As a result, the following happens.

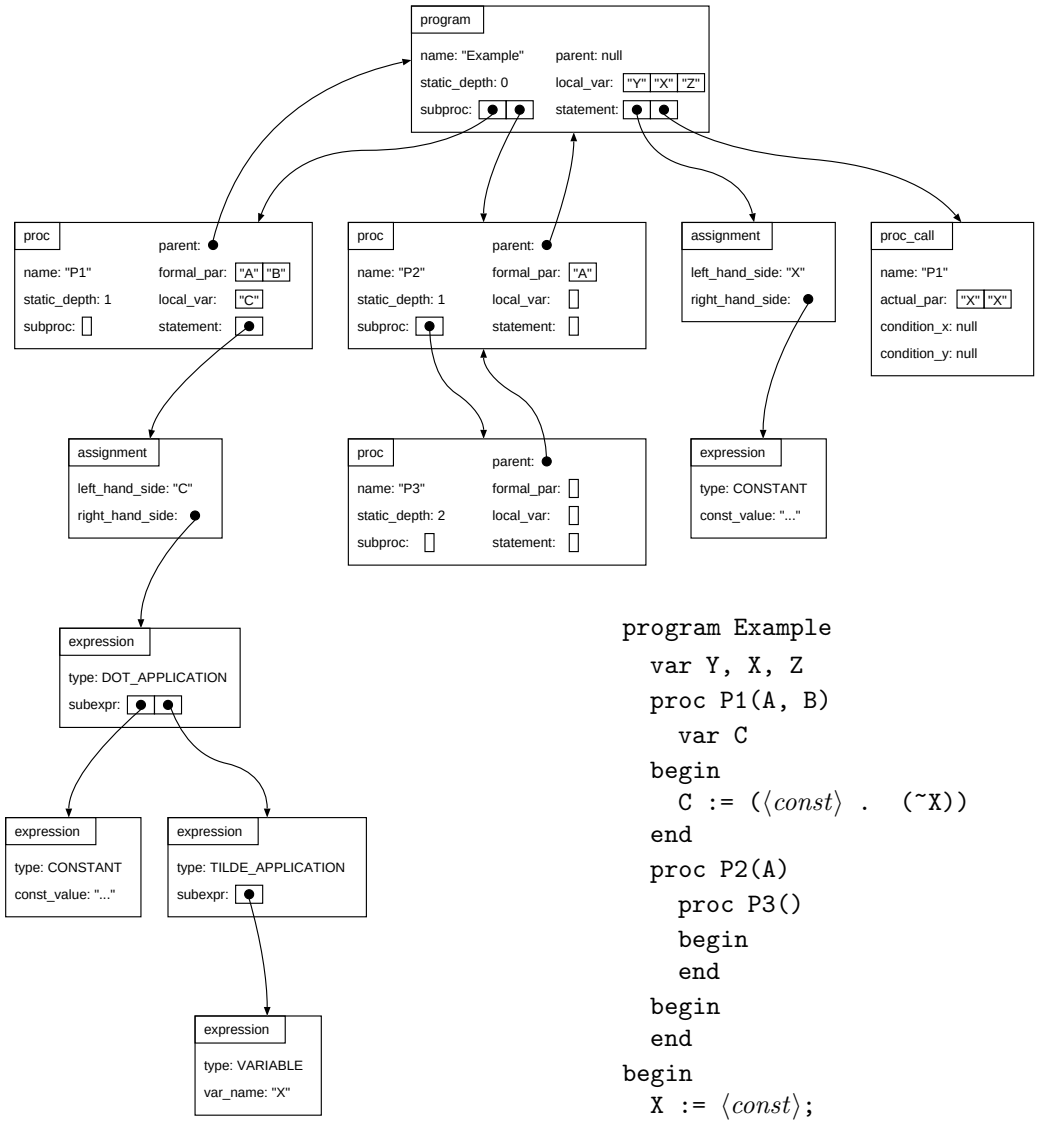
(1) The SPROL program in ⟨*file*⟩ is parsed. If no syntax errors are encountered, the parser produces an internal representation of the SPROL program, which is an object of the class `Program`.

(2) The class `Program`, which is described in more detail below, contains the method `execute()`. This method is now called by the parser in order to interpret the SPROL program.

(3) When (and if) `execute()` terminates, it prints the final values of the program variables on the standard output.

The parser does not check any aspects of static semantics. Thus, the use of undeclared variables, calls of undefined procedures, etc. are not recognized as errors until runtime. You should make sure that your interpreter reacts in an appropriate way if such situations are encountered.

The internal representation of a parsed program (see (1) above) consists of objects which correspond roughly to the SPROL language constructs. For a small sample program this structure is depicted schematically in Figure 1. Every box is an object. The class name is indicated in the upper left corner. Dots with outgoing arrows symbolize references to other objects.




```
program Example
  var Y, X, Z
  proc P1(A, B)
    var C
  begin
    C := (⟨const⟩ .  (~X))
  end
  proc P2(A)
    proc P3()
    begin
    end
  begin
  end
begin
  X := ⟨const⟩;
  P1(X,X)
end
```

Figure 1: Internal representation of a SPROL program

The only thing that you have to do regarding `Sprol.jj` is to adapt the parsing of expressions to the type you want to implement, so that the parser accepts expressions containing constants of that type. For this, you have to change the line in the definition of expressions which deals with constants, and which presently reads `t = <CONST>`. The nonterminal `<CONST>` should be replaced by another one. In the lexical part of the parser, three appropriate nonterminals `<STRING>`, `<INTEGER>`, and `<DECIMAL>`, are already defined. You may replace the `<CONST>` in the definition of expressions with one of these, or choose another type. In the latter case, you also have to add an appropriate definition to the lexical part, using a regular expression that defines the desired lexemes (see the JavaCC documentation or ask one of us if you have problems with the JavaCC syntax of regular expressions).

Even though it is not necessary for the assignment, it may be interesting to have a closer look at `Sprol.jj`. The way in which the parsing of SPROL works is quite intuitive and proceeds along the context-free rules of the grammar. For each nonterminal, JavaCC generates a method which performs the parsing. The input file `Sprol.jj` contains pieces of Java code at appropriate places in order to construct and initialize Java objects while the parsing process proceeds. Notice also the additional method `main(String[])` defined in the beginning of `Sprol.jj`. It mainly accomplishes what was described above: creation of a parser object which reads from the given file, invocation of its parsing method `program()` (which is one of the methods generated by JavaCC, based on the rules for this nonterminal further down in the file), and invocation of the `execute()` method of the resulting instance.

3.2 The class `Program`

Nothing needs to be changed here!

This class represents the parsed program. Since a program is merely a procedure without parameters, it simply extends the class `Proc`. It overwrites the method `execute()` inherited from `Proc` in order to change it in two respects. Before executing `Proc.execute()` it puts a first activation record onto the runtime stack. When `Proc.execute()` has terminated the values of local variables are looked up and printed on the standard output.

3.3 The class `Proc`

This class represents a procedure. Its instance variables contain the information gathered by the parser. You should use this information, but there is no need to change the values of these variables:

String name is the name of the procedure;

Proc parent is the static parent of the procedure;

int static_depth is the static depth of the procedure;

Vector subproc contains the subprocedures of the procedure, all of them being instances of the class `Proc`;

Vector local_var contains the names of the locally declared variables, all of them being instances of the class `String`;

Vector formal_par contains the names of the formal parameters (in the correct order, starting with the first one at index 0), all of them being instances of the class `String`;

Vector statement contains the sequence of statements of the procedure (again the first one at index 0), each of them being either an instance of the class `Assignment` or of the class `ProcCall`.

The constructor of `Proc` is of no importance—it is only used by the parser. The method `execute()` is a loop which executes the statements of this procedure one after another. For each statement, it determines whether it is an assignment or a procedure call. Accordingly, it invokes `execAssignment(···)` or `execProcCall(···)`, which implement the actual execution of the given statement. These methods take the statement to be executed as a parameter.

> Since the evaluation of expressions and the execution of procedure calls depend on the data type as well as the scope and parameter passing models, the implementation of `execAssignment(···)` and `execProcCall(···)` is your task! Furthermore, you will have to implement the auxiliary method `initRecord(···)`.

With respect to `execAssignment(···)`, two things need to be done:

(1) Evaluate the expression stored in `a.right_hand_side`, where `a` is the instance of the class `assignment` representing the assignment statement to be executed. Note that the result of this evaluation depends on the values of variables on the stack!

To evaluate the expression, `a.right_hand_side` (which is an instance of the class `Expression`) contains the method `eval()`.

(2) On the runtime stack, set the contents of the memory cell to which `a.left_hand_side` refers to the evaluation result.

The implementation of `execProcCall(···)` must do this:

(1) If there is a condition attached to the procedure call, look up the values of the two variables on the stack. If they are equal, skip the remaining steps. The names of the variables in the condition are stored in `c.condition_x` and `c.condition_y` by the parser (where `c` is the instance of `ProcCall` which represents the call statement). If there is no condition, then `c.condition_x` and `c.condition_y` are `null`.

(2) Search for the instance of `Proc` within the static scope of the present procedure whose name equals the name of the procedure to be called (which is stored in `c.name`). For this, you first have to search the vector of direct subprocedures of the present procedure (i.e., in the vector `subproc`). If you do not find it there, search the direct subprocedures of the static parent, and so on. (If the call refers to a procedure which does not exist in the static scope of the parent you will finally encounter the static parent `null`. Make sure that you exit the program gracefully with an appropriate error message in this case.)

(3) When the procedure `P` to be called has been found, set up its activation record using `P.initRecord(Vector, int)`. The first argument should be the vector of actual parameters of the call (stored in `c.actual_par`). The method `initRecord(···)` should use this in order to bind the formal parameters to the actual ones given in this vector. The second parameter is the static depth of the *calling* procedure. This is needed by `initRecord(···)` in order to set up the static link correctly.

(4) Call `P.execute()` in order to run `P`.

(5) Remove the activation record from the stack when `P.execute()` has returned. If your version of the language contains pass-by-result parameters, the corresponding parameter passing actions need to be performed here, too.

Finally, here is what `initRecord(actual, calling_depth)` should do:

(1) For each actual parameter (whose names are stored in `actual`) determine the corresponding memory cell from the stack. For this, the class `Stack` contains the method `getCellOf(String)`.

(2) Create and initialize a new activation record using the method `stack.addRecord(int)`. To set up the correct static link you need to pass the nesting depth as a parameter. (Recall how the nesting depth is determined from the static depth of the calling procedure and the static depth of this procedure, which is the called one.)

(3) For each formal parameter name (whose names are stored in the vector `formal_par`), create a variable on the stack and bind it to the corresponding actual parameter given in the vector `actual`. Depending on the parameter passing model you have chosen, this should either bind the variable to the cell determined in step (1) or to a newly created cell which gets initialized with the contents of the cell determined in step (1).

(4) For each local variable name (stored in the vector `local_var`), add a corresponding variable to the activation record.

3.4 The class `Assignment`

Nothing needs to be changed here!

This class represents an assignment. It contains two instance variables called `left_hand_side` and `right_hand_side`. The first is the variable name to the left of '`:=`' (a `String`), and the second is the expression on the right-hand side of the assignment (an instance of the class `Expression` discussed below). The constructor of this class is only used by the parser.

3.5 The class `ProcCall`

Nothing needs to be changed here!

This class represents a procedure call. Again, only its instance variables are of interest. The first is `name`, the name of the called procedure, which is a `String`. The second is `actual_par`, containing the actual parameters. This is simply a vector of strings—recall that all actual parameters are required to be variables!

Finally, there are two strings `condition_x`, `condition_y`. If the represented procedure call is subject to a condition `if X <> Y`, then these strings are the names of the two variables, otherwise they are `null`.

3.6 The class `Expression`

This class represents SPROL expressions, using the following instance variables.

int type determines whether the expression is a constant, variable, of the form ($\tilde{}$ E), or of the form ($E.E'$). Accordingly, its value may be `CONSTANT`, `VARIABLE`, `TILDE_APPLICATION`, or `DOT_APPLICATION` (which are constants defined in `Expression.java` with the obvious meaning);

String const_value contains the string representation of the expression if it is a constant (i.e., if `type` equals `CONSTANT`)

String var_name contains the name of the variable if the expression is a variable (i.e., if `type` equals `VARIABLE`)

Expression[] subexpr is the array of subexpressions in those cases where `type` equals `TILDE_APPLICATION` or `type` equals `DOT_APPLICATION` (in the first case of length 1, in the second of length 2).

There is one usual constructor and there are four static ones (for the four types of expressions), which are of interest only for the parser.

In addition, there is the method `eval()` used during the execution of assignment statements. Since it depends on the chosen data type, only a skeleton is provided. You have to fill in the details. Here are some hints:

- Recall that `constant_value` contains the string representation of a constant (exactly as the parser reads it). Thus, some conversion into a more suitable representation is probably useful in the first case (for example, using `Integer.valueOf(String)` if you choose integers as your data type).
- To evaluate a variable, you have to fetch the contents of its memory cell from the stack.
- The two remaining cases seem to be a situation where the use of recursion is a good idea `;-)`.

3.7 The class `Stack`

This class implements the runtime stack as a vector of activation records. Activation records are represented by the protected local class `ActRecord`. An activation record is a vector of variables. In addition, an activation record

contains a static link (represented as an `int`—the index of the activation record to which the link points). A variable is represented by the private local class `Variable`. It consists of a name (instance variable `name`) and a memory cell (instance variable `mem_cell`). For simplicity, memory cells have been realized as `Object` arrays of length 1. The object stored in the array entry is the contents of the memory cell.

Reflecting the fact that there should be only one runtime stack, all methods in `Stack` are static and act on a single private `Stack` instance called `st`. Here is the description of the methods in `Stack`:

addRecord(int) creates a new activation record, puts it on the top of the stack, thus making it active. The parameter is the nesting depth of the calling procedure within the called procedure's parent. It is used in order to set up the static link of the record correctly.

> This method should be used in `proc.initRecord(···)` in order to put a new activation record on the stack when a procedure is called.

deleteRecord() removes the topmost activation record from the stack.

> You will probably have to use this method at the end of your implementation of `proc.execProcCall(···)`.

addVariable(String, Object[]) adds a new variable to the currently active (= topmost) activation record. The first parameter is the variable name. The second should be an array of length 1 or `null`. In the first case it will be bound to the variable as its memory cell. In the second case a fresh memory cell with a `null` contents will be allocated and bound to the variable.

getCellOf(String) returns the memory cell to which the given variable name refers.

> This need to be filled in by you. You will probably have to use this method in your implementation of `proc.execAssignment(···)` and/or `proc.execProcCall(···)`. Note that the implementation depends on the scope model you use since you do not only have to search the current activation record but also its dynamic respectively static ancestors if the variable is not local.

getCellContentsOf(String) returns the contents of the memory cell which

is bound to the variable. This is just a convenience method which uses `getCellOf(···)`) in order to find the memory cell whose contents is asked for. The method indicates a runtime error and terminates the program if the cell is uninitialized (i.e. if its contents is `null`).

> You may use this method in your implementation of the method `proc.initRecord(···)` if you implement pass-by-value parameters, and in your implementation of `Expression.eval(···)`.

4 Suggested steps

The following steps should lead to a successful completion of the implementation:

1. Decide which scope model, parameter passing model, and primitive data type (including the two operations) you want to use.
2. Adapt the parser to your primitive data type as explained in the description of `Sprol.jj`.
3. Complete the implementation of `Stack.java`.
4. Implement the evaluation method of the class `Expression`.
5. Implement the missing parts in the methods of `Proc`.
6. Run some test programs which are simple, but nevertheless cover all relevant cases, in order to be reasonably sure that your interpreter works correctly.

If your test programs are not interpreted correctly and you cannot find out where the problem lies, it may be useful to add a simple tracing mechanism to `Stack.java`, which prints the stack (or part of it) whenever something is changed.

General information

- The assignment is due 9 January, 2009, 10:00.
- The assignment is to be done in pairs.
- A complete report is expected.