

Università degli Studi di Brescia

Department of Information Engineering



A compiler for the SOL language

Compilers' course final project

Professor

Lamperti Gianfranco

Students

Orizio Riccardo

Rizzini Mattia

Zucchelli Maurizio

Academic Year 2013/2014

Contents

I	Introduction to SOL	1
1	SOL language introduction and examples	2
1.1	A full sol program	3
2	SOL language syntax specification	4
3	SOL language semantics specification	7
II	The Compiler	9
4	Lexical and Syntactical analysis	10
4.1	Lexical analyzer	11
4.2	Syntactical analyzer	13
5	Semantical analysis	17
6	Code generation	18
6.1	S-code specification	18
6.2	The yygen function	19
6.2.1	Function problem	19
III	The Virtual Machine	20
7	Introduction and S-code execution	21
8	Graphical interface	22
	Conclusions	23

Listings

1.1	First code example	2
3.1	Definitions and assignments	7
4.1	Lex definition of lexical elements	11
4.2	Lex rule	12
4.3	Lex rule for a keyword	12
4.4	Lex rules for constants and ids	12
4.5	The Node structure	14
4.6	Structure of a translation rule	15
4.7	Extract of the translation rules for SOL	16

Part I

Introduction to SOL

Chapter 1

SOL language introduction and examples

The project here presented aims at the realization of a full Compiler and execution environment for the SOL programming language. The execution environment comprises a Virtual Machine which executes the intermediate code (namely S-code) produced as result of the compilation. Such Virtual Machine embodies an interface that allows the user to load a source or compiled SOL file and execute it (eventually after compilation) and presents a pleasant and usable graphical environment for the input and output of data.

SOL (Structured Odd Language) is a classic procedural programming language, not different from C.

In every SOL program there is a main *function* that contains the main code (just like the *main* procedure in C, with the difference that, here, we don't need to call this function in a particular way). The function is defined in a precise manner, as in Code 1.1.

```
1 func function_name( param: int; ): int
2     types
3         T: vector[10] of string;
4     var
5         c: char;
6         i, j: int;
7         x: bool;
8     const
9         name: string = "hello, world!";
```

```

10
11     func sub_function(): int
12         var
13             s: string;
14     begin sub_function
15         return 0;
16     end sub_function
17 begin function_name
18     return 0;
19 end function_name

```

Listing 1.1: First code example

As we can notice immediately, the function definition is divided in two main parts: a *header* in which the local elements (types, variables, constants and functions) are defined, and a *body* in which the function performs whatever it should.

Any function defined in the header of the main function must follow the very same structure. As we can notice from the above example, however, all the sections in the header and the function parameters are facultative. The only mandatory part is, in fact, the body.

// TODO whatelse?

1.1 A full sol program

We decided to implement *Conway's Game of Life* as an example of full program that can run with our SOL compiler and virtual machine. The program, in particular, allows us to test the I/O interface in an extensive manner.

// TODO GoL

Chapter 2

SOL language syntax specification

In this chapter is presented the formal specification of the syntax of SOL, informally presented in the previous chapter.

Note that the syntax is not left recursive, therefore it is suitable to both top-down and bottom-up parsing. The syntax is expressed in *BNF* and not in *EBNF* because we use *Yacc* to implement the parser, and *BNF* maps directly to the specification of *Yacc*.

The precedence of operators is resolved automatically by defining four levels of operations.

// TODO extend?

```
program → func_decl
func_decl → funcid ( decl_list_opt ) :
    domain type_sect_opt var_sect_opt const_sect_opt func_list_opt func_body
decl_list_opt → decl_list | ε
decl_list → decl ; decl_list | decl ;
decl → id_list : domain
id_list → id , id_list | id
domain → atomic_domain | struct_domain | vector_domain | id
atomic_domain → char | int | real | string | bool
struct_domain → struct ( decl_list )
vector_domain → vector [ intconst ] of domain
type_sect_opt → type decl_list | ε
var_sect_opt → var decl_list | ε
const_sect_opt → const const_list | ε
const_list → const_decl const_list | const_decl
```

$const_decl \rightarrow decl = expr;$
 $func_list_opt \rightarrow func_list | \epsilon$
 $func_list \rightarrow func_decl func_list | func_decl$
 $func_body \rightarrow begin id stat_list end id$
 $stat_list \rightarrow stat; stat_list | stat;$
 $stat \rightarrow assign_stat | if_stat | while_stat |$
 $for_stat | foreach_stat | return_stat | read_stat | write_stat$
 $assign_stat \rightarrow left_hand_side = expr$
 $left_hand_side \rightarrow id | fielding | indexing$
 $fielding \rightarrow left_hand_side.id$
 $indexing \rightarrow left_hand_side[expr]$
 $if_stat \rightarrow if expr then stat_list elsif_stat_list_opt else_stat_opt endif$
 $elsif_stat_list_opt \rightarrow elsif expr then stat_list$
 $elsif_stat_list_opt | \epsilon$
 $else_stat_opt \rightarrow else stat_list | \epsilon$
 $while_stat \rightarrow while expr do stat_list endwhile$
 $for_stat \rightarrow for id = expr to expr do stat_list endfor$
 $foreach_stat \rightarrow foreach id in expr do stat_list endforeach$
 $return_stat \rightarrow return expr$
 $read_stat \rightarrow read specifier_opt id$
 $specifier_opt \rightarrow [expr] | \epsilon$
 $write_stat \rightarrow write specifier_opt expr$
 $expr \rightarrow expr bool_op bool_term | bool_term$
 $bool_op \rightarrow and | or$
 $bool_term \rightarrow rel_term rel_op rel_term | rel_term$
 $rel_op \rightarrow == | != | > | >= | < | <= | in$
 $rel_term \rightarrow rel_term low_bin_op low_term | low_term$
 $low_bin_op \rightarrow + | -$
 $low_term \rightarrow low_term high_bin_op factor | factor$
 $high_bin_op \rightarrow * | /$
 $factor \rightarrow unary_op factor | (expr) | left_hand_side |$
 $atomic_const | instance_construction | func_call | cond_expr |$
 $built_in_call | dynamic_input$
 $unary_op \rightarrow _ | not | dynamic_output$
 $atomic_const \rightarrow charconst | intconst | realconst | strconst | boolconst$
 $instance_construction \rightarrow struct_construction | vector_construction$
 $struct_construction \rightarrow struct (expr_list)$
 $expr_list \rightarrow expr, expr_list | expr$

$vector_construction \rightarrow vector(expr_list)$
 $func_call \rightarrow id(expr_list_opt)$
 $expr_list_opt \rightarrow expr_list \mid \epsilon$
 $cond_expr \rightarrow if\ expr\ then\ expr\ elsif_expr_list_opt\ else\ expr\ endif$
 $elsif_expr_list_opt \rightarrow elsif\ expr\ then\ expr\ elsif_expr_list_opt \mid \epsilon$
 $built_in_call \rightarrow toint_call \mid toreal_call$
 $toint_call \rightarrow toint(expr)$
 $toreal_call \rightarrow toreal(expr)$
 $dynamic_input \rightarrow rd\ specifier_opt\ domain$
 $dynamic_output \rightarrow wr\ specifier_opt$

Chapter 3

SOL language semantics specification

This chapter presents the semantics of every statement of the SOL language in an operational way. The language used to describe the semantics is C.

// TODO explain everything in spec.pdf

Listing 3.1: Definitions and assignments

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

Part II

The Compiler

Chapter 4

Lexical and Syntactical analysis

Our compiler is written in the C language. It is divided in three main parts that correspond to the three stages of compiling, executed in sequential order:

- The lexical and syntactical analysis of the language, presented in this chapter, that together aim at determining whether the given SQL source file is well-written or not and to construct a data structure that describes the code in a functional manner;
- The semantical analysis, presented in Chapter 5, which aims at determining if the written statements (which are correct due to the previous analysis) make sense (eg, performing the sum of an integer and a string makes no sense, therefore it is not semantically correct), relying on the data structure produced by the previous analysis;
- The code generation, presented in Chapter 6, which, given that the code is both well-written and semantically correct, translates it in a lower-level and standard code, easier to execute directly (and executed by the virtual machine, of which we talk in Part III). The code is, again, generated starting from the data structure produced by the syntactical analysis, not from the “raw” code.

Our compiler uses Lex and Yacc to perform lexical and syntactical analysis, respectively. These are two languages specifically designed for this purpose and they produce complete analyzer programs written in C.

Listing 4.2. The value returned by each rule must be a unique identifier of the found lexeme.

```

1 lexeme { /*C code to execute when such lexeme is found*/
    ; return lexeme_descriptor; }

```

Listing 4.2: Lex rule

For the fixed lexeme (keywords and other simple stuff), the rules are as simple the ones in Listing 4.3. The complexe lexeme, however, comprise a “value”, since they’re not fixed. This value must be “elaborated” from the “raw” (simple string) value presented in the variable *yytext* and put in a new variable that will be used to build the symbol table. The elaboration consists, normally, in the conversion of the value to the correct type. In our program, the destination variable is *lexval*, instance of *Value*, a C union that can contain any type of value (integer, real, string..). The rules for the complexe lexeme are all presented in Listing 4.4.

```

1 func { SPAM( "FUNC" ); return( FUNC ); }
2 char { SPAM( "CHAR" ); return( CHAR ); }
3 int { SPAM( "INT" ); return( INT ); }
4 real { SPAM( "REAL" ); return( REAL ); }
5 string { SPAM( "STRING" ); return( STRING ); }

```

Listing 4.3: Lex rule for a keyword

```

1 {intconst} { SPAM( "INT_CONST" ); lexval.i_val = atoi(
    yytext ); return( INT_CONST ); }
2 {strconst} { SPAM( "STR_CONST" ); yytext[ strlen( yytext
    ) - 1 ] = '\0'; lexval.s_val = new_string( yytext + 1
    ); return ( STR_CONST ); }
3 {charconst} { SPAM( "CHAR_CONST" ); yytext[ strlen(
    yytext ) - 1 ] = '\0'; lexval.s_val = new_string(
    yytext + 1 ); return( CHAR_CONST ); }
4 {realconst} { SPAM( "REAL_CONST" ); lexval.r_val = atof(
    yytext ); return( REAL_CONST ); }
5 {boolconst} { SPAM( "BOOL_CONST" ); lexval.b_val = (
    yytext[ 0 ] == 'f' ? FALSE : TRUE ); return(
    BOOL_CONST ); }
6 {id} { SPAM( "ID" ); lexval.s_val = new_string( yytext )

```

```
    ; return( ID ); }  
7 {sugar} { SPAM( yytext ); return( yytext[ 0 ] ); }  
8 . { yyerror( STR_ERROR ); }
```

Listing 4.4: Lex rules for constants and ids

The last line of 4.4 means that whatever doesn't match the previous rules must result in an error (as in the regular expressions, "." means any value). The *SPAM* function is simply a redefinition of `fprintf` pointing to the standard error.

4.2 Syntactical analyzer

Similarly to Lex, Yacc is used to produce a syntactical analyzer in C. The compilation of the Yacc file produces a C file containing a function called *yyparse* that, given the source file and the Symbol Table produced by *yylex*, checks its syntax correctness and produces another data structure (the *Syntax Tree*) if everything is correct.

The Syntax Tree is realized with the *Node* structure, presented in Listing 4.5 along with the union *Value*. A *Node* contains:

- The number of line in the code in which it appears;
- A *type*, which says what the node represents. In particular, the type is represented as an enumerator which values are the *terminals* (integer constant, id, etc) and *nonterminals* (mathematical expressions, assignments, etc) allowed in SOL. To simplify the produced syntax tree, the nonterminals are divided in two categories: the *qualified* nonterminals are aggregated nonterminals that are then differentiated by mean of a qualifier (eg mathematical expressions are one type of nonterminal and their qualifier is the operator), while the *unqualified* nonterminals are those that cannot be aggregated (eg an if). To sum up things, the type can either be a terminal, a qualified nonterminal or the special value unqualified nonterminal. The specific type of unqualified nonterminal represented by the node is then contained in the node's value, and so does the qualifier;
- A *value*, represented as an instance of the union *Value*, that can be an elementary value (integer, string..) if the node is a terminal, a unique

identifier determining the nonterminal type if the node is an unqualified nonterminal (the identifiers are represented as possible values of the enumerator *NonTerminal*) or a unique identifier determining the qualifier to be used if the node is a qualified nonterminal (these are represented as possible values of the enumerator *Qualifier*);

- A pointer to the *leftmost child*;
- A pointer to the *first right brother*.

```
1 typedef struct snode
2 {
3     int line;
4     Value value;
5     TypeNode type;
6     struct snode* child;
7     struct snode* brother;
8 } Node;
9
10 typedef union
11 {
12     int i_val;
13     char* s_val;
14     double r_val;
15     Boolean b_val;
16     Qualifier q_val;
17     Nonterminal n_val;
18 } Value;
```

Listing 4.5: The Node structure

The syntactical analyzer (also called *parser*) stores as global variable a pointer to the root node of the tree. Note that the tree generated is not the *concrete tree* (that is, the tree that would be generated by direct application of the BNF definition) but an *abstract tree* that cuts off some node without loss of information but with great gain in space occupation and visiting time (eg the expressions are defined in 4 levels to maintain the correct precedence when analyzing the code; these levels are of no use after the code has been recognized in the correct order, therefore there are no levels in the resulting

abstract tree).

// TODO add characteristics of Yacc (leftmost lookahead etc)

The Yacc file is divided in three parts, whose purpose is the same as that of those in a Lex file. Here, in the first part instead of defining the complex lexeme we instruct Yacc about which these lexeme are, by defining all the possible unique identifiers returned by the Lex rules as *tokens*. The second part contains *translation rules* for every syntactical element of the language (all those defined in the BNF description, presented in Chapter 2), and the third part contains definitions for the C functions used in the translation rules.

A translation rule must create a Node and populate it with the appropriate informations. The structure of a translation rule is the one presented in Listing 4.6.

```

1 syntactical_element : /* definition as in the BNF */ { $$
    = /* code that creates the Node */ }
2     | /* alternate definition */ { $$ = /*
    alternate code */ }
3     ;

```

Listing 4.6: Structure of a translation rule

At the left of the colon there is the name of the element, at the right there is a sequence of definitions, each associated to a code that is executed to create the node when that particular definition is found. The definitions are separated by “|” and the rule must terminate with a semicolon.

In the code, the symbol “\$\$” represent the lhs of the rule, and the elements of a definition can be referred to as “\$n”, where n is the position of the element in the definition starting from 1.

The *yyparse* function generated starting from the Yacc file is a recursive function. The code is searched recursively for structures that match the lhs of a translation rule. Once a match is found, for every element in the rhs the function is called again and the process keeps going until every element in the rhs is either a token (which means that it has been processed by *yylex* and its value is in the Symbol Table, therefore no further processing is required) or has been processed completely by the recursion. At this point, the Node for that rule can be processed and returned to the caller (which will be another

rule or the main program if the rule was the root one).

Knowing how the parsing works, we can understand why there must always be a “root” rule that will be matched at the first call of *yyparse* (if the code is correct, obviously) and associates the result of the subsequent calls to the global *root* variable, instead of returning it to the caller (thus, assign it to *\$\$*) like the others. In Listing 4.7 we present, as an example, the root translation rule and the translation rule for a function declaration.

```

1 program : func_decl { root = $1; }
2         ;
3 func_decl : FUNC ID { $$ = new_terminal_node( T_ID,
4             lexval ); }
5             '(' par_list ')', DEFINE domain
6             type_sect_opt var_sect_opt const_sect_opt
7             func_list_opt func_body
8             {
9                 $$ = new_nonterminal_node( N_FUNC_DECL );
10                $$->child = $3;
11                Node** current = &($$->child->brother);
12                current = assign_brother( current, $5 );
13                current = assign_brother( current, $8 );
14                current = assign_brother( current, $9 );
15                current = assign_brother( current, $10 );
16                current = assign_brother( current, $11 );
17                current = assign_brother( current, $12 );
18                current = assign_brother( current, $13 );
19            }
20        ;

```

Listing 4.7: Extract of the translation rules for SOL

Note that C code can be inserted in any position between the elements of the rhs, and it must produce something that will then be referred to as *\$n*, just like a normal element. In the presented example, we use this method to create immediately a Node containing the id of the declared function, and this node is then assigned as leftmost child of the node created for the whole rule.

Chapter 5

Semantical analysis

Starting from the tree, the *yysem* function (this time written entirely by us, as there's no language for generating a semantical analyzer automatically) analyzes the whole code in search for semantical errors. To support itself in this operation, it produces a Symbol Table containing all the elements in the code, each of which will be associated with a detailed description of its position in the code, a unique identifier and a schema describing its type (simple or complex).

Please note that, even if this structure is called Symbol Table as the one produced by yylex, it is something entirely different, as yylex simply created a hashmap in which lexeme names and values were associated for simpler further reference.

// TODO IDK how this works

Chapter 6

Code generation

Starting from the tree generated by the syntactic analysis and the table produced by the semantical one, the *yygen* (again, written by us) function proceeds to the code generation. The function operates calling the recursive function *generate_code*, which proceeds starting from the root node and generating the code for all nodes from the tree's leftmost to the rightmost.

Since the function *yygen* operates on the product of the analysis steps, it doesn't check anything (if something was wrong, the compiler's execution would have been already stopped).

6.1 S-code specification

The code generation translates the SOL code in S-code code. S-code is a very low level language not dissimilar from Assembly.

Everything is done on a global stack. Every instruction has zero to three operands and operates implicitly on the last values present on the stack (generally the last one or two). For example, the instruction to perform a sum of integer is called *IPLUS* and it has no operands. What it does is take the last two values present on the stack, sum them and put the result back on the stack. Obviously, every value used is also consumed.

Being so easy, it is not difficult to generate the appropriate sequence of instructions for every instruction available in SOL.

```
// TODO include S-code generation or write "see Lamperti's stuff"?
```

6.2 The *yygen* function

When the *yygen* function is called, it automatically retrieves the root of the Syntax Tree and passes it to *generate_code*. This function consists of a big switch of the node's type and, for every type, it generates an instance of *Code* (a structure pointing to a list of pointers to another structure *Stat*, which in turn contains the actual code) in different ways depending on the type. If the type of the node is *unqualified nonterminal*, there is another big switch on the node's *n_val* (that is, the node's value determining the exact type of nonterminal represented).

The *generate_code* function returns the code which is concatenated following the order of recursion and, in the end, *yygen* gets the full code.

```
// TODO amplify
```

6.2.1 Function problem

Part III

The Virtual Machine

Chapter 7

Introduction and S-code execution

The Virtual Machine reads the S-code produced as result of the compilation and executes it.

```
// TODO all
```


Chapter 8

Graphical interface

The virtual machine has a beautiful graphical interface realized with the Qt5 graphical environment.

All the graphical part is realized entirely in Python 3.4 using the Qt5 designer editor, and integrated in a full Python program called *solGUI.py*. The interaction between the interface and the virtual machine is in both directions.

By calling *solGUI.py* a window will appear in which the user can input a SOL source file, compile it and execute the resulting S-code file (or directly input the S-code file). During the execution, in correspondance of every user input or user output, the virtual machine will query Python to open a window with which the user can input the required data or visualize the output.

// TODO expand

Conclusions

WYNOUNICODEBRO:(