Project "LISP-like language interpreter"

Armando Tacchella

Objective

Create a program interpreter for a LISP-like language containing variable definitions, input/output statements, conditional choices, and loops, limited to integer variable operations only. The interpreter must perform the following steps:

- Read a *file, called the source file below, which contains the program to be interpreted; the program syntax 'is defined by an unambiguous context-free grammar (below). • Run the program contained in the source file.*
- Ask the console if input statements enter any data and display the expression result expected from the output statements on consoles.

Input format

In the following description of context-free grammar we write the production $N \to \alpha \mid \beta$ as an abbreviation of productions

$$N \to \alpha \qquad N \to \beta \tag{1}$$

We use italics to define nonterminal symbols, and normal characters to define terminal symbols. The symbol denotes the empty string. The grammar productions are shown in Figure 1.

The *keywords* of this simple language are SET, PRINT, INPUT, IF, WHILE for statements; ADD, SUB, MUL, DIV for arithmetic operators; LT, GT, EQ for relational operators; AND, OR, NOT for Boolean operators, TRUE, and FALSE for their constants. For example, a possible input program' is as follows:

```
(BLOCK
  (INPUT n)
  (SET result 1)
  (WHILE (GT n 0)
     (BLOCK
        (MUL result n))
        (SET n (SUB n 1)))
  )
  (PRINT result))
```

There are no assumptions about the number of instructions contained in a particular line, while spaces, tabs and blank lines are not part of the syntax of the program, but should be treated as "white space" and ignored. The previous program could also be written correctly like this:

Program → *STMT Block*

stmt block → *statement* | (BLOCK *statement_list*)
statement list → statement statement list / *Statement*

statement → *variable def* |
I STMT | *cond*
stmt *STMT*
Loop

variable def → *(SET variable* id num expr) io stmt → *(PRINT num expr*) | (INPUT *variable id*)
cond stmt → (IF *bool expr stmt block stmt block*)
loop stmt → (WHILE *bool expr stmt block*)

num expr → (ADD *num expr num expr*) | (SUB *num expr num expr*)
| (MUL *num expr num expr*)
| (DIV *num expr num expr*)
| *Number*
| *variable id*

bool expr (LT *num expr num expr*)
| (GT *num expr num expr*)
| (EQ *num expr num expr*)
| (AND *bool expr bool expr*)
| (OR *bool expr bool expr*)
| (NOT *bool expr*) |
TRUE | FALSE

variable id → *alpha list*
alpha list → *alpha alpha list* | *Alpha*
Alpha → A | B | C | ... | z | A | B | C | ... | Z

number → - *posnumber* | *posnumber*
posnumber → 0 | *Sigdigit REST*
Sigdigit → 1 | ... | 9 rest → digit
rest / *digit* → 0 | *sigdigit*

Figure 1: Interpreter input grammar.

(BLOCK(INPUT n)(SET result 1)(WHILE(GT n 0)(BLOCK(SET result(MUL result n))(SET n (SUB n 1))))
(PRINT result))

that is, with the minimum number of spaces to make the program syntactically correct, or like this:

```
(BLOCK

   (INPUT n)
   (SET result 1)
   (WHILE (GT n 0)

      (BLOCK
         (MUL result n))
         (SET n (SUB n 1)))
   )

   (PRINT result)

)
```

i.e. with additional spaces and tabs that should be ignored.

The programme is not correct:

(PRINT (ADD (MUL14 10)) 25)
as MUL14 is not a correct operator name (MUL), but neither is the name of a variable (identifiers do not allow a nueric part).

It is also useful to make the following clarifications:'

- The grammar **does not allow** the definition of variables that are not numeric: SET requires a *num _expr*.
- In the case of INPUT, the constraint is imposed that the read value is an integer.
- There is a single global scope for variables.
- Variables are defined dynamically (you don't need to declare them before using them for the first time as in C/C++/Java).

As a further simplification, assume that the integers supplied at the input are always representable in 64-bit registers (the long type of Java), i.e. for any number *n used we have that −263 ≤* n < *263.*

Output Format

SET Statement.

The result corresponding to a statement

(SET *variable _id num expr*)

It must be:

- the **creation** of the variable *variable id,* if it had not already been defined previously, and
- **Assigning** the *value _ of the EXPR* num expression **to the variable** variable ID .

The following errors may occur:

- the statement is not syntactically correct,
- *variable _id* is a keyword, or
- *num expr* is not syntactically correct, or contains a semantic error (e.g., division by zero).

An error is obviously generated even if the *num expr expression _contains the same variable as variable expr _ and the latter has not already been defined previously.*

The INPUT statement.

The result corresponding to a statement

(INPUT *variable id*)

It must be:

- the **creation** of the variable *variable id,* if it had not already been defined previously, and
- Assigning **the** console-read *value _to the variable* variable **id**.

The following errors may occur:

- the instruction is not syntactically correct,
- *variable _id* is a keyword, or
- An illicit value is entered from the console (**only** positive and negative integers are allowed).

PRINT statement.

The result corresponding to a statement

(PRINT *expr number_*)

Must be **the console view of** the **value** of the *EXPR num expression*.

The following errors may occur:

- the instruction is not syntactically correct,
- *num expr* is not syntactically correct, or contains variables not previously defined with SET or INPUT, or contains a semantic error (e.g., division by zero).

IF Instruction.

The result corresponding to a statement

(IF *bool expr stmt block1 stmt block2_*)

It must be:

- running *stmtblock1* if *boolexpr* evaluates it to be true, or
- Running *STMT_Block2* in case *BOOL_EXPR* evaluates a false.

The following errors may occur:

- the statement is not syntactically correct,
- *bool expr* is not syntactically correct or contains variables not previously defined, or
- One of the two *STMT _blocks* is not syntactically correct, or, if executed, contains a semantic error (e.g., division by zero).

The WHILE statement.

The result corresponding to a statement

(WHILE *bool expr stmt block*)

It must be the execution of *stmt _block* as long as the expression bool expr *evaluates a true; the execution does not occur if* bool expr 'and false at the start. The following errors may occur:

- the statement is not syntactically correct,
- *bool expr* is not syntactically correct or contains variables not previously defined, or
- The *STMT block* is not syntactically correct, or, if executed, contains a semantic error (e.g., division by zero).

The BLOCK statement.

The result corresponding to a statement

(BLOCK *statement1 ...statementn*)

It must be the sequential execution of the different statements. The following errors may occur:

- the statement is not syntactically correct,
- For some *i,* statmenti *is not syntactically correct, contains variables not previously defined, or contains a semantic error (e.g., division by zero).*

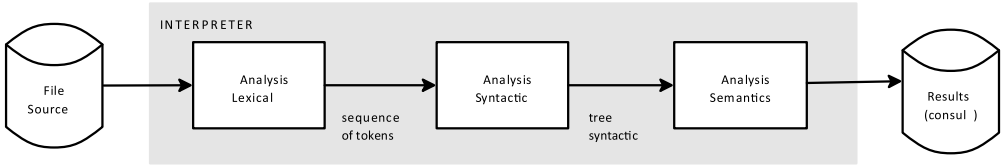

Figure 2: Functional architecture of the interpreter.

Expressions.

Numerical expressions are interpreted with the obvious semantics for values and operators: sum for ADD, subtraction for SUB, etc. In Boolean expressions, relational operators have the following semantics:

- GT stands for "greater than", so (GT *a b) 'e true if the value* of a *' is strictly greater than that of* b*;*
- Similarly, LT stands for "lesser than" and EQ stands for "equal" with its intended meaning.

The Boolean operators AND, OR, NOT also have the obvious semantics; AND and OR are **short-circuited**, i.e.

- (a *AND* b) if *a* is rated as *false and immediately rated as false (*b *is not valued);*
- (a *OR* b) if *a evaluates a true and immediately evaluates a true (* b *is not evaluated).*

For example, at the program shown above, the output would be as follows:

3628800

The errors present in the file must be treated by raising exceptions that must be handled appropriately with the use of try … catch (*i.e.*, the main must not propagate exceptions).

Design and implementation tips

To simplify the solution, it is advisable to divide the implementation into three distinct parts, used in sequence: lexical analysis, syntactic analysis (parsing and construction of the syntactic tree) and semantic analysis (evaluation). The corresponding structure of the program at the functional level is shown in Figure 2.

Lexical analysis.

It has the task of reading the source file containing the program and iterating it by providing the output sequence *of lexical elements* (*words* or *tokens*) of the program. Formally, for the grammar of source files, a lexical element corresponds to one of the following groups:

- a keyword: SET, PRINT, INPUT, IF, WHILE, ADD, SUB, MUL, DIV, GT, LT, EQ, AND, OR, NOT, TRUE, FALSE;
- an open or closed parenthesis;
- a number (defined with number rules);
- a variable (defined with *variable id rules*);

The use of tokens corresponds to using for the next step of syntactic parsing the rules shown in Figure 3 in which the tokens were indicated using bold. In case the lexical analyzer encounters an unexpected token, it returns an error message.

Program → *STMT Block*
stmt block → *statement* | **(BLOCK** *statement list* **)**
statement list → statement statement list / *Statement*

statement → *variable def* |
I STMT |
cond _ stmt
STMT Loop

variable def → (SET **variable id num expr) io stmt** → *(PRINT* num
expr **) / (INPUT** *variable _id* **)**

cond stmt → **(IF** *bool _expr stmt block stmt block* **)**
loop stmt → **(WHILE** *bool expr stmt block* **)**

num expr → **(ADD** *num expr num expr _* **) | (SUB**
num expr num expr **)**
| **(MUL** *num expr num expr* **)**
| **(DIV** *num expr num expr* **)**
| *Number*
| *variable id*

bool expr **(LT** *num _expr num expr* **)**
| **(GT** *num expr numexpr* **)**
| **(EQ** *num expr num_expr* **)**
| **(AND** *bool expr bool expr* **)**
| **(OR** *bool expr bool expr* **)**
| **(NOT** *bool expr* **)**
| **TRUE** | **FALSE**

Figure 3: Abstract grammar for syntactic analysis.

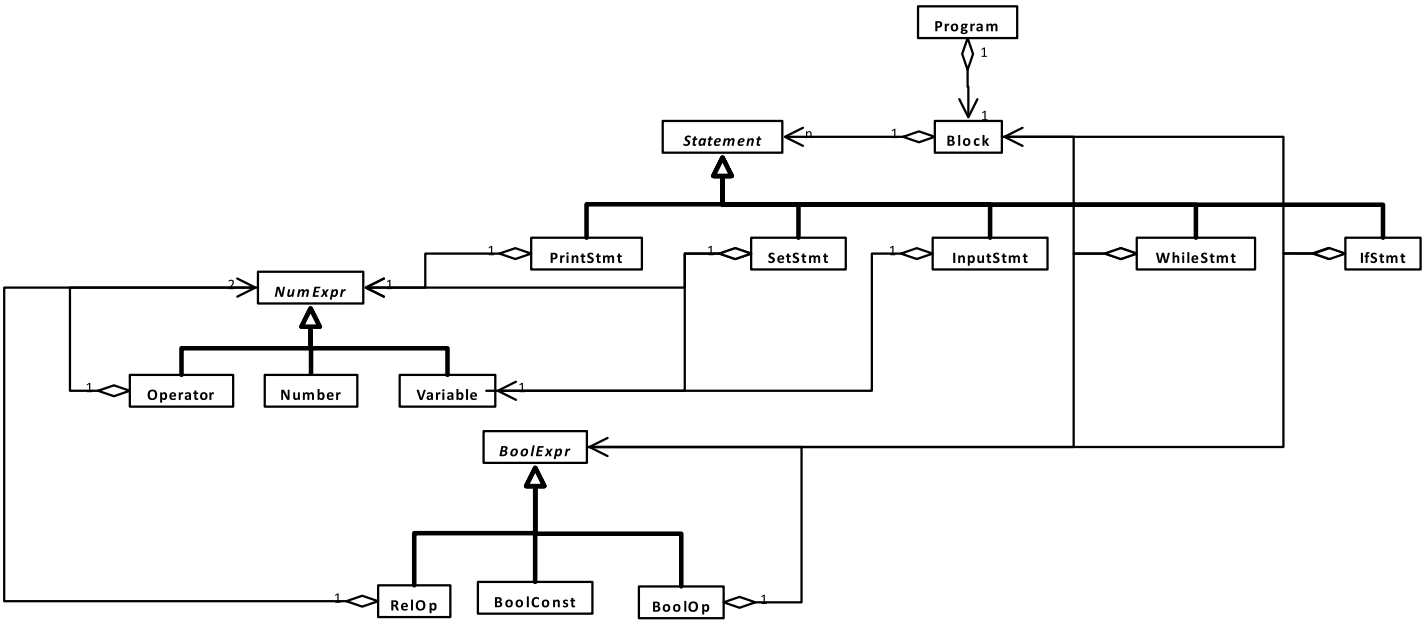

Figure 4: Class hierarchy for constructing the syntactic tree (Pattern Interpreter).

Syntactic analysis.

It has the task of reading the sequence of tokens provided by the lexical analyzer and checking the syntactic correctness of the constructs. Provides an outgoing description in the form of an input program tree (c.d. *syntactic tree*) The syntactic tree can be constructed using the "Interpreter" pattern based on the class hierarchy shown in Figure 4. In case of syntactic errors, an error message is returned at the first encounter.

Semantic analysis.

It has the task of executing/evaluating statements taking into account assignments, input/output statements, conditional choices and loops. Constructs the symbol table and adds items as variable definitions are encountered and their values are assigned. Identify semantic errors such as divisions by zero (overflow and underflow are not considered). However, it operates a visit of the syntactic tree and produces directly in output what is required by the PRINT instructions. To perform the evaluation of the expressions you can use a "Visitor" object with appropriate methods that allow the analysis of the various elements of the "Interpreter" objects.