Formal Languages and Compilers Project Report

18/06/2021

Riccardo Rigoni, Luca Taddeo, Sebastian Cavada

Language Description

The language is an **extension** of the files that were used in the laboratory of the Formal Languages and Compilers Course. It extends the concept of calculator with additional variables' types, including double, int and boolean. Furthermore, it introduces two control structures: *if-then-else* and *while loop*.

Recalling the C language, only the first statement after a control flow structure is associated with it, and the others are treated as independent statements. Therefore, multi-line blocks of code need to be enclosed into curly brackets to be associated with a flow structure.

The language allows multiple scoping, therefore, the visibility of the variables depends on the position of their declaration. Code blocks that are associated with a control flow statement are considered as sub-scopes of the main one, and are independent from the other. Consequently all such rules related to scoping will apply.

N.B. variables redeclaration is only possible if the same name belong to different scopes

Formal Grammar

```
The grammar is formally described by the tuple G = <V,N,P,S>.

V = { the set of all terminal symbols} = { BOOLEAN, INT, DOUBLE,
IDENTIFIER,ASSIGN_OP, BIGGER_THAN, SMALLER_THAN, EQUAL_TO, NOT_EQUAL_TO,
BIGGER_EQ_THAN,SMALLER_EQ_THAN, NOT, BOOL_VAL, PLUS, MINUS, MOLT, DIV, INT_VAL,
DOUBLE_VAL, O_PAR,C_PAR, IF, ELSE, WHILE, PRINT }

N = { the set of all non-terminal symbols} = { enter_sub_scope, exit_sub_scope,
decl, typename, assign, list_stmt, stmt, ctrl_stmt, cond_stmt, while_stmt,
if_stmt, else_stmt, expr_stmt, expr, a_expr, b_expr}
```

```
P = ( the set of all production rules ) = {
 list_stmt : | list_stmt stmt

 stmt  : ';' | PRINT expr ';' | ctrl_stmt | expr_stmt  ';'

 ctrl_stmt :  while_stmt | cond_stmt

 while_stmt :  WHILE expr enter_sub_scope stmt exit_sub_scope |  WHILE expr '{'
enter_sub_scope list_stmt exit_sub_scope '}'

 cond_stmt : if_stmt |  if_stmt else_stmt

  if_stmt  :  IF  expr  enter_sub_scope  stmt   exit_sub_scope  |   IF  expr  '{'
enter_sub_scope list_stmt exit_sub_scope '}'

 else_stmt : ELSE enter_sub_scope stmt exit_sub_scope | ELSE '{' enter_sub_scope
list_stmt exit_sub_scope '}'

 expr_stmt : expr | decl | assign

 expr  : O_PAR expr C_PAR | a_expr | b_expr | IDENTIFIER

 a_expr : expr PLUS expr | expr MINUS expr | expr MOLT expr | expr DIV expr |
MINUS expr | INT_VAL | FLOAT_VAL

 b_expr : expr BIGGER_THAN expr | expr BIGGER_EQ_THAN expr | expr SMALLER_THAN
expr | expr SMALLER_EQ_THAN expr | expr EQUAL_TO expr | expr NOT_EQUAL_TO expr |
NOT expr | BOOL_VAL

 decl : typename IDENTIFIER | typename IDENTIFIER ASSIGN_OP expr

 typename : BOOLEAN | INT | DOUBLE

 assign : IDENTIFIER ASSIGN_OP expr

 enter_sub_scope :

 exit_sub_scope :
}

S = { scope of the language } = { list_stmt }
```

Compiler

The implementation and functioning of the compiler will be described in the following paragraphs through its three different phases: Lexical, Syntactic and Semantic Analysis.

Lexical Analysis

To better suit the needs of the language, the global variable *yylval* was changed as follows:

```
struct parse_tree_node {
      int type;
      double value;
      char* lexeme;
};

union yylval {
       char* lexeme;
       double value;
       int type;
       struct parse_tree_node parse_tree_node_info;
       }
```

The additional struct *parser_tree_node* has been added to store the relevant information about the expression under evaluation, making both the result evaluation and the type checking possible.

Syntactic Analysis

Since the grammar stated before is not *LR(0)*, disambiguation rules have been defined to ensure correctness of the syntactic analysis.
The main ambiguity issues faced were:

1. Operators precedence issue: to solve this problem, following prioritization system was adopted
   - High priority: arithmetic operators
   - Middle priority: boolean operators
   - Low priority: assignment operator
2. If-Then-Else issue: also to solve this problem, a prioritization approach was followed. Lower priority was given to the if-clause, so that the else-branch is always evaluated first and connected to the closest if-clause

Follows the declaration of the rules:

```
%nonassoc ENTER_SUB_SCOPE EXIT_SUB_SCOPE

%nonassoc INT_VAL FLOAT_VAL BOOL_VAL IDENTIFIER BOOLEAN INT DOUBLE WHILE IF
PRINT ';' '}'

// operators precedence
%right ASSIGN_OP
%left BIGGER_THAN SMALLER_THAN EQUAL_TO NOT_EQUAL_TO BIGGER_EQ_THAN
SMALLER_EQ_THAN
%left NOT
%left MINUS PLUS
%left MOLT DIV
```

```
%left C_PAR
%right UMINUS O_PAR

// if-then-else issue
%nonassoc IFX
%nonassoc ELSE

%nonassoc LIST_STMT
```

Semantic Analysis

Semantic correctness can not be directly ensured through the grammar definition. Therefore, semantic actions have been associated with each production rule and they aim to evaluate the result of the encountered expressions during the parsing phase as well.

There are different categories of rules and they are:

- result evaluation: these rules computes the result of arithmetic and boolean expressions
- variable declaration: these rules check whether a new variable could be declared and if it is the case, they update the symbol table
- type compatibility: these rules check whether the types associated to either an operation or an assignment are compatible.
- type coercion: these rules apply implicit type coercion when the two types are not equal but compatible

Symbol Table

The symbol table is composed of 3 main objects: the table_obj, the table_node and the variable.

TABLE_OBJ

It contains a pointer to a specific list of nodes belonging to a specific scope. Every table_obj instance defines the starting point of a new subtable within the symbol table, and every subtable of the symbol table corresponds to a different variables' scope.

TABLE_NODE

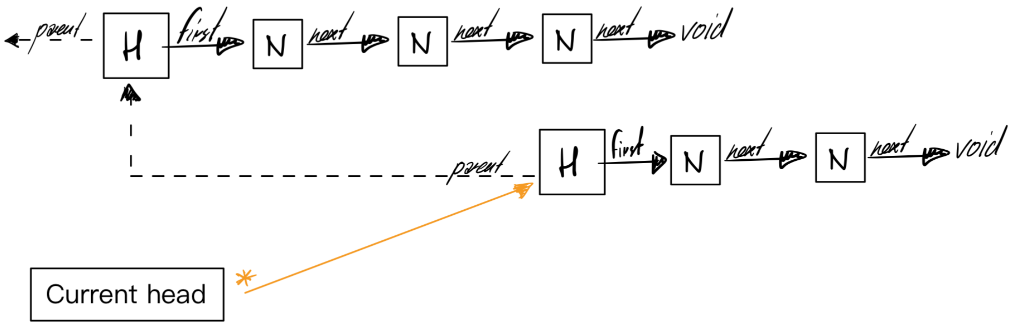
Every table_node stores information of a specific variable and contains a pointer to the next node (in fact the previously added node). If the variable is declared, but not initialized, the enclosed number object is not empty, since the parser inserts a default value for not initialized variables.

VARIABLE

The purpose of the variable struct is to store relevant data over the declared variable, so to exploit this information to evaluate expression results.

N.B. In the implemented symbol table, for simplicity purposes, all sub-tables were only backward-linked. This concept allowed lighter data structures, without compromising the functionalities for the purpose of this project.

The main idea behind the symbol table can be visualized in the following diagram:



The *current head* pointer keeps track of the current TABLE_OBJ to make insertion of new variables easier and faster. Whenever there is the need to enter into or exit from a sub-table, the current header pointer changes. In the first case, when a new sub-table must be created, a new table_obj (head) is initialized and the *current head* pointer is updated to point to the newly created head. To exit from the sub-table, the current-head is set to point to the **parent** of the current head.

Two slightly different approaches were implemented for search over the symbol table:

- A global search is performed for **variables' retrieval:** since a variable could be found both in the current sub-table as well as in some previous higher level table, the search must continue all the way up, until a root head is found. If a root head is reached, and the variable still was not found, the algorithm can prove the non existence of the searched variable
- A local search is performed for **variables' insertion:** since a new variable, that was already declared in a higher-level table, can be redeclared in a sub-table, upon inserting a new variable, the search to see if it was already declared can be limited to the current sub-table

A boolean variable works as toggle between global and local variables' search.

Type System

The approach to the type system adopted for the language was to have a strong and static type checking.

Type casting

Automatic casting is done at parsing time when integer expressions appear with double expressions in an operation. In this case, the integer is cast to double, and the final expression result is computed.

Input

The adopted syntax is very similar to python code, with the main difference being that the control structures use curly brackets instead of indentation. Some examples are provided in the folder inputs.

Initialization and Usage

Environment execution:

IMPORTANT!
To run the compiler we strongly recommend the UNIX environment, since we weren't able to guarantee the correct compilation of the lex and yacc files in the Windows environment. Some compatibility issues have been encountered regarding the C99 standard for the C programming language, and we weren't able to implement a consistent solution to avoid these issues.

Compiler's execution:

to run the compiler, the user must have **the gcc compiler** installed.
If this is the case, execute from the directory where the zipped submission has been unzipped the following commands:

```
gcc y.tab.c -ly -ll
./a.out -i <input_file> [-o <output_file>]
```

NOTE: Execution of the parser requires ***explicit specification of an input file*** *that will be compiled. The input file is specified as "-i <input_file>". Examples of input files are in the folder "sample_programs". Therefore, an example of parser execution is:*

./a.out -i sample_programs/well_formatted/cond_stmt/a.txt;

Furthermore, the command can be extended with the optional "-o <output_file>" flag. If this command is not added explicitly, a default output file called "default.out" is created, whereas with this command it is possible to specify a custom name for the output file.

Additional material

Notion of expressions

An expression could be of different kinds:

- a constant value: `10; true; ...`
- the result unary expression: `Op Operand`
- the result binary expression: `Operand Op Operand`

- a variable evaluation: `a;`, where a is a previously defined variable

Expression Evaluation

Expressions are evaluated following a bottom-up procedure, starting from the single operands' value, up to the final result.

Following paragraphs specify how the different kinds of expressions are evaluated.

Constants

The evaluation in this case is straightforward and very intuitive. The Lexical Analyzer identifies the different tokens representing constant values, that are: *true*, *false*, *integer* and *decimal numbers.* It returns them to the Parser and places the relevant information in the YYLVAL variable. The parser assigns the constants to their correspondent type and value attributes, based on the information stored in the global variable, and populates the *parse_tree_node_info* variable.

Two different evaluation approaches were adopted for boolean and number constants. For the boolean constants, evaluation is trivial, since the TRUE and FALSE tokens directly identify the true and false values. For the numerical constants, the actual values are extracted invoking the atof() C function.

Unary Expression Evaluation

Expressions of this type are composed of two nested expressions. Therefore, to compute the final result, the parent attributes can be evaluated only after successfully evaluating the inner expression and ensuring type compatibility with the outer operator. If the inner expression's type is compatible with the outer operator, attributes of the whole expression are evaluated. Otherwise, a semantic error is generated.

Binary Expression Evaluation

These expressions are composed of three expressions: the parent (most general), and the two sub-expression representing the operands. As for the above mentioned unary case, the whole expression is evaluated only if the types of the inner-expressions are compatible with the operator in between them.

Variable Evaluation

To evaluate expressions of this kind, a table lookup operation needs to be performed. When the Lexical Analyzer recognizes a variable, it communicates the lexeme to the Parser, which uses the variable name to lookup the table. The search can either be successful, if the variable was declared, or fail. In the first case, attributes of the expression associated with the variable are returned, whereas in the second case, a PARSING ERROR is returned.

Operations Supported

The Parser implementation supports the following operations:

- Arithmetic Operations:
    - plus (+)
    - minus (-)
    - multiplication (*)
    - division (/)

- Boolean Operations:
    - greater than (>)
    - greater or equal than (>=)
    - smaller than (<)
    - greater or equal than (<=)
    - equality (==)
    - inequality (!=)