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

List of Code chunks

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 (Structured Odd Language) 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. An explanation of how this interface works and how it can be used to compile and execute a SOL file is presented in Chapter 7.

SOL is a classic procedural programming language.

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 hello_world(): int
2  begin hello_world
3    write "Hello world!";
4    return 0;
5  end hello_world
```

Code 1.1: Hello world program

In this first example we can notice that a function definition is essentially divided in two parts: a *header*, in which the function's name and its return

6

type are declared, and a *body* in which the function's instructions are written.

This first example, obviously, does not comprise all the elements allowed in a function's header. The purpose of the header is to define all the objects of the function's local environment, that is, all the objects usable in the function's body. These objects fall in five categories, and their definitions must be written in the presented order:

The function's parameters These are defined in the round brackets after the function's name as a list of variable definitions (as can be seen in Code 1.2). A variable definition must be in the form *variable_ name: type;* and any number of variable of the same type can be defined with a single instruction by listing all the variables' names before the colon separated by commas. The types allowed in SOL are the simple types *int, real, bool, string, char*, the two complex ones *vector* and *struct* (whose syntax is defined later) and all the user-defined types.

```
1  func program( par, par_two: int; par_three: string; ): int
```

Code 1.2: Parameters

A list of types The definition of a type has the very same purpose of the instruction *typedef* in C, and any type can be redefined with a custom name, although this is particularly useful only with complex types. The syntax of a type definition is very similar to that of a variable definition, as can be seen in Code 1.3. In the same example we can also see how a complex type is defined. A vector must follow the syntax *vector[size] of element_ type;* while for a struct one must write the keyword *struct* followed by round brackets in which a list of variables is contained. The variables in the list are the fields of the structure. The types are completely orthogonal (one can define a vector of structs containing vectors, for example).

```
1    type
2      from_slides: vector[ 10 ] of
3          struct( la: int: lala: vector[ 20 ] of vector[ 5 ] of real );
4      T2: string;
```

Code 1.3: Types

A list of variables This is similar to that in the definition of parameters, as in Code 1.4.

```
1    var
2      c: char;
3      i: int;
4      x, y, z: real;
5      s: string;
6      b: bool;
7      r: struct( a: char; b: string; );
8      v: vector [ 5 ] of int;
9      w: vector [ 100 ] of struct( a: int; b: char; );
10     out_x: real;
11     out_v: vector [ 10 ] of real;
```

Code 1.4: Variables

A list of constants The definition of a constant is identical to that of variables except for the fact that a value must be assigned to each constant at definition time (see Code 1.5).

```
1    const
2      MAX: int = 100;
3      name: T = "alpha";
4      PAIR: struct( a: int; b: char; ) = struct( 25, 'c' );
5      VECT: vector [ 5 ] of real = vector( 2.0, 3.12, 4.67, 1.1, 23.0 );
6      MAT: vector [ 2 ] of vector [ 5 ] of real =
7          vector( VECT, vector( x, y, z, 10.0, x+y+z ) );
```

Code 1.5: Constants

A list of functions Every function is defined exactly as the main one. These functions will be visible inside the main function since they are part of its environment. They can, obviously, contain other functions' definitions, and these are also visible by their parent function but not from their brothers and their brothers' children.

These are all the things that a function's header can contain. Note that none of this parts is mandatory. The only mandatory part is, in fact, the body (which can contain any instruction except a definition).

The body of a function must always contain a *return* statement in every branch (if the body is branched by a conditional statement and a branch contains a *return*, then every other branch must terminate with one too).

1.1 The body of a function - instructions of SOL

In this section we present all the instructions allowed in a SOL program (except for the definitions, explained before). They follow a syntax which is pretty standard, anyway.

1.1.1 Access to struct fields and vector values

Access to a field of a structure is done with a dot, while indexing a vector is done by enclosing the index in square brackets. Both operations are exemplified in Code 1.6. Note that a double dash starts a comment.

```
1  i, j: int;
2  l: string;
3  t: vector [ 5 ] of int;
4  s: struct( a: int; b: struct( c: string; ); );
5  v: vector [ 10 ] of vector [ 5 ] of int;
6  ...
7  -- Reference to the field a of the structure s
8  i = s.a;
9  -- Reference to the field c of the field b of the structure s
10 l = s.b.c;
11 -- Indexing of the fourth element of v
12 t = v[3];
13 -- Indexing of the second element of the first element of v
14 j = v[0][1];
```

Code 1.6: Examples of indexing and fielding

Also note that, as in the presented example, every indexing and fielding must be used as an operator of an expression, and an expression cannot be written as a standalone instruction (it can be used as rhs of an assignment, in another expression, as value passed to a parameter, etc).

1.1.2 Arithmetic expressions

In SOL, any numerical (integer or real) variable can be involved in an arithmetic expression, but the language does not provide implicit type coercion. This means that an expression can contain only real or only integer operands, but there are two functions that allow to mix things up by providing explicit casting. These two functions are *toint* and *toreal* (with obvious semantics).

The operands are the same for real and integer variables and are + (plus), - (minus), / (divide) and * (multiply). The minus operand can be applied to a single value to obtain its opposite ($-a = -1*a$), otherwise all the operands are binary, left associative, and multiply and divide have higher precedence than minus and plus (the unary minus has the highest precedence, though).

The operands in an expression are evaluated from left to right.

An example of arithmetic expression is presented in Code 1.7.

```
1  i, j: int;
2  x, y: real;
3  ...
4  x = toreal( i + j ) * ( r - toreal( i ) );
5  j = toint( x + y - 1.25 );
```

Code 1.7: Example of arithmetic expression

In this example we can also notice the syntax of an assignment, simply *variable_ name = value;*.

1.1.3 Conditional constructs and logical expressions

There are two conditional constructs in sol: the conditional *expression* is an expression that assumes different values when different conditions apply, while the conditional *statement* is a construct that allows different lists of statements to be executed when different conditions apply.

The syntax of a conditional expression is presented in Code 1.8, while that of a conditional statement is presented in Code 1.9.

```
1  a, b, c: int;
2  ...
3  a = if b > c then b + c elsif b == c + 1 then b - c else a + 1 endif;
```

Code 1.8: Example of conditional expression

```
1  a, b, c: int;
2  ...
3  if a == b then
4     b = c;
5  elsif a > b then
6     c = b + a;
7  else
8     c = b - a;
9  endif;
```

Code 1.9: Example of conditional statement

Both constructs use the same keywords in the same order, in both cases after an *if* or an *elsif* there must be a logical expression and both constructs terminate with the keyword *endif* followed by a semicolon. The main difference is that in a conditional expression what follows a *then* or an *else* must be an expression returning a single value, while in a conditional construct such keywords can be followed by any number of statements. Moreover, in a conditional expression the *else* clause is mandatory (to ensure that the expression always assumes a value), while in a conditional statement is facultative. In both constructs the *elsif* clauses are not mandatory.

The logical expressions allows the boolean binary operators *and* (conjunction) and *or* (disjunction) and the unary *not* (negation). The negation operator has higher precedence than the other two.

Operands of a logical expression can be relational expressions, that is, expressions with arithmetic expressions as operands involving the operators $==$ (equality), $!=$ (inequality), $>$ (greater than), $>=$ (greater than or equal), $<$ (less than), $<=$ (less than or equal), *in* (membership). The equality and inequality operators can be applied to any type of operands, while the other (except for the membership, which is somewhat special) can only be applied to integer, real or string operands. Operands must be of the same type. In the case of the membership, the second operand must be a vector and the first operand must be of the same type of the vector's elements.

Combining all the types of operators, the precedence goes as in Table 1.1 (precedence increases with the number of line).

and, or
$==$, $!=$, $>$, $>=$, $<$, $<=$, in
+, - (binary)
*, /
- (unary), not

Table 1.1: Precedence of operators

1.1.4 Cycles

There are three types of loop constructs in SOL: the *while* (Code 1.10), the *for* (Code 1.11) and the *foreach* (Code 1.12).

```
1  a, b, c: int;
```

```
2  ...
3  while a >= b do
4    a = a - c;
5  endwhile;
```

Code 1.10: Example of while loop

```
1  i, k: int;
2  v: vector [ 100 ] of int;
3  ...
4  for i = 1 to 100 do
5    k = k + v[ i ];
6  endfor;
```

Code 1.11: Example of for loop

```
1  i, k: int;
2  v: vector [ 100 ] of int;
3  ...
4  foreach i in v do
5    k = k + i;
6  endforeach;
```

Code 1.12: Example of foreach loop

In the for loop, the counting variable (i, in the example) cannot be assigned within the loop body. The semantics of the while and for loops is the classical one, and the foreach is, intuitively, a loop in which, at every iteration, the "counting" (it's not really counting) variable assumes the following value of the vector (in the example, v). It is a nice method of vector iteration.

1.1.5   Input/Output

SOL feature two instructions to produce output (*write* and *wr*) and two to request input (*read* and *rd*). The syntax of the four instructions is presented in Code 1.13.

```
1  a, b, c: int;
2  filename: string;
3  ...
4  write [ filename ] a;
5  b = wr [ filename ] a + c;
6
7  read [ filename ] a;
8  b = rd [ filename ] int;
```

Code 1.13: Examples of I/O instructions

The *write* and *wr* instructions both require to specify the name of the file on which the data has to be written in square brackets (not mandatory, if missing the output is presented on the standard output, ie the monitor), followed by an expression which result is the data to write. The difference is that *wr* also assumes that value and, therefore, it can be used as rhs (right hand side) of an assignment.

The *read* and *rd* instructions are, instead, slightly different from each other. Both require to specify the name of the file from which to read in square brackets (again, not mandatory), but *read* then requires the name of the variable in which the read data has to be saved, while *rd* requires the type of the data read. It then returns the read value assuming that it is of the specified type.

1.1.6 Function call

A function can be called simply by writing its name followed by a list of values for the parameters enclosed in round brackets, but it always have to be used as an operand of an expression.

1.2 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 of the virtual machine in an extensive manner.

```
1  func game_of_life () : int
2
3    type
4      lines: vector [ 15 ] of bool;
5      grid: vector [ 15 ] of lines;
6    var
7      state: struct( generation: int; world: grid; );
8      input: struct( filename: string; load: bool; );
9      generations: int;
10   const
11     world_size: int = 15;
12     str_summary: string = "Welcome to ORZ's Conway's Game of Life!";
13     str_goodbye: string = "Thanks for playing with ORZ's Conway's Game of
```

```
14            Life! \n\n\tBye!";
15        str_saved: string = "Your data has been successfully saved in the
16          following file:";
17        enter_filename: string = "Enter the filename of your world and if you'd
18          like to load from a saved state.";
19        enter_generations: string = "Enter for how many generations would you
20          like to watch your world go by.";
21        enter_world: string = "Your world doesn't exist yet.\nEnter it now.";
22
23    -- Rules:
24    -- * Any live cell with fewer than two live neighbours dies, as if caused
25    --     by under-population.
26    -- * Any live cell with two or three live neighbours lives on to the next
27    --     generation.
28    -- * Any live cell with more than three live neighbours dies, as if by
29    --     overcrowding.
30    -- * Any dead cell with exactly three live neighbours becomes a live
31    --     cell, as if by reproduction.
32
33    func next_state( current_state: grid; ) : grid
34      var
35        i, j, k, h: int;
36        neighbours: int;
37        state: grid;
38      const
39        neighbour_offset: vector[ 3 ] of int = vector( -1, 0, 1 );
40
41    begin next_state
42      for i = 0 to world_size-1 do
43        for j = 0 to world_size-1 do
44          neighbours = 0;
45          foreach k in neighbour_offset do
46            foreach h in neighbour_offset do
47              if k != 0 or h != 0 then
48                if i+k >= 0 and i+k < world_size and
49                   j+h >= 0 and j+h < world_size then
50                  if current_state[ i + k ][ j + h ] then
51                    neighbours = neighbours + 1;
52                  endif;
53                endif;
54              endif;
55            endforeach;
56          endforeach;
57
58          state[ i ][ j ] = if current_state[ i ][ j ]
59                            then neighbours == 2 or neighbours == 3
60                            else neighbours == 3 endif;
61        endfor;
62      endfor;
63
64      return state;
65    end next_state
66
67 begin game_of_life
68   write str_summary;
69   write enter_filename;
70   read input;
```

```
71
72    if input.load then
73      read [ input.filename ] state;
74    else
75      write enter_world;
76      state.world = rd grid;
77    endif;
78
79    write enter_generations;
80    read generations;
81
82    for generations = 0 to generations-1 do
83      state.world = next_state( state.world );
84      state.generation = state.generation + 1;
85      write state;
86    endfor;
87
88    write [ input.filename ] state;
89    write struct( str_goodbye, struct( str_saved, input.filename ) );
90
91    return 0;
92 end game_of_life
```

Code 1.14: Game of Life

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.

$program \rightarrow func\_decl$
$func\_decl \rightarrow$ **func id** ($decl\_list\_opt$) :
   $domain\, type\_sect\_opt\, var\_sect\_opt\, const\_sect\_opt\, func\_list\_opt\, func\_body$
$decl\_list\_opt \rightarrow decl\_list \,|\, \epsilon$
$decl\_list \rightarrow decl \,;\, decl\_list \,|\, decl \,;$
$decl \rightarrow id\_list\, :\, domain$
$id\_list \rightarrow$ **id** , $id\_list \,|\,$ **id**
$domain \rightarrow atomic\_domain \,|\, struct\_domain \,|\, vector\_domain \,|\,$ **id**
$atomic\_domain \rightarrow$ **char** | **int** | **real** | **string** | **bool**
$struct\_domain \rightarrow$ **struct** ($decl\_list$)
$vector\_domain \rightarrow$ **vector [intconst] of** $domain$
$type\_sect\_opt \rightarrow$ **type** $decl\_list \,|\, \epsilon$
$var\_sect\_opt \rightarrow$ **var** $decl\_list \,|\, \epsilon$
$const\_sect\_opt \rightarrow$ **const** $const\_list \,|\, \epsilon$
$const\_list \rightarrow const\_decl\, const\_list \,|\, const\_decl$
$const\_decl \rightarrow decl\, =\, expr \,;$

$func\_list\_opt \rightarrow func\_list \,|\, \epsilon$

$func\_list \rightarrow func\_decl \, func\_list \,|\, func\_decl$

$func\_body \rightarrow \textbf{begin id} \, stat\_list \, \textbf{end id}$

$stat\_list \rightarrow stat \,;\, stat\_list \,|\, stat \,;$

$stat \rightarrow assign\_stat \,|\, if\_stat \,|\, while\_stat \,|$
$\quad for\_stat \,|\, foreach\_stat \,|\, return\_stat \,|\, read\_stat \,|\, write\_stat$

$assign\_stat \rightarrow left\_hand\_side \,=\, expr$

$left\_hand\_side \rightarrow \textbf{id} \,|\, fielding \,|\, indexing$

$fielding \rightarrow left\_hand\_side \,.\, \textbf{id}$

$indexing \rightarrow left\_hand\_side \,[\, expr \,]$

$if\_stat \rightarrow \textbf{if} \, expr \, \textbf{then} \, stat\_list \, elsif\_stat\_list\_opt \, else\_stat\_opt \, \textbf{endif}$

$elsif\_stat\_list\_opt \rightarrow \textbf{elsif} \, expr \, \textbf{then} \, stat\_list$

$elsif\_stat\_list\_opt \rightarrow$

$, \textbf{else} \,|\, \epsilon$

$else\_stat\_opt \rightarrow \textbf{else} \, stat\_list \,|\, \epsilon$

$while\_stat \rightarrow \textbf{while} \, expr \, \textbf{do} \, stat\_list \, \textbf{endwhile}$

$for\_stat \rightarrow \textbf{for id} \,=\, expr \, \textbf{to} \, expr \, \textbf{do} \, stat\_list \, \textbf{endfor}$

$foreach\_stat \rightarrow \textbf{foreach id in} \, expr \, \textbf{do} \, stat\_list \, \textbf{endforeach}$

$return\_stat \rightarrow \textbf{return} \, expr$

$read\_stat \rightarrow \textbf{read} \, specifier\_opt \, \textbf{id}$

$specifier\_opt \rightarrow [\, expr \,] \,|\, \epsilon$

$write\_stat \rightarrow \textbf{write} \, specifier\_opt \, expr$

$expr \rightarrow expr \, bool\_op \, bool\_term \,|\, bool\_term$

$bool\_op \rightarrow \textbf{and} \,|\, \textbf{or}$

$bool\_term \rightarrow rel\_term \, rel\_op \, rel\_term \,|\, rel\_term$

$rel\_op \rightarrow == \,|\, != \,|\, > \,|\, >= \,|\, < \,|\, <= \,|\, \textbf{in}$

$rel\_term \rightarrow rel\_term \, low\_bin\_op \, low\_term \,|\, low\_term$

$low\_bin\_op \rightarrow + \,|\, \_$

$low\_term \rightarrow low\_term \, high\_bin\_op \, factor \,|\, factor$

$high\_bin\_op \rightarrow * \,|\, /$

$factor \rightarrow unary\_op \, factor \,|\, (\, expr \,) \,|\, left\_hand\_side \,|$
$\quad atomic\_const \,|\, instance\_construction \,|\, func\_call \,|\, cond\_expr \,|$
$\quad built\_in\_call \,|\, dynamic\_input$

$unary\_op \rightarrow \_ \,|\, \textbf{not} \,|\, dynamic\_output$

$atomic\_const \rightarrow \textbf{charconst} \,|\, \textbf{intconst} \,|\, \textbf{realconst} \,|\, \textbf{strconst} \,|\, \textbf{boolconst}$

$instance\_construction \rightarrow struct\_construction \,|\, vector\_construction$

$struct\_construction \rightarrow \textbf{struct} \, (\, expr\_list \,)$

$expr\_list \rightarrow expr \,,\, expr\_list \,|\, expr$

vector_construction → **vector** (*expr_list*)
func_call → **id** (*expr_list_opt*)
expr_list_opt → *expr_list* | ϵ
cond_expr → **if** *expr* **then** *expr* *elsif_expr_list_opt* **else** *expr* **endif**
elsif_expr_list_opt → **elsif** *expr* **then** *expr* *elsif_expr_list_opt* | ϵ
built_in_call → *toint_call* | *toreal_call*
toint_call → **toint** (*expr*)
toreal_call → **toreal** (*expr*)
dynamic_input → **rd** *specifier_opt* *domain*
dynamic_output → **wr** *specifier_opt*

Part II

The Compiler

Chapter 3

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 SOL 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 4, which aims at determining if the written statements (which are correct thanks to the previous analyses) make sense (e.g., 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 5, 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 will 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 analysis programs written in C.

3.1 Lexical analyzer

LEX is used to produce a lexical analyzer in C language. After the LEX file compilation, we get a C file defining a function called *yylex*. This function reads from the input file, whose reference is stored in the variable *yyin*, and returns to the caller the first token found. While doing this, it creates a data structure (called *symbol table* and realized through an hash-map) containing all the symbols found in the code (that is, string constants and ids); this allows to create a single instance for every string given in input and to reduce name-checking from a comparison between strings to a comparison between pointers. If, during the analysis, an error is encountered (i.e., in the file is present something that isn't part of the language, like an id starting with a digit), the *yylex* function stops and produces an error calling the *yyerror* function.

This function will be called by the syntactical analyzer to check the syntax and produce the *syntax tree*, of which we talk in the next section.

The LEX file is divided in three parts. In the first part of the LEX file, the lexical elements (or *lexemes*) that need to be defined with a regular expression (such as the id) are defined, in the second part these lexemes are associated to a rule which defines the behavior of *yylex* when the specific lexeme is found, and the last part contains specific C functions used in the LEX rules of the second part. The lexemes that don't need to be defined in the first part are those whose denotation is fixed, such as keywords and operators.

The definition of lexemes for the SOL language is, for our compiler, the one presented in 3.1.

```
1   alpha              [a-zA-Z]
2   digit              [0-9]
3
4   id                 {alpha}({alpha}|{digit}|_)*
5
6   charconst          '([^\']|\\.)'
7   intconst           {digit}+
8   realconst          {digit}+\.{digit}+
9   boolconst          true|false
10
11  comment            --.*
12  spacing            ([ \t])+
13  sugar              [()\[\]{}.,;]
14
15  %x strconst
```

Code 3.1: Lex definition of lexical elements

The rules associated to each lexeme must be in the form presented in Code 3.2. The value returned by each rule must be an identifier (i.e., the value of an enumerator) of the found lexeme.

```
1  lexeme   { /*action when such lexeme is found*/; return lexeme_descriptor; }
```

Code 3.2: Lex rule

For the fixed lexemes (keywords and other simple stuff), the rules are usually as simple the ones in in Code 3.3. The complex lexemes, however, have a dynamic structure and hence are attached a value. This value must be elaborated from the raw textual value contained in the variable *yytext* and put in a new variable that will be used throughout the compiler. The elaboration consists, normally, in the conversion of the value to the correct type; for strings and ids the elaboration includes the addition of the textual value to the lexical symbol table. In our program, the destination variable is *lexval*, instance of *Value,* a union that can contain any type of value accepted by SOL (integer, real, string,...). The rules for the complex lexemes are all presented in Code 3.4.

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

Code 3.3: Lex rule for a keyword

```
1  {intconst}           { lexval.i_val = atoi( yytext );
2                         return( INT_CONST ); }
3  "\""                 { BEGIN strconst;
4                         strbuf = malloc( sizeof( char ) ); }
5  <strconst>([^"\n])*  { concatenate_string( &strbuf, yytext ); }
6  <strconst>\n[ \t]*   ;
7  <strconst>\"         { lexval.s_val = new_string( strbuf );
8                         BEGIN 0;
9                         return( STR_CONST ); }
10 {charconst}          { yytext[ strlen( yytext ) - 1 ] = '\0';
11                        lexval.s_val = new_string( yytext + 1 );
12                        return( CHAR_CONST ); }
13 {realconst}          { lexval.r_val = atof( yytext );
14                        return( REAL_CONST ); }
15 {boolconst}          { lexval.b_val = ( yytext[ 0 ] == 'f'
```

```
16                                          ? FALSE
17                                          : TRUE );
18                              return( BOOL_CONST ); }
19   {id}                      { lexval.s_val = new_string( yytext );
20                                return( ID ); }
21   {sugar}                   { return( yytext[ 0 ] ); }
22   .                         { yyerror( STR_ERROR ); }
```

Code 3.4: Lex rules for constants and ids

The last line of 3.4 means that whatever doesn't match the previous rules must result in an error (in the regular expressions, "." means any character). Our LEX file contains also, for each rule, some debugging (enabled with an apposite flag) code which is not included here for clarity.

3.2   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, through calls to *yylex*, checks syntax's correctness of the file and produces another data structure, the *Syntax Tree*, if everything is correct.

The syntax tree is realized using the *Node* structure, presented in Code 3.5 along with the union *Value*. A *Node* contains:

- The number of the line of code in which the represented syntactical symbol 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, which are aggregates of nonterminals differentiated by mean of a qualifier (e.g. mathematical expressions are one type of nonterminal and their qualifier is the mathematical operator), and the *unqualified* nonterminals, which are those that cannot be aggregated (e.g. an if statement). 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, as does the qualifier for qualified nonterminals;

- A *value*, represented through 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;
```

Code 3.5: 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 nodes without loss of information but with great gain in space occupation an visiting time (e.g., 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 in the resulting abstract tree just the most specific level is preserved).

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 lexemes we instruct YACC about which these lexemes 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 Code 3.6.

```
1  syntactical_element : /*BNF definition*/ { $$ = /*code to the Node*/ }
2                      | /*alternate definition*/ { $$ = /*alternate code*/ }
3                      ;
```

Code 3.6: 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 a pipe and the rule must terminate with a semicolon.

In the code, the symbol $$ represents the lefthand-side 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 implements a *Bottom-Up Parsing* method. In simple terms, the function is composed of two actions working on a stack, the action to execute is chosen basing the stack and on the next token. The first action, called *shift*, consists in pushing on the stack the newly found token; the second action, called *reduce*, simplifies a part of the stack according to the given BNF rules and executes the relative instructions

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 assigning it to $$, like the other rules. In Code 3.7 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, lexval ); }
4              '(' par_list ')' DEFINE domain type_sect_opt var_sect_opt
5              const_sect_opt func_list_opt func_body
6              {
```

```
7                        $$ = new_nonterminal_node( N_FUNC_DECL );
8                        $$->child = $3;
9                        Node** current = &($$->child->brother);
10                       current = assign_brother( current, $5 );
11                       current = assign_brother( current, $8 );
12                       current = assign_brother( current, $9 );
13                       current = assign_brother( current, $10 );
14                       current = assign_brother( current, $11 );
15                       current = assign_brother( current, $12 );
16                       current = assign_brother( current, $13 );
17                }
18           ;
```

Code 3.7: Extract of the translation rules for SOL

Note that C code can be inserted in any position between the elements of the righthand-side, 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 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. This is a required practice for ids and other constants since their value is readable from *lexval* only after the token has been recognized and it's overwritten when the next token is found.

Chapter 4

Semantical analysis

The main function of this part is *yysem* (this time written entirely by us, as there's no language for generating a semantical analyzer automatically) in which we look through the whole Abstract Syntax Tree, generated before, and check if there are any error types between nodes, looking for semantical errors. To support itself in this operation, the analyzer 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 during the lexical analysis, it is something entirely different, as in the lexical analyzer we have simply created an hashmap containing lexemes values of strings and identifiers, so there couldn't be any repetition of the same lexeme. [magari è già stato descritto durante l'analizi lessicale, quindi si può togliere da qui]

The Symbol Table is based on the following auto-esplicative C structures (Code 4.1):

```
1  // Structure to represent the Schema of a Symbol
2  typedef struct schema
3  {
4    TypeSchema type;
5    char* id;
6    int size;
7    struct schema* child;
8    struct schema* brother;
9  } Schema;
10
11 // Structure to represent a Symbol in the Symbol Table
12 typedef struct symtab
```

```
13   {
14     // Name of the Symbol
15     char* name;
16     // Unique identifier in this scope
17     int oid;
18     // Class of the Symbol
19     ClassSymbol clazz;
20     // Pointer to the schema of this Symbol
21     Schema* schema;
22     // Environment/scope in which this Symbol is defined
23     map_t locenv;
24     // Scope deepness of the Symbol definition
25     int nesting;
26     // Number of oids' defined in this scope
27     int last_oid;
28     // Number of formal parameters (used only with functions)
29     int formals_size;
30     // Pointer to the formal parameters (only with functions)
31     struct symtab** formals;
32   } Symbol;
```

Code 4.1: Symbol Table structure

Every symbol, depending on its type, has to be enumerated so it could be uniquely found when executing the code. We will differentiate between two possible enumerations: the first one is global, associated to every function defined in the code, and the second one is relative to the scope of every function, so we could define, for example, variables with same name, but each one belonging to different function scopes.

To make the relative enumeration possible and to check the visibility of a symbol, we have defined a stack representing the scope of the function in where we are.

Most of the semantic checks are related to the type schema compatibility between nodes in the same expression, so the first critical point is to create the correct schema of every node, in this way we will just have to check the equality of these schemas. Every time a function, a variable or a parameter, a constant and a type definition is found in the code, the Symbol Table is updated with its schema, so it will be more simply to access in the future and we don't have to recalculate its schema every time we found it.

Here we show an example of a Symbol Table (Code 4.2):

```
1   FUNC    1      prog        0    INT
2     CONST  15     PAIR        0    STRUCT( ATTR: a ( INT ), ATTR: b ( CHAR ) )
3     CONST  14     name        0    STRING
4     VAR    7      b           0    BOOL
5     VAR    1      c           0    CHAR
6     TYPE   0      T2          0    STRING
7     VAR    6      s           0    STRING
```

```
 8    VAR     12    out_v        0    VECTOR[10]( REAL )
 9    VAR     11    out_x        0    REAL
10    VAR     4     y            0    REAL
11    VAR     10    v2           0    VECTOR[100]( STRUCT( ATTR: a ( INT ), ATTR: b (
         CHAR ) ) )
12    CONST   16    VECT         0    VECTOR[5]( REAL )
13    VAR     3     x            0    REAL
14    CONST   17    MAT          0    VECTOR[2]( VECTOR[5]( REAL ) )
15    TYPE    0     from_slides  0    VECTOR[10]( STRUCT( ATTR: la ( INT ), ATTR:
         lala ( VECTOR[20]( VECTOR[5]( REAL ) ) ) ) )
16    VAR     9     v1           0    VECTOR[5]( INT )
17    FUNC    2     ref          1    INT
18      VAR     2     y            1    REAL
19      VAR     1     x            1    REAL
20      VAR     5     v            1    VECTOR[10]( REAL )
21      VAR     3     r1           1    STRUCT( ATTR: a ( INT ), ATTR: b ( STRING ) )
22      VAR     4     r2           1    STRUCT( ATTR: a ( INT ), ATTR: b ( STRING ) )
23    VAR     8     r            0    STRUCT( ATTR: a ( CHAR ), ATTR: b ( STRING ) )
24    CONST   13    MAX          0    INT
25    VAR     2     i            0    INT
26    VAR     5     z            0    REAL
```

Code 4.2: Symbol Table example

The type checking has also to be done in expressions where there aren't only variables with a known schema: in this case we have to infere the schema of the not known part and check if it is equal to the known part, or also check if two unknown schemas are compatible each other.

To create a schema we used a recursive approach, descending through the node until its leaves, which have to be one of the possible *Atomic Domain* types.

A really useful function that we decided to implement is the *infere_ lhs_ schema*, needed to infere the schema of a *lhs term* that could be an orthogonal innested composition of *indexing* and *fielding* nodes, reaching at the end an *Atomic Domain* type. Our solution for *vector* and *structures* cases is here reported (Code 4.3):

```
 1   switch( node->value.n_val )
 2   {
 3     case N_FIELDING:
 4       result = infere_lhs_schema( node->child, is_assigned );
 5       if( result->type != TS_STRUCT )
 6         yysemerror( node->child,
 7                 PRINT_ERROR( STR_CONFLICT_TYPE,
 8                     "not a struct" ) );
 9
10       result = result->child;
11       while( result != NULL )
12       {
13         if( result->id == node->child->brother->value.s_val )
```

```
14            return result -> child;
15          result = result -> brother;
16        }
17        yysemerror( node -> child -> brother ,
18              PRINT_ERROR( STR_UNDECLARED ,
19                    "not a struct attribute" ) );
20        break ;
21
22    case N_INDEXING :
23        result = infere_lhs_schema( node -> child , is_assigned );
24        if( result -> type != TS_VECTOR )
25          yysemerror( node -> child ,
26                PRINT_ERROR( STR_CONFLICT_TYPE ,
27                      "not a vector" ) );
28
29        simplify_expression( node -> child -> brother );
30        if( infere_expression_schema( node -> child -> brother ) -> type != TS_INT )
31          yysemerror( node -> child ,
32                PRINT_ERROR( STR_CONFLICT_TYPE ,
33                      "expression must be integer" ) );
34
35        result = result -> child;
36        break ;
37
38    default :
39        yysemerror( node ,
40              PRINT_ERROR( STR_BUG ,
41                    "unknown unqualified nonterminal expression" ) );
42        break ;
43  }
```

Code 4.3: Symbol Table example

As we could see in this piece of code, we are checking the integrity of types, checking that the inferred schema is compatible with the case in which we are and, if this will not happen, we stop the analysis throwing a semantical error, using the *yysemerror* function, which tries to explain the error that is occurred. We could also see that a *simplify_ expression* function is called whenever is possible, trying to simplify some a a priori computational parts, such as mathematical and logical operations with known values.

I'm going to list now some relevant type checks that a user should know to correctly use SOL language:

- relational expressions work only with *Boolean* types;

- *in* statement requires could only be applied to a *vector;*

- $<$, $\leq$, $>$, $\geq$ could be applied only on *char*, *int*, *real* and *string* types, not to composition nor structured types;

- mathematical expressions work only with numbers, both *integers* and *reals;*

- *toint* and *toreal* statements work only on their opposite types, correspondingly *reals* and *integers;*

- assignment work only with *parameters* or *variables;*

- the iterative variable of the *for* loop cannot be re-assigned inside the cycle itself.

Our implementation of this last check is a little tricky, (we mean, yo dawg) because we temporary changed the type of the iterative variable to a *constant,* so per definition, it's not possible to change its value.

Chapter 5

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 with 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).

5.1   S-code specification

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

Everything is done on three global stacks[1]: the *activation* stack, the *object* stack and the *instance* stack. The activation stack contains the function's activation records, describing a function's local environment, and an activation record is added on the stack at the moment of a function's call. An activation record contains a reference to the starting point of that function's objects on the object stack. The object stack contains object descriptors, each of which describing a single object with its size and value, or a reference to the position of the value on the instance stack if that's the memorization mode of the object. The instance stack contains instances of the objects.

---

[1]More details on the stacks are provided in Chapter ??.

Temporary values, such as partial results of an expression, are put on the instance stack and referred to through a temporary object on the object stack.

Every instruction has from 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 integers is called *IPLUS* and it has no operands: it takes the last two values present on the stack, sum them and put the result back on the stack. Obviously, as a standard procedure when using stacks, every value used is also consumed.

Note that the object stack contains an environment associated to every called function, and this environment can be divided in two parts: one "permanent" containing the objects defined in the function's header, and one "temporary" containing the objects used by the S-code instructions. The instructions that need to use the value of an object in the permanent part (such as an expression involving variables) make use of the instruction *LOD* to copy that variable's value on top of the stack (or a composition of multiple instructions to refer to a struct's field or a position in a vector) and, then, use the copied temporary value instead of the permanent one. In the same way, an assignment is performed by calculating the assigned value on the top of the stack and, then, copying it in the permanent part with the instruction *STO*.

5.2   The yygen function

When the *yygen* function is called, it automatically retrieves the root of the Syntax Tree and passes it to the *generate_code* function. 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 instructions, see Code 5.1 for the structures definition) 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).

```
1  typedef struct code {
2    Stat* head;
3    int size;
4    Stat* tail;
5  } Code;
```

```
 6
 7  typedef struct stat {
 8    int address;
 9    Operator op;
10    Lexval args[ MAX_ARGS ];
11    struct stat* next;
12  } Stat;
```

Code 5.1: Code and Stat structures

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

The code is represented, as can be deduced by the structure definition in Code 5.1, as a list of S-code statements, each of which is represented with its address (number of line), operator, an array of arguments (with a maximum number of 3 arguments) and a pointer to the next instruction.

At the end of the code generation, the instructions are printed to a file with extension *ohana* (name funnily derived from the *sol* extension, because sol..solo..han..han solo..ohana :D) using a function called *output_code*.

5.2.1 Function problem

The generation of the code for function declarations and calls caused some problems because the call (which is translated as in Code 5.2) needs informations that can only be given after its code generation is done, but knowing that, for example, a function is allowed to call itself, the two code generations collide.

```
1  PUSH   <number of formal parameters>
2      <number of local parameters>
3      <distance between the call environment and the definition one>
4  GOTO <entry point of function in S-code>
5  POP
```

Code 5.2: S-code of a function call

Assuming that a function will only be called after it is defined, we decided to build a hashmap in which every function will put informations about itself at the time of its definition. This informations are the nesting of the environment in which it is defined and a reference to the *Stat* containing its first statement (that is, the instantiation of its first parameter, if the parameters are present). The hashmap also contains the number of objects defined in the function's environment, but this information can only be determined at the end of the function's body computation (the temporary variables for

for cycles, for example, are part of the function's environment but are not defined in the header)[toglierei tutta la ()]. The hashmap uses the function's oid as key.

When the code for a function call has to be generated, it retrieves the hashmap entry relative to the called function and it generates the instructions PUSH, GOTO and POP. At the moment of the call we can only be sure about the correctness of the second argument of PUSH (computed as actual nesting - definition nesting, the latter retrieved from the hasmap), therefore, the other two arguments are set as 0. At the end of the call, a new entry is put in a *stacklist,* containing the function's oid and a copy of the *Code* generated for the call (thus containing a pointer to the PUSH, GOTO and POP statements).

When the whole code has been generated, we process the entries in the call stacklist. Since now every function definition has been processed in full and the whole code has been produced, every entry in the hashmap will contain for sure the correct informations about the number of objects in the functions' environments, and the pointer to the first statement of every function will feature the correct code address. Therefore, for every entry in the stack we can retrieve the corresponding function descriptor (thanks to the oid) and substitute the first arguments of PUSH and GOTO with the right values.

5.2.2   Instantiation of temporary variables

While executing some examples, we have found that the *istack* wasn't empty at the end of the execution, but it still contains a lot of instances. We also noticed that this problem is code-dependent, in particular depending on the fact that the code that we are executing contains or not any sort of cycle. Looking at the generated code we have found that, every time a cycle were in an example, a lot of instances were allocated in the *istack* due to the fact that the temporary variables were allocated every time a cycle was used. The problem will get stronger when the examples contained some nested cycles.

Our solution for this issue was to separate the execution from the variable instantiation code, saving those two parts separately and then appending the code for the variable instantiation after the the function variable instantiation, so every variable, temporary or not, defined in the whole function will be allocated only once.

Part III

The Virtual Machine

Chapter 6

Structure of the virtual machine

The Virtual Machine is built as a standalone program. This means that it is not directly fed with the compiled code, but it has to read it from a *ohana* file produced by the compiler. The file is read using LEX and YACC, and the Code structure originally generated by *yygen* is rebuilt inside the virtual machine, this time simplified in a vector of *Stat* structures. Then, the function *yyvm* is called. This function takes the code, saved in the global variable *program*, and executes it statement by statement.

Being the *program* variable a vector, the *yyvm* function can simply iterate over its elements and execute each one of them using the function *execute*, implemented as a big *switch* over the statement's instruction that calls the appropriate execution function in correspondance of each instruction. The iteration is done using a global counting variable *pc*. Using a global variable that indicates the actual statement may seem very bad, but it allows us to perform a jump in the code (needed, since all the conditional constructs, loops and even function calls are performed via conditioned or unconditioned jumps) simply by changing the value of that variable.

6.1 Virtual machine structures

As anticipated in Chapter 5, S-code is a language designed to be executed with the support of three global stacks. In our virtual machine, these stacks are called *astack*, *ostack* and *istack* and are implemented as vectors of, respectively, pointers to *Adescr*, pointers to *Odescr* and *byte*s (which are simply chars, redefined for clarity with a *#define*). The implementation as vectors

allows for a simpler handling of the allocation, deallocation and reference of elements (the latter is done simply by recording the index of the referenced element).

All the instances are recorded as arrays of bytes.

The stacks, their base element's structures and a number of methods to interact with them are all defined in a file separated from the one defining the execution methods, called *support_structures.h*. Each stack has the method *top* to access its last element (easily done considering that there are three global variables, *ap*, *op* and *ip*, that reference to the first empty position in each stack), the *astack* and *ostack* have *pop* and *push* methods while the *istack* has two particular methods to perform the same thing on multiple entries, called *allocate_istack* and *deallocate_istack*. The different in the approach is useful because, while the first two stacks are normally required to allocate/deallocate one element at a time, it is almost always the case that the *istack* needs to allocate multiple elements (eg if I want to put an integer on the stack, I will need four bytes or more, therefore I will have to allocate four elements).

Since the *ostack* is required to allocate a fixed number of objects at the moment of a function call, there is also a method to perform such task called *enlarge_ostack*.

The global variables *asize*, *osize* and *isize* are used to keep track of the stacks' size and simplify their handling (eg check if a stack is full and needs to be reallocated in correspondance of a push).

The *Adescr* and *Odescr* structures are defined as in Code 6.1.

```
1  typedef struct {
2    // Modality of saving of the object, either embedded or in the instance
         stack
3    Mode mode;
4    // Size of the object in bytes
5    int size;
6    // Value
7    ObjectVal inst;
8  } Odescr;
9
10 typedef struct {
11   // Number of objects contained in the activation record
12   int obj_number;
13   // Pointer to the first object of the activation record in object_stack
14   int first_object;
15   // Address were to return
16   int raddr;
17   // Address of the father (definition) in the astack
18   int alink;
```

`} Adescr;`

Code 6.1: Adescr and Odescr objects

An *Adescr* represents the activation record of a function, and is created and pushed on the *astack* every time a function is called. It contains informations on the function's local environment along with the reference to statement at which to return when the function's execution is done.

An *Odescr* represents an object and contains its instance's size and a reference to such instance. The instance can be saved in two "modes":

- **Embedded mode**: the *inst* field of the object is an array of bytes containing the instance;

- **Stack mode**: the *inst* field of the object is an integer referencing the position of the *istack* containing the first byte of the instance;

It's obvious that the stack mode will be preferred in the case of complex objects (structs and vectors), while the embedded mode is normally used for simple objects. All the "temporary" objects, simple or complex, are created in stack mode.

Moreover, a number of "mask" methods is used to push and pop temporary values of specific type on the stacks. These methods take care of splitting the values into bytes in the right way during the push and putting them back togheter at the moment of pop. These methods also create objects on the *ostack* to reference the temporary values put on the *istack*. There is one of them for every elementary type of SOL (int, real, string, char, booleans are treated as chars) and they rely on the methods *push_ bytearray* and *pop_ bytearray*.

6.2 Example of execution method[1]

Given the premises of the previous section, the general execution scheme is pretty simple: access to temporary values is done via the pop/push mask functions, and access to objects is done by computing the object's index on the *ostack* starting from its environment offset (the definition environment

---

[1]Only one example is reported because the details of all the instructions' execution methods are not particularly interesting and the code is heavily commented, if one would like to inspect it.

of the object is retrieved on the *astack* by iterating over the *alink* of the activation records) and applying the object's oid as index within the activation record's list of objects.

The execution procedure for a function's call is explained in the following subsection, as an example and because more complex than the others.

6.2.1 Function call execution

A function call, as we can recall from Section 5.2.1, is composed of three separate instructions: *PUSH*, *GOTO* and *POP*. The execution methods for these instructions is reported in Code 6.2.

```
1  // Push the chain and element_number on the istack, in preparation of the
       call to GOTO, and instantiate a new activation record
2  int sol_push( Value* args )
3  {
4    int formals_size = args[ 0 ].i_val;
5    int locals_size = args[ 1 ].i_val;
6    int chain = args[ 2 ].i_val;
7
8    enlarge_ostack( locals_size );
9
10   push_int( formals_size );
11   push_int( locals_size );
12
13  #ifdef DEBUG
14    fprintf( stderr, "SOL pushed el#: %d, %d\n", formals_size, locals_size );
15  #endif
16
17    push_int( chain );
18
19  #ifdef DEBUG
20    fprintf( stderr, "SOL pushed chain: %d\n", chain );
21  #endif
22
23    return MEM_OK;
24  }
25
26  // GOTO is used ONLY after a push, to perform a function call
27  int sol_goto( Value* args )
28  {
29    int entry_point = args[ 0 ].i_val;
30    int chain = pop_int();
31    int locals_size = pop_int();
32    int formals_size = pop_int();
33
34    Adescr* function_ar;
35
36  #ifdef DEBUG
37    fprintf( stderr, "SOL goto chain: %d\n", chain );
38    fprintf( stderr, "SOL goto el#: %d, %d\n", formals_size, locals_size );
```

```
39   #endif
40
41     // The number of elements is given, the start point for its objects is the
42     // top of the stack (the objects will be instantiated as part of the
43     // function call, not before)
44     function_ar = malloc( sizeof( Adescr ) );
45     function_ar->obj_number = formals_size + locals_size;
46     function_ar->first_object = op - formals_size;
47     function_ar->raddr = pc + 1;
48     function_ar->alink = ap - 1;
49
50     while( chain-- > 0 )
51       function_ar->alink = astack[ function_ar->alink ]->alink;
52
53     push_astack( function_ar );
54
55     // Jump to the entry point (first instruction will be the definition of
         the formals)
56     pc = entry_point - 1;
57
58     return MEM_OK;
59   }
60
61   // Clean the stacks after the last function call
62   int sol_pop()
63   {
64     int i;
65     ByteArray function_result = pop_bytearray();
66
67     for( i = 0; i < top_astack()->obj_number; i++ )
68     {
69       // All the instances of the current environment are on top of the
70       // istack, all I care about is to pop the correct total number of
71       // cells, not the exact cells for every object
72       if( top_ostack()->mode == STA )
73         deallocate_istack( top_ostack()->size );
74
75       pop_ostack();
76     }
77
78     pop_astack();
79
80     // Restores the result obtained from the called function
81     push_bytearray( function_result.value, function_result.size );
82
83     return MEM_OK;
84   }
```

Code 6.2: Execution of a function call

Essentially, the *PUSH* instruction pushes its arguments on the stack. In addition to this, it calls the function *enlarge_ ostack* to allocate enough space on the stack to accomodate all the objects in the function's local environment, avoiding the need for a realloc at every object push when the function's header code is executed (that code will contain a *NEW* or *NEWS* instruction

for every parameter, variable and constant of the function; those instructions simply perform a push on the *ostack*, and this causes the stack to be reallocated by the size one element if it is full; the enlargement performed by *PUSH* prevents these multiple reallocations).

The *GOTO* instruction retrieves the values put on the stack by the *PUSH* and creates a new activation record with these informations and puts it on the *astack*. Note that the *raddr* field is, simply, a reference to the statement that follows the *PUSH*. After that, it performs a jump to the first instruction of the function.

Finally, the *POP* clears the entries of the local environment from the three stacks, taking care of putting the function's return value back on the stack.

Chapter 7

Input-Output handling

7.1 Interprocess communication

The communication between the virtual machine end the input-output re-
sources is characterized by two kinds of data: a string containing a compact
representation of the schema and an array of bytes containing the raw data
to be passed from one side to the other.

The textual representation of the schema follows the simple EBNF rules
shown in (7.1.1) and can be easily created and parsed with a recursive switch
function.

$$
\begin{aligned}
\textit{format} \quad &\rightarrow \quad \textit{atomic-format} \,|\, \textit{struct-format} \,|\, \textit{vector-format} \\
\textit{atomic-format} \quad &\rightarrow \quad \mathbf{c} \,|\, \mathbf{i} \,|\, \mathbf{r} \,|\, \mathbf{s} \,|\, \mathbf{b} \\
\textit{struct-format} \quad &\rightarrow \quad (\textit{attr} \,\{,\textit{attr}\}\,) \\
\textit{attr} \quad &\rightarrow \quad \mathbf{id:}\textit{format} \\
\textit{vector-format} \quad &\rightarrow \quad [\mathbf{num},\textit{format}]
\end{aligned}
\tag{7.1.1}
$$

The raw data matches the representation used inside the virtual machine's
stack, with the only difference that the string pointers are replaced with the
real content of the string, terminator included. Knowing the size of each
element or, in the case of strings, having a known terminator makes it easy
to pack and unpack the data from this representation to the needed one.

7.2 File system interface

When reading from a file, the request contains just the filename and returns the raw data, which is then parsed and translated into stack-acceptable data. If the file doesn't contain the expected data or doesn't exist, the virtual machine stops and a segmentation fault error is given to the user.

When writing to a file, the request contains the filename and the raw data to write.

7.3 Graphical interface

The virtual machine uses a graphical interface realized with the Qt5 graphical environment for the interaction of the program with the user (that is, the commands *READ*, *WRITE*, *RD* and *WR* make calls to the GUI).

All the graphical part is realized and managed in Python3 using the *ui* files created by the Qt5 Designer editor and is divided in two sub-scripts.

The first script, *solGUI.py*, simplifies the user's interaction with both the compiler and the virtual machine, allowing to open a file, compile it (if the opened file is a SOL source file) and execute it (if the opened file is an S-Code file or a SOL source file which has been compiled with the GUI); while performing these actions, *solGUI* always redirects the called executable's textual output to a text-box. The window appearing at the call of solGUI is shown in Figures 7.3.1 and 7.3.2; since in OS X and many recent operative systems the menu bar has its own dedicated space on the top of the screen, a particular of the *File* menu is shown in Figures 7.3.3 and 7.3.4.

The second script is called directly from within the virtual machine and shows the user specialized dialogs representing the given data's schema. If the intention was to show the user some data, the request contains also the raw data and the fields become read-only. Examples of the interface in these cases is shown in Figures 7.3.5 and 7.3.6 for data input and 7.3.7 and 7.3.8 for data output.

These dialogs are constructed through a simple automatic composition of widgets, done by the Python code. The dialog widgets are in fact divided in a widget for every type available in SOL plus a widget dedicated to the nesting of complex types. This widget is used when the represented type has, in the schema, a nesting level greater than 1 (e.g., a vector of vector of structs becomes, in the dialog, a matrix of nested widgets, each of which describing
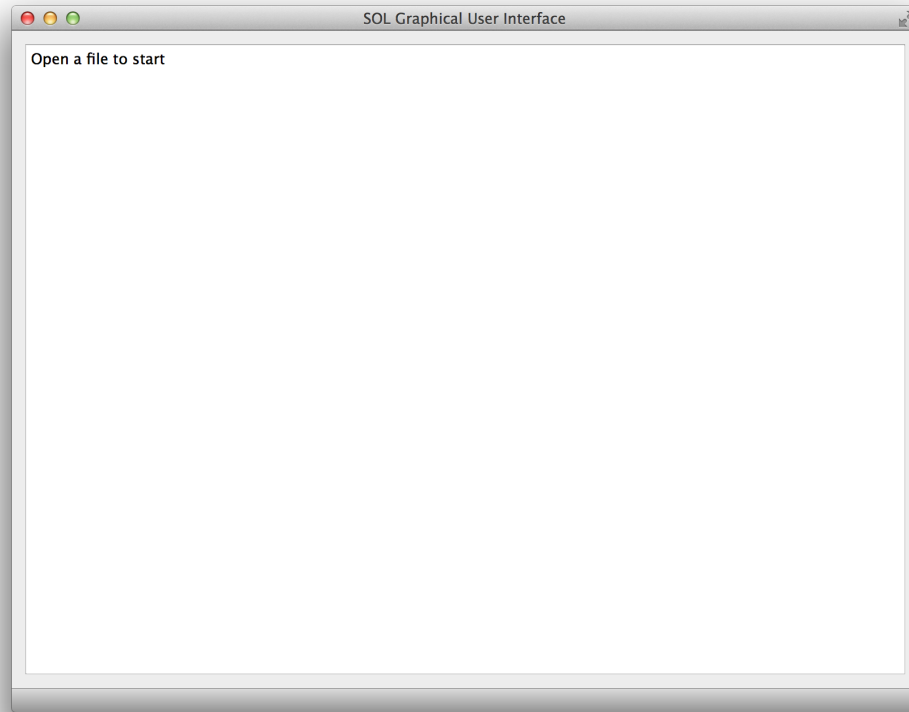
Figure 7.3.1: The main window of the GUI in OS X

the struct). This allows a generally simple and understandable representation of every type combination, although problems could arise in cases like three-dimensional matrixes, which become a matrix of nested widgets when the programmer might have meant to represent a vector of matrixes (hence a vector of nested widgets).

The composition of widgets is done easily thanks to the use of polymorphism and a method designed to translate the textual representation of the schema into the appropriate widgets. The method reads the first symbol of the schema and returns the appropriate instance of the class *DataWidget*, which will eventually parse a part of the schema to construct itself (e.g., the *VectorWidget* reads the number of elements) and then call once more the translation method, to get the widgets relative to its children. The code of the translation function is shown in Algorithm 7.1.

Figure 7.3.2: The main window of the GUI in Ubuntu 14.04
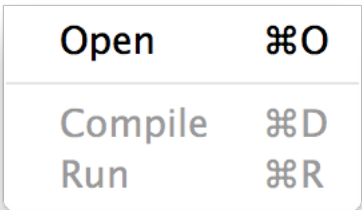

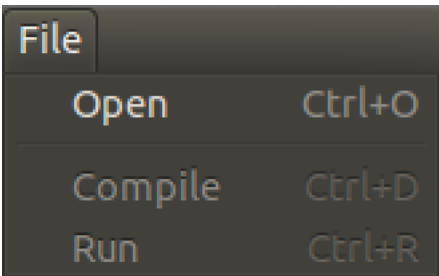Figure 7.3.3: The File menu of the GUI on OS X

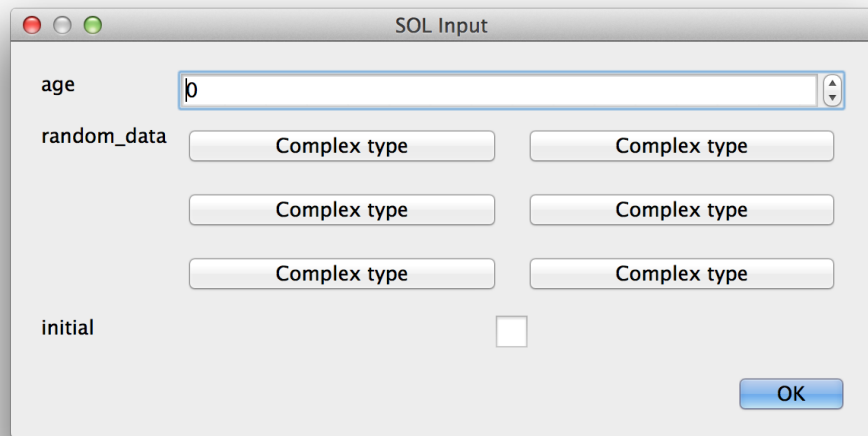Figure 7.3.4: The File menu of the GUI on Ubuntu 14.04

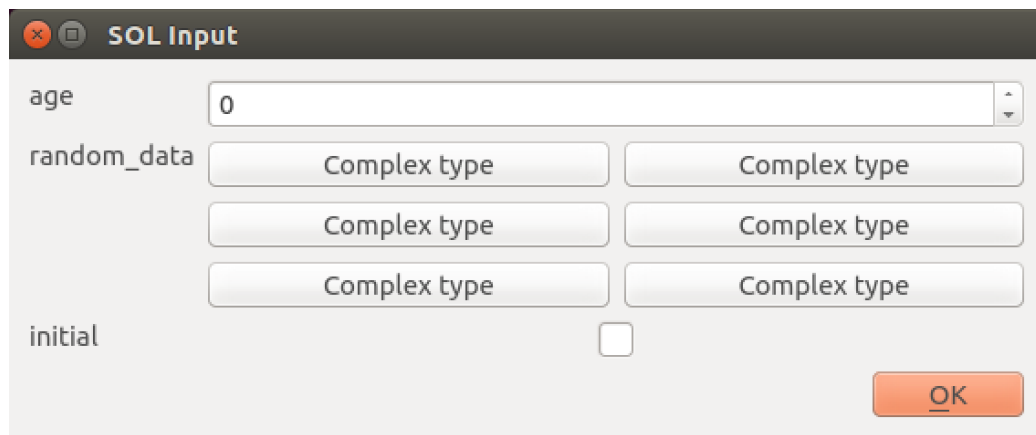Figure 7.3.5: An example of an input dialog of the GUI on OS X

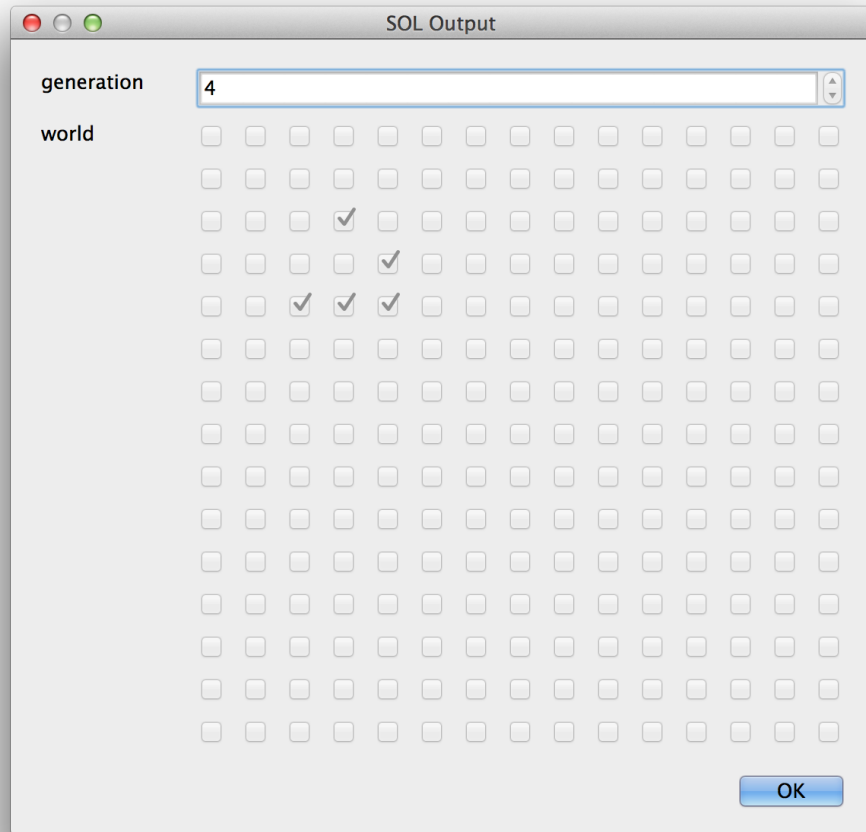Figure 7.3.6: An example of an input dialog of the GUI on Ubuntu 14.04

Figure 7.3.7: An example (taken from the execution of Algorithm 1.14) of an output dialog of the GUI on OS X
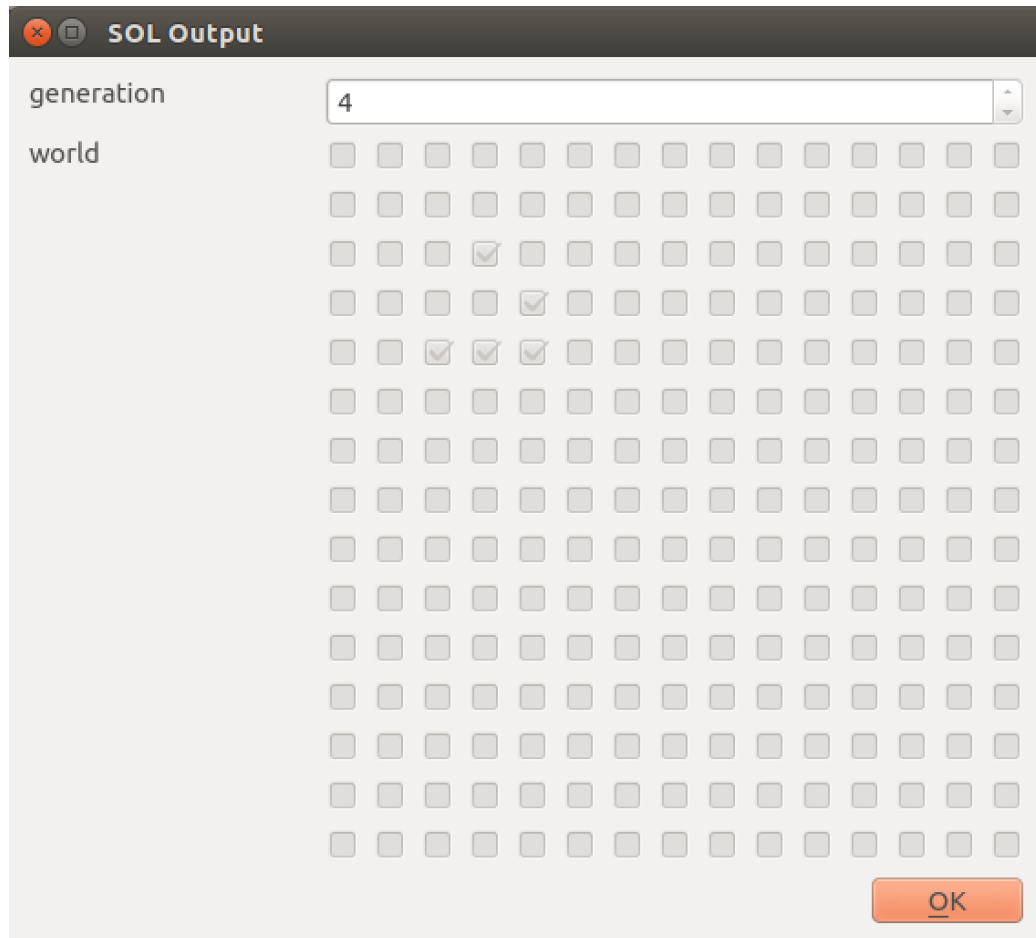
Figure 7.3.8: An example (taken from the execution of Algorithm 1.14) of an output dialog of the GUI on Ubuntu 14.04

Algorithm 7.1 The method used to translate the textual schema into the GUI widget

```python
@staticmethod
def resolveSchema(schema, nesting=None, editable=True):
    """
    Transforms a string schema into a widget (or series of nested
    widgets)
    """
    if len(schema) == 0:
        raise IndexError()

    if nesting is None:
        nesting = {'s': 0, 'v': 0}

    element = schema.popleft()
    if element == "i":
        return IntegerWidget().setEditable(editable)
    elif element == "r":
        return RealWidget().setEditable(editable)
    elif element == "c":
        return CharacterWidget().setEditable(editable)
    elif element == "s":
        return StringWidget().setEditable(editable)
    elif element == "b":
        return BooleanWidget().setEditable(editable)
    elif element == "[":
        if nesting['s'] > 1 or nesting['v'] > 1:
            schema.appendleft(element)
            return NestedWidget(schema, editable)
        else:
            return VectorWidget(schema, dict(nesting), editable)
    elif element == "(":
        if nesting['s'] > 1 or nesting['v'] > 1:
            schema.appendleft(element)
            return NestedWidget(schema, editable)
        else:
            return StructWidget(schema, dict(nesting), editable)
    else:
        return DataDialog.resolveSchema(schema, nesting, editable)
```

Part IV

Conclusions and examples

Chapter 8

Conclusions

It works. Whatelse?

Appendix A

Examples of programs with compiled S-code

Here we report some program example written in SOL, along with their compiled S-code code, created while implementing the whole project.

A.1 Hello, World!

The *.sol* code could be seen in the tutorial section (Code 1.1), instead here we report it S-code (Code A.1).

```
1   SCODE 9
2   PUSH 0 0 -1
3   GOTO 4
4   POP
5   HALT
6   FUNC 1
7   LDS "Hello world!"
8   WRITE "s"
9   LDI 0
10  RETURN
```

Code A.1: Hello world S code

A.2 Permutations

This example calculates the permutations of a given set of characters, which are inserted by the user, in both recursive and iterative modes.

SOL code A.2

```
1   func main() : int
2
3     type
4       pseudo_string: vector[ 15 ] of char;
5       permutation: struct( str: pseudo_string; perm: int; );
6
7     var
8       i: int;
9       p: permutation;
10      word: pseudo_string;
11
12    const
13      MAX_LEN: int = 15;
14      INTRO: string = "Insert a word:";
15
16    func new_pseudo_string() : pseudo_string
17      var
18        result: pseudo_string;
19        i: int;
20  begin new_pseudo_string
21      for i = 0 to MAX_LEN - 1 do
22        result[ i ] = '\0';
23      endfor;
24      return result;
25    end new_pseudo_string
26
27    func strlen( s: pseudo_string; ) : int
28      var i: int;
29  begin strlen
30      for i = 0 to MAX_LEN - 1 do
31        if s[ i ] == '\0' then
32          break;
33        endif;
34      endfor;
35      return i;
36    end strlen
37
38    func strcat( s1, s2: pseudo_string; ) : pseudo_string
39      var
40        result, s: pseudo_string;
41        c: char;
42  begin strcat
43      if strlen( s1 ) + strlen( s2 ) <= MAX_LEN then
44        foreach s in vector( s1, s2 ) do
45          foreach c in s do
46            result[ strlen( result ) ] = c;
47          endforeach;
48        endforeach;
49      endif;
50
51      return result;
52    end strcat
53
54    func strcpy( str: pseudo_string; ) : pseudo_string
```

```
55        var
56          result: pseudo_string;
57          i: int;
58     begin strcpy
59        for i = 0 to strlen( str ) - 1 do
60          result[ i ] = str[ i ];
61        endfor;
62        return result;
63     end strcpy
64
65     func recursive_factorial( number: int; ) : int
66     begin recursive_factorial
67        if number <= 2 then
68          return number;
69        endif;
70        return number * recursive_factorial( number - 1 );
71     end recursive_factorial
72
73     func recursive_permutation( to_process: pseudo_string;
74                                 base: pseudo_string;
75                                 number: int; ) : pseudo_string
76        var
77          next_step, current_base, result, temp: pseudo_string;
78          i, j: int;
79
80     begin recursive_permutation
81        -- Returning if i have only one char to process
82        if strlen( to_process ) == 1 then
83          write vector( struct( "Recursive permutation:",
84                                number + 1,
85                                strcat( base, to_process ) ) );
86          return strcat( base, to_process );
87        endif;
88
89        j = 0;
90        temp = new_pseudo_string();
91        while j < strlen( to_process ) do
92
93          -- Resetting base to the argument value and resetting next_step
94          current_base = strcpy( base );
95          next_step = new_pseudo_string();
96          -- Creating the new string to pass as argument
97          for i = 0 to strlen( to_process ) - 1 do
98            temp[ 0 ] = to_process[ i ];
99            if i != j then
100               next_step = strcat( next_step, temp );
101             else
102               current_base = strcat( current_base, temp );
103             endif;
104          endfor;
105
106          -- Decrementing the size of the permutations
107          result = recursive_permutation( next_step, current_base, number );
108          number = number + recursive_factorial( strlen( next_step ) );
109
110          j = j + 1;
111        endwhile;
```

```
112
113        -- Completly useless
114      return result;
115    end recursive_permutation
116
117    func factorial( i: int; ) : int
118      var result: int;
119    begin factorial
120      result = 1;
121      for i = 2 to i do
122        result = result * i;
123      endfor;
124
125      return result;
126    end factorial
127
128    func ceil( r: real; ) : int
129      var result: int;
130    begin ceil
131      result = toint( r );
132      if toreal( result ) > r then
133        result = result - 1;
134      endif;
135
136      return result;
137    end ceil
138
139    func floor( r: real; ) : int
140      var result: int;
141    begin floor
142      result = toint( r );
143      if toreal( result ) < r then
144        result = result + 1;
145      endif;
146
147      return result;
148    end floor
149
150    func mod( a, b: int; ) : int
151    begin mod
152      return a - ceil( toreal( a ) / toreal( b ) ) * b;
153    end mod
154
155    func is_even( a: int; ) : bool
156    begin is_even
157      return mod( a, 2 ) == 0;
158    end is_even
159
160    func xor( a, b: bool; ) : bool
161    begin xor
162      return ( a and not b ) or ( not a and b );
163    end xor
164
165    func next_permutation( p: permutation; ) : permutation
166      var
167        result: permutation;
168        len, i: int;
```

```
169
170      func circ_shift( s: pseudo_string; a, b: int; ) : pseudo_string
171        var
172          i: int;
173          temp: char;
174      begin circ_shift
175        for i = a to b-1 do
176          temp = s[ i ];
177          s[ i ] = s[ i + 1 ];
178          s[ i + 1 ] = temp;
179        endfor;
180        return s;
181      end circ_shift
182
183      func flip( s: pseudo_string; a, b: int; ) : pseudo_string
184        var
185          i: int;
186          temp: char;
187      begin flip
188        for i = 0 to floor( toreal( b - a ) / 2.0 ) - 1 do
189          temp = s[ a + i ];
190          s[ a + i ] = s[ b - i ];
191          s[ b - i ] = temp;
192        endfor;
193        return s;
194      end flip
195
196    begin next_permutation
197      len = strlen( p.str );
198
199      p.perm = p.perm + 1;
200      if p.perm == factorial( len ) then
201        p.perm = 0;
202        p.str = flip( p.str, 0, len - 1 );
203
204        return p;
205      endif;
206
207      for i = 2 to len do
208        if mod( p.perm, factorial( i ) ) != 0 then
209          p.str = circ_shift( flip( p.str, len - i + 1, len - 1 ),
210                              len - i,
211                              len - 1 );
212          break;
213        endif;
214      endfor;
215
216      return p;
217    end next_permutation
218
219 begin main
220   write INTRO;
221   p.str = rd pseudo_string;
222   word = p.str;
223
224   write vector( struct( "Word:", word ) );
225
```

```
226    -- Iterative permutations
227    for i = 0 to factorial( strlen( p.str ) ) - 1 do
228      p = wr next_permutation( p );
229    endfor;
230
231    -- Recursive permutations
232    word = recursive_permutation( word, new_pseudo_string(), 0 );
233    return 0;
234  end main
```

Code A.2: SOL code of Permutations

S Code A.3

```
1   SCODE 666
2   PUSH 0 7 -1
3   GOTO 4
4   POP
5   HALT
6   FUNC 1
7   NEW 4
8   NEWS 19
9   NEWS 15
10  NEWS 15
11  NEW 4
12  NEW 8
13  LDI 15
14  STO 0 5
15  LDS "Insert a word:"
16  STO 0 6
17  NEW 4
18  LOD 0 6
19  WRITE "s"
20  LDA 0 2
21  RD "[15,c]"
22  IST
23  LDA 0 2
24  EIL 15
25  STO 0 3
26  LDS "Word:"
27  LOD 0 3
28  CAT 2 23
29  CAT 1 23
30  WRITE "[1,(:s,:[15,c])]"
31  LDA 0 2
32  EIL 15
33  PUSH 1 2 0
34  GOTO 95
35  POP
36  PUSH 1 2 0
37  GOTO 371
38  POP
39  LDI 1
40  IMINUS
41  STO 0 7
42  LDI 0
```

```
43   STO 0 1
44   LOD 0 1
45   LOD 0 7
46   ILE
47   JMF 12
48   LOD 0 2
49   PUSH 1 4 0
50   GOTO 469
51   POP
52   WR "(str:[15,c],perm:i)"
53   STO 0 2
54   LOD 0 1
55   LDI 1
56   IPLUS
57   STO 0 1
58   JMP -14
59   LOD 0 3
60   PUSH 0 3 0
61   GOTO 69
62   POP
63   LDI 0
64   PUSH 3 7 0
65   GOTO 248
66   POP
67   STO 0 3
68   LDI 0
69   RETURN
70   FUNC 2
71   NEWS 15
72   NEW 4
73   NEW 4
74   LOD 1 5
75   LDI 1
76   IMINUS
77   STO 0 3
78   LDI 0
79   STO 0 2
80   LOD 0 2
81   LOD 0 3
82   ILE
83   JMF 11
84   LDA 0 1
85   LOD 0 2
86   IXA 1
87   LDC '\0'
88   IST
89   LOD 0 2
90   LDI 1
91   IPLUS
92   STO 0 2
93   JMP -13
94   LOD 0 1
95   RETURN
96   FUNC 3
97   NEW 4
98   NEW 4
99   LOD 1 5
```

```
100    LDI  1
101    IMINUS
102    STO  0 3
103    LDI  0
104    STO  0 2
105    LOD  0 2
106    LOD  0 3
107    ILE
108    JMF  15
109    LDA  0 1
110    LOD  0 2
111    IXA  1
112    EIL  1
113    LDC  '\0'
114    EQU
115    JMF  3
116    JMP  7
117    JMP  1
118    LOD  0 2
119    LDI  1
120    IPLUS
121    STO  0 2
122    JMP  -17
123    LOD  0 2
124    RETURN
125    FUNC 4
126    NEWS 15
127    NEWS 15
128    NEW  1
129    NEWS 30
130    NEW  4
131    NEW  4
132    NEWS 15
133    NEW  4
134    NEW  4
135    LOD  0 1
136    PUSH 1 2 1
137    GOTO 95
138    POP
139    LOD  0 2
140    PUSH 1 2 1
141    GOTO 95
142    POP
143    IPLUS
144    LOD  1 5
145    ILE
146    JMF  52
147    LOD  0 1
148    LOD  0 2
149    CAT  2 30
150    STO  0 6
151    LDI  1
152    STO  0 8
153    LDI  0
154    STO  0 7
155    LOD  0 7
156    LOD  0 8
```

```
157   ILE
158   JMF 39
159   LDA 0 6
160   LOD 0 7
161   IXA 15
162   EIL 15
163   STO 0 4
164   LOD 0 4
165   STO 0 9
166   LDI 14
167   STO 0 11
168   LDI 0
169   STO 0 10
170   LOD 0 10
171   LOD 0 11
172   ILE
173   JMF 19
174   LDA 0 9
175   LOD 0 10
176   IXA 1
177   EIL 1
178   STO 0 5
179   LDA 0 3
180   LOD 0 3
181   PUSH 1 2 1
182   GOTO 95
183   POP
184   IXA 1
185   LOD 0 5
186   IST
187   LOD 0 10
188   LDI 1
189   IPLUS
190   STO 0 10
191   JMP -21
192   LOD 0 7
193   LDI 1
194   IPLUS
195   STO 0 7
196   JMP -41
197   JMP 1
198   LOD 0 3
199   RETURN
200   FUNC 5
201   NEWS 15
202   NEW 4
203   NEW 4
204   LOD 0 1
205   PUSH 1 2 1
206   GOTO 95
207   POP
208   LDI 1
209   IMINUS
210   STO 0 4
211   LDI 0
212   STO 0 3
213   LOD 0 3
```

```
214   LOD  0 4
215   ILE
216   JMF  14
217   LDA  0 2
218   LOD  0 3
219   IXA  1
220   LDA  0 1
221   LOD  0 3
222   IXA  1
223   EIL  1
224   IST
225   LOD  0 3
226   LDI  1
227   IPLUS
228   STO  0 3
229   JMP  -16
230   LOD  0 2
231   RETURN
232   FUNC 6
233   LOD  0 1
234   LDI  2
235   ILE
236   JMF  4
237   LOD  0 1
238   RETURN
239   JMP  1
240   LOD  0 1
241   LOD  0 1
242   LDI  1
243   IMINUS
244   PUSH 1 0 1
245   GOTO 231
246   POP
247   ITIMES
248   RETURN
249   FUNC 7
250   NEWS 15
251   NEWS 15
252   NEWS 15
253   NEWS 15
254   NEW  4
255   NEW  4
256   NEW  4
257   LOD  0 1
258   PUSH 1 2 1
259   GOTO 95
260   POP
261   LDI  1
262   EQU
263   JMF  20
264   LDS  "Recursive  permutation:"
265   LOD  0 3
266   LDI  1
267   IPLUS
268   LOD  0 2
269   LOD  0 1
270   PUSH 2 9 1
```

```
271   GOTO 124
272   POP
273   CAT 3 27
274   CAT 1 27
275   WRITE "[1,(:s,:i,:[15,c])]"
276   LOD 0 2
277   LOD 0 1
278   PUSH 2 9 1
279   GOTO 124
280   POP
281   RETURN
282   JMP 1
283   LDI 0
284   STO 0 9
285   PUSH 0 3 1
286   GOTO 69
287   POP
288   STO 0 7
289   LOD 0 9
290   LOD 0 1
291   PUSH 1 2 1
292   GOTO 95
293   POP
294   ILT
295   JMF 75
296   LOD 0 2
297   PUSH 1 3 1
298   GOTO 199
299   POP
300   STO 0 5
301   PUSH 0 3 1
302   GOTO 69
303   POP
304   STO 0 4
305   LOD 0 1
306   PUSH 1 2 1
307   GOTO 95
308   POP
309   LDI 1
310   IMINUS
311   STO 0 10
312   LDI 0
313   STO 0 8
314   LOD 0 8
315   LOD 0 10
316   ILE
317   JMF 31
318   LDA 0 7
319   LDI 0
320   IXA 1
321   LDA 0 1
322   LOD 0 8
323   IXA 1
324   EIL 1
325   IST
326   LOD 0 8
327   LOD 0 9
```

```
328    NEQ
329    JMF 8
330    LOD 0 4
331    LOD 0 7
332    PUSH 2 9 1
333    GOTO 124
334    POP
335    STO 0 4
336    JMP 7
337    LOD 0 5
338    LOD 0 7
339    PUSH 2 9 1
340    GOTO 124
341    POP
342    STO 0 5
343    LOD 0 8
344    LDI 1
345    IPLUS
346    STO 0 8
347    JMP -33
348    LOD 0 4
349    LOD 0 5
350    LOD 0 3
351    PUSH 3 7 1
352    GOTO 248
353    POP
354    STO 0 6
355    LOD 0 3
356    LOD 0 4
357    PUSH 1 2 1
358    GOTO 95
359    POP
360    PUSH 1 0 1
361    GOTO 231
362    POP
363    IPLUS
364    STO 0 3
365    LOD 0 9
366    LDI 1
367    IPLUS
368    STO 0 9
369    JMP -80
370    LOD 0 6
371    RETURN
372    FUNC 8
373    NEW 4
374    NEW 4
375    LDI 1
376    STO 0 2
377    LOD 0 1
378    STO 0 3
379    LDI 2
380    STO 0 1
381    LOD 0 1
382    LOD 0 3
383    ILE
384    JMF 10
```

```
385   LOD  0 2
386   LOD  0 1
387   ITIMES
388   STO  0 2
389   LOD  0 1
390   LDI  1
391   IPLUS
392   STO  0 1
393   JMP  -12
394   LOD  0 2
395   RETURN
396   FUNC 9
397   NEW  4
398   LOD  0 1
399   TOINT
400   STO  0 2
401   LOD  0 2
402   TOREAL
403   LOD  0 1
404   RGT
405   JMF  6
406   LOD  0 2
407   LDI  1
408   IMINUS
409   STO  0 2
410   JMP  1
411   LOD  0 2
412   RETURN
413   FUNC 10
414   NEW  4
415   LOD  0 1
416   TOINT
417   STO  0 2
418   LOD  0 2
419   TOREAL
420   LOD  0 1
421   RLT
422   JMF  6
423   LOD  0 2
424   LDI  1
425   IPLUS
426   STO  0 2
427   JMP  1
428   LOD  0 2
429   RETURN
430   FUNC 11
431   LOD  0 1
432   LOD  0 1
433   TOREAL
434   LOD  0 2
435   TOREAL
436   RDIV
437   PUSH 1 1 1
438   GOTO 395
439   POP
440   LOD  0 2
441   ITIMES
```

```
442   IMINUS
443   RETURN
444   FUNC 12
445   LOD 0 1
446   LDI 2
447   PUSH 2 0 1
448   GOTO 429
449   POP
450   LDI 0
451   EQU
452   RETURN
453   FUNC 13
454   LOD 0 1
455   JMF 4
456   LOD 0 2
457   NEG
458   JMP 2
459   LDC '0'
460   JMF 3
461   LDC '1'
462   JMP 7
463   LOD 0 1
464   NEG
465   JMF 3
466   LOD 0 2
467   JMP 2
468   LDC '0'
469   RETURN
470   FUNC 14
471   NEWS 19
472   NEW 4
473   NEW 4
474   NEW 4
475   LDA 0 1
476   EIL 15
477   PUSH 1 2 1
478   GOTO 95
479   POP
480   STO 0 3
481   LDA 0 1
482   FDA 15
483   LDA 0 1
484   FDA 15
485   EIL 4
486   LDI 1
487   IPLUS
488   IST
489   LDA 0 1
490   FDA 15
491   EIL 4
492   LOD 0 3
493   PUSH 1 2 1
494   GOTO 371
495   POP
496   EQU
497   JMF 19
498   LDA 0 1
```

```
499   FDA 15
500   LDI 0
501   IST
502   LDA 0 1
503   LDA 0 1
504   EIL 15
505   LDI 0
506   LOD 0 3
507   LDI 1
508   IMINUS
509   PUSH 3 3 0
510   GOTO 612
511   POP
512   IST
513   LOD 0 1
514   RETURN
515   JMP 1
516   LOD 0 3
517   STO 0 5
518   LDI 2
519   STO 0 4
520   LOD 0 4
521   LOD 0 5
522   ILE
523   JMF 45
524   LDA 0 1
525   FDA 15
526   EIL 4
527   LOD 0 4
528   PUSH 1 2 1
529   GOTO 371
530   POP
531   PUSH 2 0 1
532   GOTO 429
533   POP
534   LDI 0
535   NEQ
536   JMF 27
537   LDA 0 1
538   LDA 0 1
539   EIL 15
540   LOD 0 3
541   LOD 0 4
542   IMINUS
543   LDI 1
544   IPLUS
545   LOD 0 3
546   LDI 1
547   IMINUS
548   PUSH 3 3 0
549   GOTO 612
550   POP
551   LOD 0 3
552   LOD 0 4
553   IMINUS
554   LOD 0 3
555   LDI 1
```

```
556   IMINUS
557   PUSH 3 3 0
558   GOTO 569
559   POP
560   IST
561   JMP 7
562   JMP 1
563   LOD 0 4
564   LDI 1
565   IPLUS
566   STO 0 4
567   JMP -47
568   LOD 0 1
569   RETURN
570   FUNC 15
571   NEW 4
572   NEW 1
573   NEW 4
574   LOD 0 3
575   LDI 1
576   IMINUS
577   STO 0 6
578   LOD 0 2
579   STO 0 4
580   LOD 0 4
581   LOD 0 6
582   ILE
583   JMF 28
584   LDA 0 1
585   LOD 0 4
586   IXA 1
587   EIL 1
588   STO 0 5
589   LDA 0 1
590   LOD 0 4
591   IXA 1
592   LDA 0 1
593   LOD 0 4
594   LDI 1
595   IPLUS
596   IXA 1
597   EIL 1
598   IST
599   LDA 0 1
600   LOD 0 4
601   LDI 1
602   IPLUS
603   IXA 1
604   LOD 0 5
605   IST
606   LOD 0 4
607   LDI 1
608   IPLUS
609   STO 0 4
610   JMP -30
611   LOD 0 1
612   RETURN
```

```
613    FUNC 16
614    NEW 4
615    NEW 1
616    NEW 4
617    LOD 0 3
618    LOD 0 2
619    IMINUS
620    TOREAL
621    LDR 2.000000
622    RDIV
623    PUSH 1 1 2
624    GOTO 412
625    POP
626    LDI 1
627    IMINUS
628    STO 0 6
629    LDI 0
630    STO 0 4
631    LOD 0 4
632    LOD 0 6
633    ILE
634    JMF 32
635    LDA 0 1
636    LOD 0 2
637    LOD 0 4
638    IPLUS
639    IXA 1
640    EIL 1
641    STO 0 5
642    LDA 0 1
643    LOD 0 2
644    LOD 0 4
645    IPLUS
646    IXA 1
647    LDA 0 1
648    LOD 0 3
649    LOD 0 4
650    IMINUS
651    IXA 1
652    EIL 1
653    IST
654    LDA 0 1
655    LOD 0 3
656    LOD 0 4
657    IMINUS
658    IXA 1
659    LOD 0 5
660    IST
661    LOD 0 4
662    LDI 1
663    IPLUS
664    STO 0 4
665    JMP -34
666    LOD 0 1
667    RETURN
```

Code A.3: S code of Permutations

A.3   ORZ's Conway's Game of Life

The *.sol* code could be seen in the tutorial section (Code 1.14). Here we report its S code (Code A.4).

```
1   SCODE 270
2   PUSH 0 11 -1
3   GOTO 4
4   POP
5   HALT
6   FUNC 1
7   NEWS 229
8   NEWS 9
9   NEW 4
10  NEW 4
11  NEW 8
12  NEW 8
13  NEW 8
14  NEW 8
15  NEW 8
16  NEW 8
17  LDI 15
18  STO 0 4
19  LDS "Welcome to ORZ's Conway's Game of Life!"
20  STO 0 5
21  LDS "Thanks for playing with ORZ's Conway's Game of Life!\n\n\tBye!"
22  STO 0 6
23  LDS "Your data has been successfully saved in the following file:"
24  STO 0 7
25  LDS "Enter the filename of your world and if you'd like to load from a saved
         state."
26  STO 0 8
27  LDS "Enter for how many generations would you like to watch your world go by
         ."
28  STO 0 9
29  LDS "Your world doesn't exist yet.\nEnter it now."
30  STO 0 10
31  NEW 4
32  LOD 0 5
33  WRITE "s"
34  LOD 0 8
35  WRITE "s"
36  READ 0 2 "(filename:s,load:b)"
37  LDA 0 2
38  FDA 8
39  EIL 1
40  JMF 5
41  LDA 0 2
42  EIL 8
43  FREAD 0 1 "(generation:i,world:[15,[15,b]])"
44  JMP 7
45  LOD 0 10
46  WRITE "s"
47  LDA 0 1
48  FDA 4
```

```
49   RD "[15,[15,b]]"
50   IST
51   LOD 0 9
52   WRITE "s"
53   READ 0 3 "i"
54   LOD 0 3
55   STO 0 11
56   LDI 0
57   STO 0 3
58   LOD 0 3
59   LOD 0 11
60   ILE
61   JMF 23
62   LDA 0 1
63   FDA 4
64   LDA 0 1
65   FDA 4
66   EIL 225
67   PUSH 1 15 0
68   GOTO 96
69   POP
70   IST
71   LDA 0 1
72   LDA 0 1
73   EIL 4
74   LDI 1
75   IPLUS
76   IST
77   LOD 0 1
78   WRITE "(generation:i,world:[15,[15,b]])"
79   LOD 0 3
80   LDI 1
81   IPLUS
82   STO 0 3
83   JMP -25
84   LOD 0 1
85   LDA 0 2
86   EIL 8
87   FWRITE "(generation:i,world:[15,[15,b]])"
88   LOD 0 6
89   LOD 0 7
90   LDA 0 2
91   EIL 8
92   CAT 2 16
93   CAT 2 24
94   WRITE "(:s,:(:s,:s))"
95   LDI 0
96   RETURN
97   FUNC 2
98   NEW 4
99   NEW 4
100  NEW 4
101  NEW 4
102  NEW 4
103  NEWS 225
104  NEWS 12
105  LDI -1
```

```
106   LDI  0
107   LDI  1
108   CAT  3  12
109   STO  0  8
110   NEW  4
111   NEW  4
112   NEWS 12
113   NEW  4
114   NEW  4
115   NEWS 12
116   NEW  4
117   NEW  4
118   LOD  1  4
119   STO  0  9
120   LDI  0
121   STO  0  2
122   LOD  0  2
123   LOD  0  9
124   ILE
125   JMF  145
126   LOD  1  4
127   STO  0  10
128   LDI  0
129   STO  0  3
130   LOD  0  3
131   LOD  0  10
132   ILE
133   JMF  132
134   LDI  0
135   STO  0  6
136   LOD  0  8
137   STO  0  11
138   LDI  2
139   STO  0  13
140   LDI  0
141   STO  0  12
142   LOD  0  12
143   LOD  0  13
144   ILE
145   JMF  89
146   LDA  0  11
147   LOD  0  12
148   IXA  4
149   EIL  4
150   STO  0  4
151   LOD  0  8
152   STO  0  14
153   LDI  2
154   STO  0  16
155   LDI  0
156   STO  0  15
157   LOD  0  15
158   LOD  0  16
159   ILE
160   JMF  69
161   LDA  0  14
162   LOD  0  15
```

```
163    IXA  4
164    EIL  4
165    STO  0  5
166    LOD  0  4
167    LDI  0
168    NEQ
169    JMF  3
170    LDC  '1'
171    JMP  4
172    LOD  0  5
173    LDI  0
174    NEQ
175    JMF  49
176    LOD  0  2
177    LOD  0  4
178    IPLUS
179    LDI  0
180    IGE
181    JMF  7
182    LOD  0  2
183    LOD  0  4
184    IPLUS
185    LOD  1  4
186    ILT
187    JMP  2
188    LDC  '0'
189    JMF  7
190    LOD  0  3
191    LOD  0  5
192    IPLUS
193    LDI  0
194    IGE
195    JMP  2
196    LDC  '0'
197    JMF  7
198    LOD  0  3
199    LOD  0  5
200    IPLUS
201    LOD  1  4
202    ILT
203    JMP  2
204    LDC  '0'
205    JMF  18
206    LDA  0  1
207    LOD  0  2
208    LOD  0  4
209    IPLUS
210    IXA  15
211    LOD  0  3
212    LOD  0  5
213    IPLUS
214    IXA  1
215    EIL  1
216    JMF  6
217    LOD  0  6
218    LDI  1
219    IPLUS
```

```
220   STO  0 6
221   JMP  1
222   JMP  1
223   JMP  1
224   LOD  0 15
225   LDI  1
226   IPLUS
227   STO  0 15
228   JMP  -71
229   LOD  0 12
230   LDI  1
231   IPLUS
232   STO  0 12
233   JMP  -91
234   LDA  0 7
235   LOD  0 2
236   IXA  15
237   LOD  0 3
238   IXA  1
239   LDA  0 1
240   LOD  0 2
241   IXA  15
242   LOD  0 3
243   IXA  1
244   EIL  1
245   JMF  11
246   LOD  0 6
247   LDI  2
248   EQU
249   JMF  3
250   LDC  '1'
251   JMP  4
252   LOD  0 6
253   LDI  3
254   EQU
255   JMP  4
256   LOD  0 6
257   LDI  3
258   EQU
259   IST
260   LOD  0 3
261   LDI  1
262   IPLUS
263   STO  0 3
264   JMP  -134
265   LOD  0 2
266   LDI  1
267   IPLUS
268   STO  0 2
269   JMP  -147
270   LOD  0 7
271   RETURN
```

Code A.4: S code of ORZ's Conway's Game of Life