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

List of Figures

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.

```
func hello_world(): int
begin hello_world
  write "Hello world!";
  return 0;
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 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.

```
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).

```
 type
   from_slides: vector[ 10 ] of
       struct( la: int: lala: vector[ 20 ] of vector[ 5 ] of real );
   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.

```
var
  c: char;
  i: int;
  x, y, z: real;
  s: string;
  b: bool;
  r: struct( a: char; b: string; );
  v: vector [ 5 ] of int;
  w: vector [ 100 ] of struct( a: int; b: char; );
  out_x: real;
  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).

```
const
  MAX: int = 100;
  name: T = "alpha";
  PAIR: struct( a: int; b: char; ) = struct( 25, 'c' );
  VECT: vector [ 5 ] of real = vector( 2.0, 3.12, 4.67, 1.1, 23.0 );
  MAT: vector [ 2 ] of vector [ 5 ] of real =
      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.

```
i, j: int;
l: string;
t: vector [ 5 ] of int;
s: struct( a: int; b: struct( c: string; ); );
v: vector [ 10 ] of vector [ 5 ] of int;
...
-- Reference to the field a of the structure s
i = s.a;
-- Reference to the field c of the field b of the structure s
l = s.b.c;
-- Indexing of the fourth element of v
t = v[3];
-- Indexing of the second element of the first element of v
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.

```
i, j: int;
x, y: real;
...
x = toreal( i + j ) * ( r - toreal( i ) );
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.

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

```
a, b, c: int;
...
if a == b then
  b = c;
elsif a > b then
  c = b + a;
else
  c = b - a;
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).

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

```
...
while a >= b do
  a = a - c;
endwhile;
```

Code 1.10: Example of while loop

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

Code 1.11: Example of for loop

```
i, k: int;
v: vector [ 100 ] of int;
...
foreach i in v do
  k = k + i;
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.

```
a, b, c: int;
filename: string;
...
write [ filename ] a;
b = wr [ filename ] a + c;

read [ filename ] a;
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.

```
func game_of_life() : int

  type
    lines: vector [ 15 ] of bool;
    grid: vector [ 15 ] of lines;
  var
    state: struct( generation: int; world: grid; );
    input: struct( filename: string; load: bool; );
    generations: int;
  const
    world_size: int = 15;
    str_summary: string = "Welcome to ORZ's Conway's Game of Life!";
    str_goodbye: string = "Thanks for playing with ORZ's Conway's Game of
```

```
      Life! \n\n\tBye!";
    str_saved: string = "Your data has been successfully saved in the
      following file:";
    enter_filename: string = "Enter the filename of your world and if you'd
      like to load from a saved state.";
    enter_generations: string = "Enter for how many generations would you
      like to watch your world go by.";
    enter_world: string = "Your world doesn't exist yet.\nEnter it now.";

  -- Rules:
  --  * Any live cell with fewer than two live neighbours dies, as if caused
  --    by under-population.
  --  * Any live cell with two or three live neighbours lives on to the next
  --    generation.
  --  * Any live cell with more than three live neighbours dies, as if by
  --    overcrowding.
  --  * Any dead cell with exactly three live neighbours becomes a live
  --    cell, as if by reproduction.

  func next_state( current_state: grid; ) : grid
    var
      i, j, k, h: int;
      neighbours: int;
      state: grid;
    const
      neighbour_offset: vector[ 3 ] of int = vector( -1, 0, 1 );

  begin next_state
    for i = 0 to world_size-1 do
      for j = 0 to world_size-1 do
        neighbours = 0;
        foreach k in neighbour_offset do
          foreach h in neighbour_offset do
            if k != 0 or h != 0 then
              if i+k >= 0 and i+k < world_size and
                 j+h >= 0 and j+h < world_size then
                if current_state[ i + k ][ j + h ] then
                  neighbours = neighbours + 1;
                endif;
              endif;
            endif;
          endforeach;
        endforeach;

        state[ i ][ j ] = if current_state[ i ][ j ]
                            then neighbours == 2 or neighbours == 3
                            else neighbours == 3 endif;
      endfor;
    endfor;

    return state;
  end next_state

begin game_of_life
  write str_summary;
  write enter_filename;
  read input;
```

```

  if input.load then
    read [ input.filename ] state;
  else
    write enter_world;
    state.world = rd grid;
  endif;

  write enter_generations;
  read generations;

  for generations = 0 to generations-1 do
    state.world = next_state( state.world );
    state.generation = state.generation + 1;
    write state;
  endfor;

  write [ input.filename ] state;
  write struct( str_goodbye, struct( str_saved, input.filename ) );

  return 0;
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.

$program \to func\_decl$
$func\_decl \to \textbf{func id (} decl\_list\_opt \textbf{)} :$
$\quad domain\ type\_sect\_opt\ var\_sect\_opt\ const\_sect\_opt\ func\_list\_opt\ func\_body$
$decl\_list\_opt \to decl\_list \mid \boldsymbol{\epsilon}$
$decl\_list \to decl\,\textbf{;}\,decl\_list \mid decl\,\textbf{;}$
$decl \to id\_list : domain$
$id\_list \to \textbf{id}\,,\,id\_list \mid \textbf{id}$
$domain \to atomic\_domain \mid struct\_domain \mid vector\_domain \mid \textbf{id}$
$atomic\_domain \to \textbf{char} \mid \textbf{int} \mid \textbf{real} \mid \textbf{string} \mid \textbf{bool}$
$struct\_domain \to \textbf{struct (} decl\_list \textbf{)}$
$vector\_domain \to \textbf{vector [intconst] of}\ domain$
$type\_sect\_opt \to \textbf{type}\ decl\_list \mid \boldsymbol{\epsilon}$
$var\_sect\_opt \to \textbf{var}\ decl\_list \mid \boldsymbol{\epsilon}$
$const\_sect\_opt \to \textbf{const}\ const\_list \mid \boldsymbol{\epsilon}$
$const\_list \to const\_decl\ const\_list \mid const\_decl$
$const\_decl \to decl = expr\,\textbf{;}$

$func\_list\_opt \rightarrow func\_list \,|\, \boldsymbol{\epsilon}$
$func\_list \rightarrow func\_decl\, func\_list \,|\, func\_decl$
$func\_body \rightarrow$ **begin id** $stat\_list$ **end id**
$stat\_list \rightarrow stat$ **;** $stat\_list \,|\, stat$ **;**
$stat \rightarrow assign\_stat \,|\, if\_stat \,|\, while\_stat \,|$
$\quad for\_stat \,|\, foreach\_stat \,|\, return\_stat \,|\, read\_stat \,|\, write\_stat$
$assign\_stat \rightarrow left\_hand\_side = expr$
$left\_hand\_side \rightarrow$ **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** $stat\_list$
$elsif\_stat\_list\_opt \rightarrow$
, **else** | $\boldsymbol{\epsilon}$
$else\_stat\_opt \rightarrow else\; stat\_list \,|\, \boldsymbol{\epsilon}$
$while\_stat \rightarrow$ **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**
$specifier\_opt \rightarrow$ **[** $expr$ **]** $|\, \boldsymbol{\epsilon}$
$write\_stat \rightarrow$ **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**
$rel\_term \rightarrow rel\_term\; low\_bin\_op\; low\_term \,|\, low\_term$
$low\_bin\_op \rightarrow + \,|\, \_$
$low\_term \rightarrow low\_term\; high\_bin\_op\; factor \,|\, factor$
$high\_bin\_op \rightarrow * \,|\, /$
$factor \rightarrow unary\_op\; factor \,|$ **(** $expr$ **)** $|\, left\_hand\_side \,|$
$\quad atomic\_const \,|\, instance\_construction \,|\, func\_call \,|\, cond\_expr \,|$
$\quad built\_in\_call \,|\, dynamic\_input$
$unary\_op \rightarrow \_ \,|$ **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** ($expr\_list\_opt$)
$expr\_list\_opt \rightarrow expr\_list \mid \boldsymbol{\epsilon}$
$cond\_expr \rightarrow$ **if** $expr$ **then** $expr$ $elsif\_expr\_list\_opt$ **else** $expr$ **endif**
$elsif\_expr\_list\_opt \rightarrow$ **elsif** $expr$ **then** $expr$ $elsif\_expr\_list\_opt \mid \boldsymbol{\epsilon}$
$built\_in\_call \rightarrow toint\_call \mid toreal\_call$
$toint\_call \rightarrow$ **toint** ($expr$)
$toreal\_call \rightarrow$ **toreal** ($expr$)
$dynamic\_input \rightarrow$ **rd** $specifier\_opt$ $domain$
$dynamic\_output \rightarrow$ **wr** $specifier\_opt$

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.

```
alpha           [a-zA-Z]
digit           [0-9]

id              {alpha}({alpha}|{digit}|_)*

charconst       '([^\']|\\.)'
intconst        {digit}+
realconst       {digit}+\.{digit}+
boolconst       true|false

comment         --.*
spacing         ([ \t])+
sugar           [()\[\]{}.,;]

%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.

```
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.

```
func                    { return( FUNC ); }
char                    { return( CHAR ); }
int                     { return( INT ); }
real                    { return( REAL ); }
string                  { return( STRING ); }
```

Code 3.3: Lex rule for a keyword

```
{intconst}              { lexval.i_val = atoi( yytext );
                          return( INT_CONST ); }
"\""                    { BEGIN strconst;
                          strbuf = malloc( sizeof( char ) ); }
<strconst>([^"\n])*     { concatenate_string( &strbuf, yytext ); }
<strconst>\n[ \t]*      ;
<strconst>\"            { lexval.s_val = new_string( strbuf );
                          BEGIN 0;
                          return( STR_CONST ); }
{charconst}             { yytext[ strlen( yytext ) - 1 ] = '\0';
                          lexval.s_val = new_string( yytext + 1 );
                          return( CHAR_CONST ); }
{realconst}             { lexval.r_val = atof( yytext );
                          return( REAL_CONST ); }
{boolconst}             { lexval.b_val = ( yytext[ 0 ] == 'f'
```

```
                                          ? FALSE
                                          : TRUE );
                          return( BOOL_CONST ); }
{id}                    { lexval.s_val = new_string( yytext );
                          return( ID ); }
{sugar}                 { return( yytext[ 0 ] ); }
.                       { 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.*

```
typedef struct snode
{
  int line;
  Value value;
  TypeNode type;
  struct snode* child;
  struct snode* brother;
} Node;

typedef union
{
  int i_val;
  char* s_val;
  double r_val;
  Boolean b_val;
  Qualifier q_val;
  Nonterminal n_val;
} 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.

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

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.

```
program : func_decl { root = $1; }
        ;
func_decl : FUNC ID { $$ = new_terminal_node( T_ID, lexval ); }
            '(' par_list ')' DEFINE domain type_sect_opt var_sect_opt
            const_sect_opt func_list_opt func_body
            {
```

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

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):

```
// Structure to represent the Schema of a Symbol
typedef struct schema
{
  TypeSchema type;
  char* id;
  int size;
  struct schema* child;
  struct schema* brother;
} Schema;

// Structure to represent a Symbol in the Symbol Table
typedef struct symtab
```

```
{
  // Name of the Symbol
  char* name;
  // Unique identifier in this scope
  int oid;
  // Class of the Symbol
  ClassSymbol clazz;
  // Pointer to the schema of this Symbol
  Schema* schema;
  // Environment/scope in which this Symbol is defined
  map_t locenv;
  // Scope deepness of the Symbol definition
  int nesting;
  // Number of oids' defined in this scope
  int last_oid;
  // Number of formal parameters (used only with functions)
  int formals_size;
  // Pointer to the formal parameters (only with functions)
  struct symtab** formals;
} 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):

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

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

```
switch( node->value.n_val )
{
  case N_FIELDING:
    result = infere_lhs_schema( node->child, is_assigned );
    if( result->type != TS_STRUCT )
      yysemerror( node->child,
            PRINT_ERROR( STR_CONFLICT_TYPE,
                  "not a struct" ) );

    result = result->child;
    while( result != NULL )
    {
      if( result->id == node->child->brother->value.s_val )
```

```
    return result->child;
   result = result->brother;
  }
  yysemerror( node->child->brother,
     PRINT_ERROR( STR_UNDECLARED,
        "not a struct attribute" ) );
  break;

 case N_INDEXING:
  result = infere_lhs_schema( node->child, is_assigned );
  if( result->type != TS_VECTOR )
   yysemerror( node->child,
      PRINT_ERROR( STR_CONFLICT_TYPE,
         "not a vector" ) );

  simplify_expression( node->child->brother );
  if( infere_expression_schema( node->child->brother )->type != TS_INT )
   yysemerror( node->child,
      PRINT_ERROR( STR_CONFLICT_TYPE,
         "expression must be integer" ) );

  result = result->child;
  break;

 default:
  yysemerror( node,
     PRINT_ERROR( STR_BUG,
        "unknown unqualified nonterminal expression" ) );
  break;
}
```

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 6.

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).

```
typedef struct code {
  Stat* head;
  int size;
  Stat* tail;
} Code;
```

```

typedef struct stat {
  int address;
  Operator op;
  Lexval args[ MAX_ARGS ];
  struct stat* next;
} 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.

```
PUSH  <number of formal parameters>
    <number of local parameters>
    <distance between the call environment and the definition one>
GOTO <entry point of function in S-code>
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.

```
typedef struct {
  // Modality of saving of the object, either embedded or in the instance
    stack
  Mode mode;
  // Size of the object in bytes
  int size;
  // Value
  ObjectVal inst;
} Odescr;

typedef struct {
  // Number of objects contained in the activation record
  int obj_number;
  // Pointer to the first object of the activation record in object_stack
  int first_object;
  // Address were to return
  int raddr;
  // Address of the father (definition) in the astack
  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.

```
// Push the chain and element_number on the istack, in preparation of the
    call to GOTO, and instantiate a new activation record
int sol_push( Value* args )
{
  int formals_size = args[ 0 ].i_val;
  int locals_size = args[ 1 ].i_val;
  int chain = args[ 2 ].i_val;

  enlarge_ostack( locals_size );

  push_int( formals_size );
  push_int( locals_size );

#ifdef DEBUG
  fprintf( stderr, "SOL pushed el#: %d, %d\n", formals_size, locals_size );
#endif

  push_int( chain );

#ifdef DEBUG
  fprintf( stderr, "SOL pushed chain: %d\n", chain );
#endif

  return MEM_OK;
}

// GOTO is used ONLY after a push, to perform a function call
int sol_goto( Value* args )
{
  int entry_point = args[ 0 ].i_val;
  int chain = pop_int();
  int locals_size = pop_int();
  int formals_size = pop_int();

  Adescr* function_ar;

#ifdef DEBUG
  fprintf( stderr, "SOL goto chain: %d\n", chain );
  fprintf( stderr, "SOL goto el#: %d, %d\n", formals_size, locals_size );
```

```
#endif

  // The number of elements is given, the start point for its objects is the
  // top of the stack (the objects will be instantiated as part of the
  // function call, not before)
  function_ar = malloc( sizeof( Adescr ) );
  function_ar->obj_number = formals_size + locals_size;
  function_ar->first_object = op - formals_size;
  function_ar->raddr = pc + 1;
  function_ar->alink = ap - 1;

  while( chain-- > 0 )
    function_ar->alink = astack[ function_ar->alink ]->alink;

  push_astack( function_ar );

  // Jump to the entry point (first instruction will be the definition of the formals)
  pc = entry_point - 1;

  return MEM_OK;
}

// Clean the stacks after the last function call
int sol_pop()
{
  int i;
  ByteArray function_result = pop_bytearray();

  for( i = 0; i < top_astack()->obj_number; i++ )
  {
    // All the instances of the current environment are on top of the
    // istack, all I care about is to pop the correct total number of
    // cells, not the exact cells for every object
    if( top_ostack()->mode == STA )
      deallocate_istack( top_ostack()->size );

    pop_ostack();
  }

  pop_astack();

  // Restores the result obtained from the called function
  push_bytearray( function_result.value, function_result.size );

  return MEM_OK;
}
```

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 resources 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.

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

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 through 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.

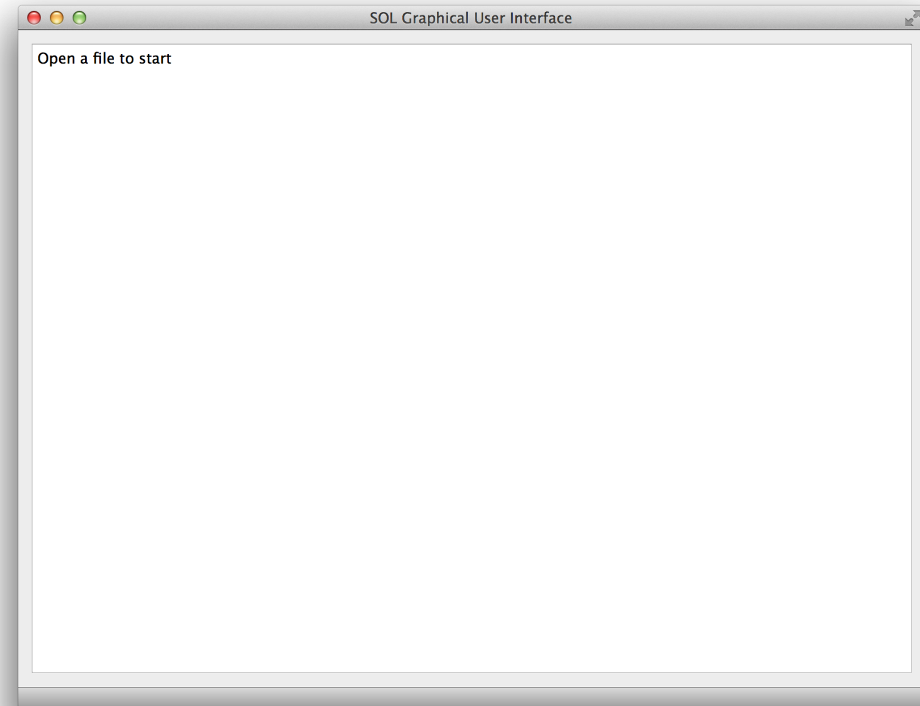

Figure 7.3.1: The main window of the GUI in OS X

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

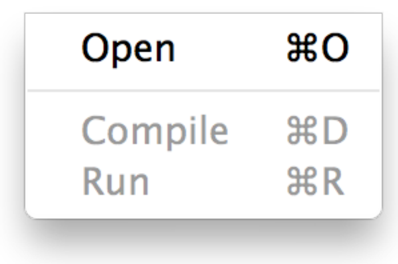

Figure 7.3.3: The File menu of the GUI on OS X

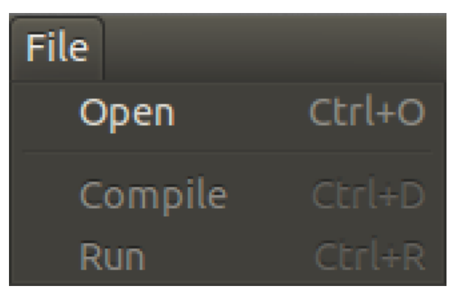

Figure 7.3.4: The File menu of the GUI on Ubuntu 14.04

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.

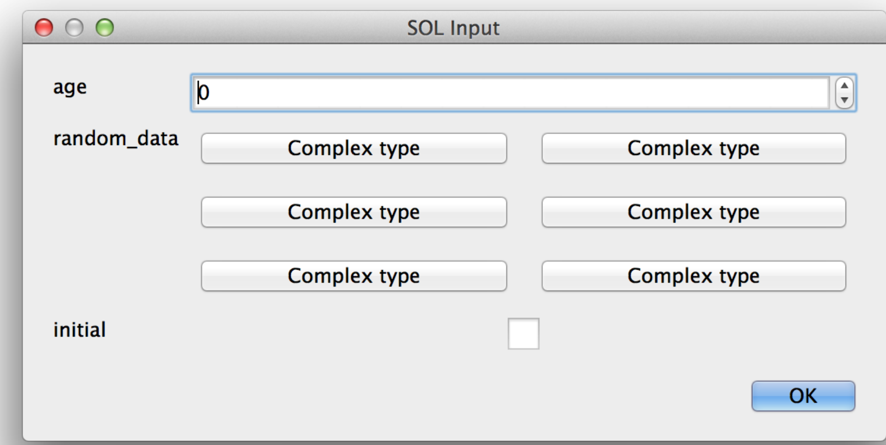

Figure 7.3.5: An example of an input dialog of the GUI on OS X

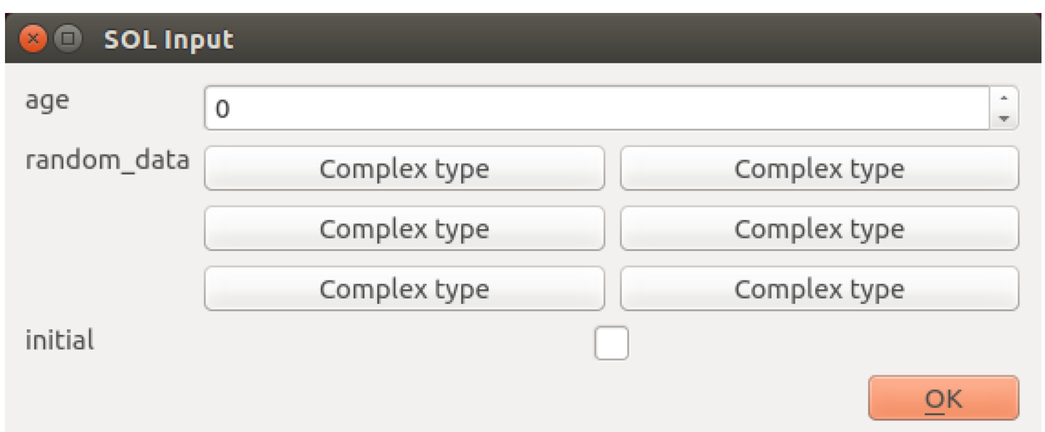

Figure 7.3.6: An example of an input dialog of the GUI on Ubuntu 14.04

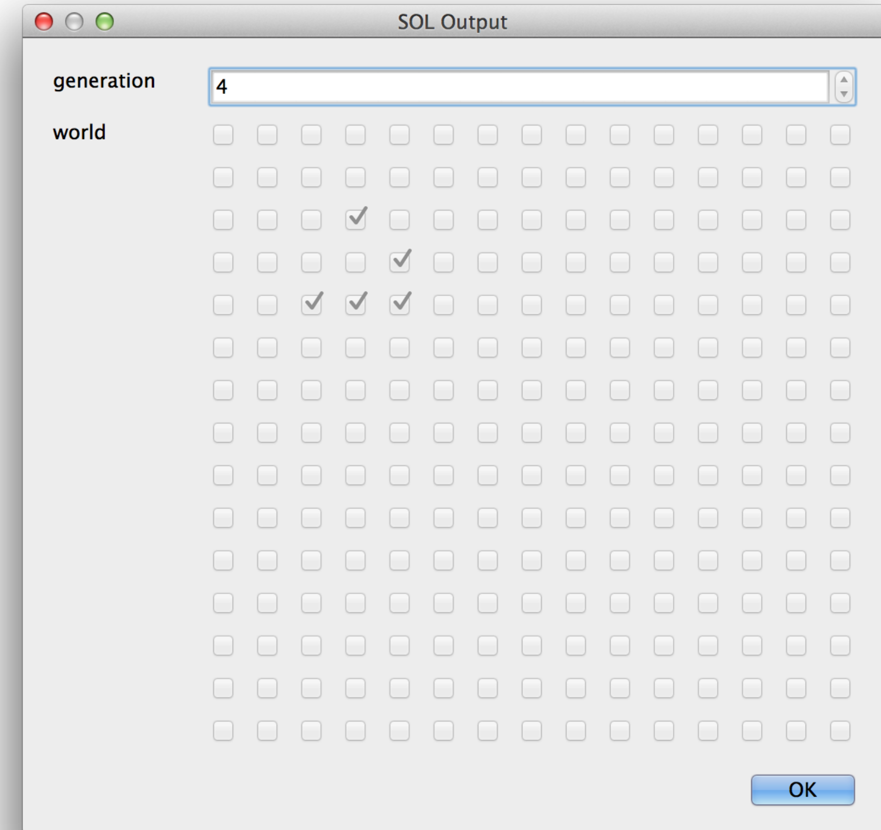

Figure 7.3.7: An example (taken from the execution of Algorithm 1.14) of an output dialog of the GUI on OS X

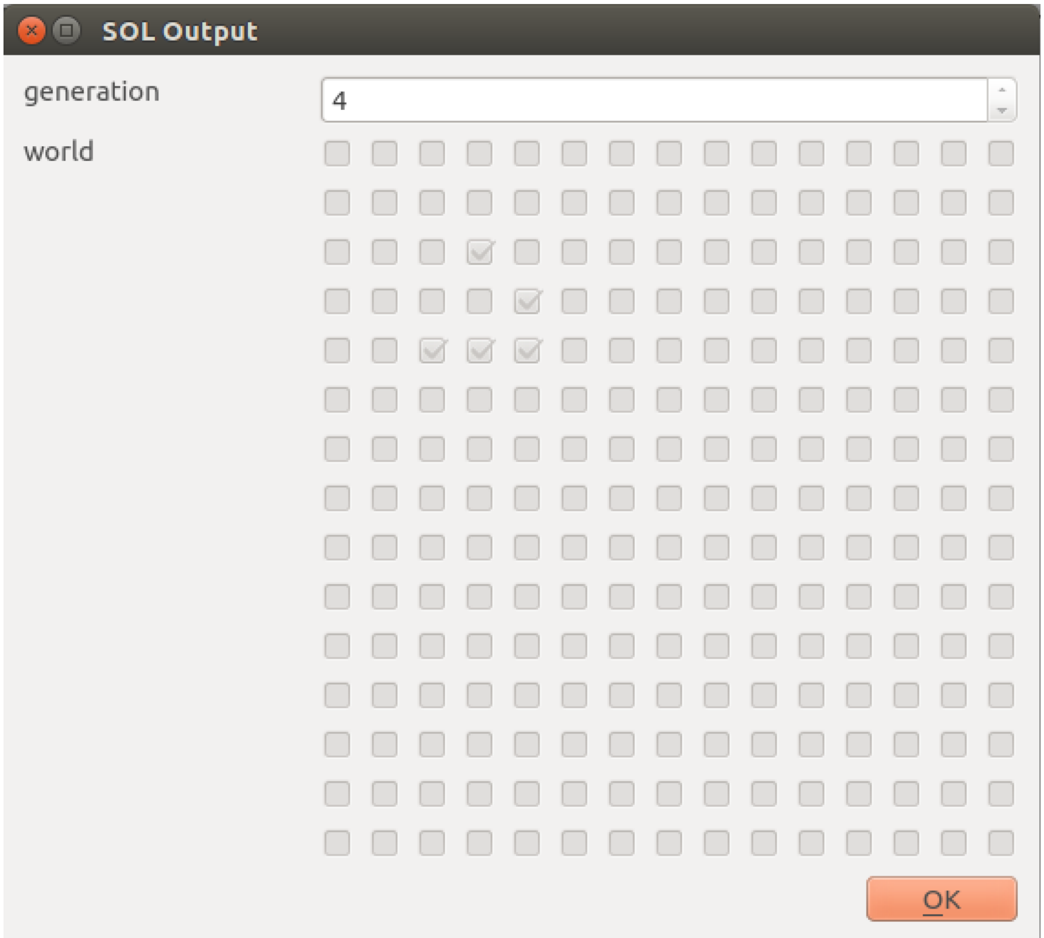

Figure 7.3.8: An example (taken from the execution of Algorithm 1.14) of an output dialog of the GUI on Ubuntu 14.04

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 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 Code 7.1.

```
@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)
```

Code 7.1: The method bla bla

As it can be noticed by the shown dialog examples, the only way to give hints to the user about the structure of the data is through the labels of the used struct. This creates a usability issue and the only solution we have found is, for the *WRITE* operator, to encapsulate the data with a struct and add string constants to illustrate the data and, for *READ*, *RD* and *WR*[1] operators, to precede the command with a *WRITE* of a string presenting the data.

[1]The solution adopted with the *WRITE* operator can't be adopted with *WR* since *WR* also returns the shown data. This would mean that the descriptive string would be included in the result.

Part IV

Conclusions and examples

Chapter 8

Conclusions

It works very well because we have done things like they have to be done. Nespresso, Whatelse?

We would like to thank ?(M|G)Vim, Google+ Hangouts, Blizzard and Valve, our computers (not every computer at all, but..), obviously Tecnomaster for making this project start just in time (with only 30 days of delay).

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).

```
SCODE 9
PUSH 0 0 -1
GOTO 4
POP
HALT
FUNC 1
LDS "Hello world!"
WRITE "s"
LDI 0
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

```
func main() : int

  type
    pseudo_string: vector[ 15 ] of char;
    permutation: struct( str: pseudo_string; perm: int; );

  var
    i: int;
    p: permutation;
    word: pseudo_string;

  const
    MAX_LEN: int = 15;
    INTRO: string = "Insert a word:";

  func new_pseudo_string() : pseudo_string
    var
      result: pseudo_string;
      i: int;
  begin new_pseudo_string
    for i = 0 to MAX_LEN - 1 do
      result[ i ] = '\0';
    endfor;
    return result;
  end new_pseudo_string

  func strlen( s: pseudo_string; ) : int
    var i: int;
  begin strlen
    for i = 0 to MAX_LEN - 1 do
      if s[ i ] == '\0' then
        break;
      endif;
    endfor;
    return i;
  end strlen

  func strcat( s1, s2: pseudo_string; ) : pseudo_string
    var
      result, s: pseudo_string;
      c: char;
  begin strcat
    if strlen( s1 ) + strlen( s2 ) <= MAX_LEN then
      foreach s in vector( s1, s2 ) do
        foreach c in s do
          result[ strlen( result ) ] = c;
        endforeach;
      endforeach;
    endif;

    return result;
  end strcat

  func strcpy( str: pseudo_string; ) : pseudo_string
```

```
  var
    result: pseudo_string;
    i: int;
begin strcpy
  for i = 0 to strlen( str ) - 1 do
    result[ i ] = str[ i ];
  endfor;
  return result;
end strcpy

func recursive_factorial( number: int; ) : int
begin recursive_factorial
  if number <= 2 then
    return number;
  endif;
  return number * recursive_factorial( number - 1 );
end recursive_factorial

func recursive_permutation( to_process: pseudo_string;
                            base: pseudo_string;
                            number: int; ) : pseudo_string
  var
    next_step, current_base, result, temp: pseudo_string;
    i, j: int;

begin recursive_permutation
  -- Returning if i have only one char to process
  if strlen( to_process ) == 1 then
    write vector( struct( "Recursive permutation:",
                          number + 1,
                          strcat( base, to_process ) ) );
    return strcat( base, to_process );
  endif;

  j = 0;
  temp = new_pseudo_string();
  while j < strlen( to_process ) do

    -- Resetting base to the argument value and resetting next_step
    current_base = strcpy( base );
    next_step = new_pseudo_string();
    -- Creating the new string to pass as argument
    for i = 0 to strlen( to_process ) - 1 do
      temp[ 0 ] = to_process[ i ];
      if i != j then
        next_step = strcat( next_step, temp );
      else
        current_base = strcat( current_base, temp );
      endif;
    endfor;

    -- Decrementing the size of the permutations
    result = recursive_permutation( next_step, current_base, number );
    number = number + recursive_factorial( strlen( next_step ) );

    j = j + 1;
  endwhile;
```

```

  -- Completly useless
  return result;
end recursive_permutation

func factorial( i: int; ) : int
  var result: int;
begin factorial
  result = 1;
  for i = 2 to i do
    result = result * i;
  endfor;

  return result;
end factorial

func ceil( r: real; ) : int
  var result: int;
begin ceil
  result = toint( r );
  if toreal( result ) > r then
    result = result - 1;
  endif;

  return result;
end ceil

func floor( r: real; ) : int
  var result: int;
begin floor
  result = toint( r );
  if toreal( result ) < r then
    result = result + 1;
  endif;

  return result;
end floor

func mod( a, b: int; ) : int
begin mod
  return a - ceil( toreal( a ) / toreal( b ) ) * b;
end mod

func is_even( a: int; ) : bool
begin is_even
  return mod( a, 2 ) == 0;
end is_even

func xor( a, b: bool; ) : bool
begin xor
  return ( a and not b ) or ( not a and b );
end xor

func next_permutation( p: permutation; ) : permutation
  var
    result: permutation;
    len, i: int;
```

```

    func circ_shift( s: pseudo_string; a, b: int; ) : pseudo_string
      var
        i: int;
        temp: char;
    begin circ_shift
      for i = a to b-1 do
        temp = s[ i ];
        s[ i ] = s[ i + 1 ];
        s[ i + 1 ] = temp;
      endfor;
      return s;
    end circ_shift

    func flip( s: pseudo_string; a, b: int; ) : pseudo_string
      var
        i: int;
        temp: char;
    begin flip
      for i = 0 to floor( toreal( b - a ) / 2.0 ) - 1 do
        temp = s[ a + i ];
        s[ a + i ] = s[ b - i ];
        s[ b - i ] = temp;
      endfor;
      return s;
    end flip

  begin next_permutation
    len = strlen( p.str );

    p.perm = p.perm + 1;
    if p.perm == factorial( len ) then
      p.perm = 0;
      p.str = flip( p.str, 0, len - 1 );

      return p;
    endif;

    for i = 2 to len do
      if mod( p.perm, factorial( i ) ) != 0 then
        p.str = circ_shift( flip( p.str, len - i + 1, len - 1 ),
                            len - i,
                            len - 1 );
        break;
      endif;
    endfor;

    return p;
  end next_permutation

begin main
  write INTRO;
  p.str = rd pseudo_string;
  word = p.str;

  write vector( struct( "Word:", word ) );

```

```
  -- Iterative permutations
  for i = 0 to factorial( strlen( p.str ) ) - 1 do
    p = wr next_permutation( p );
  endfor;

  -- Recursive permutations
  word = recursive_permutation( word, new_pseudo_string(), 0 );
  return 0;
end main
```

Code A.2: SOL code of Permutations

S-Code A.3

```
SCODE 666
PUSH 0 7 -1
GOTO 4
POP
HALT
FUNC 1
NEW 4
NEWS 19
NEWS 15
NEWS 15
NEW 4
NEW 8
LDI 15
STO 0 5
LDS "Insert a word:"
STO 0 6
NEW 4
LOD 0 6
WRITE "s"
LDA 0 2
RD "[15,c]"
IST
LDA 0 2
EIL 15
STO 0 3
LDS "Word:"
LOD 0 3
CAT 2 23
CAT 1 23
WRITE "[1,(:s,:[15,c])]"
LDA 0 2
EIL 15
PUSH 1 2 0
GOTO 95
POP
PUSH 1 2 0
GOTO 371
POP
LDI 1
IMINUS
STO 0 7
LDI 0
```

```
STO 0 1
LOD 0 1
LOD 0 7
ILE
JMF 12
LOD 0 2
PUSH 1 4 0
GOTO 469
POP
WR "(str:[15,c],perm:i)"
STO 0 2
LOD 0 1
LDI 1
IPLUS
STO 0 1
JMP -14
LOD 0 3
PUSH 0 3 0
GOTO 69
POP
LDI 0
PUSH 3 7 0
GOTO 248
POP
STO 0 3
LDI 0
RETURN
FUNC 2
NEWS 15
NEW 4
NEW 4
LOD 1 5
LDI 1
IMINUS
STO 0 3
LDI 0
STO 0 2
LOD 0 2
LOD 0 3
ILE
JMF 11
LDA 0 1
LOD 0 2
IXA 1
LDC '\0'
IST
LOD 0 2
LDI 1
IPLUS
STO 0 2
JMP -13
LOD 0 1
RETURN
FUNC 3
NEW 4
NEW 4
LOD 1 5
```

```
LDI 1
IMINUS
STO 0 3
LDI 0
STO 0 2
LOD 0 2
LOD 0 3
ILE
JMF 15
LDA 0 1
LOD 0 2
IXA 1
EIL 1
LDC '\0'
EQU
JMF 3
JMP 7
JMP 1
LOD 0 2
LDI 1
IPLUS
STO 0 2
JMP -17
LOD 0 2
RETURN
FUNC 4
NEWS 15
NEWS 15
NEW 1
NEWS 30
NEW 4
NEW 4
NEWS 15
NEW 4
NEW 4
LOD 0 1
PUSH 1 2 1
GOTO 95
POP
LOD 0 2
PUSH 1 2 1
GOTO 95
POP
IPLUS
LOD 1 5
ILE
JMF 52
LOD 0 1
LOD 0 2
CAT 2 30
STO 0 6
LDI 1
STO 0 8
LDI 0
STO 0 7
LOD 0 7
LOD 0 8
```

```
ILE
JMF 39
LDA 0 6
LOD 0 7
IXA 15
EIL 15
STO 0 4
LOD 0 4
STO 0 9
LDI 14
STO 0 11
LDI 0
STO 0 10
LOD 0 10
LOD 0 11
ILE
JMF 19
LDA 0 9
LOD 0 10
IXA 1
EIL 1
STO 0 5
LDA 0 3
LOD 0 3
PUSH 1 2 1
GOTO 95
POP
IXA 1
LOD 0 5
IST
LOD 0 10
LDI 1
IPLUS
STO 0 10
JMP -21
LOD 0 7
LDI 1
IPLUS
STO 0 7
JMP -41
JMP 1
LOD 0 3
RETURN
FUNC 5
NEWS 15
NEW 4
NEW 4
LOD 0 1
PUSH 1 2 1
GOTO 95
POP
LDI 1
IMINUS
STO 0 4
LDI 0
STO 0 3
LOD 0 3
```

```
LOD 0 4
ILE
JMF 14
LDA 0 2
LOD 0 3
IXA 1
LDA 0 1
LOD 0 3
IXA 1
EIL 1
IST
LOD 0 3
LDI 1
IPLUS
STO 0 3
JMP -16
LOD 0 2
RETURN
FUNC 6
LOD 0 1
LDI 2
ILE
JMF 4
LOD 0 1
RETURN
JMP 1
LOD 0 1
LOD 0 1
LDI 1
IMINUS
PUSH 1 0 1
GOTO 231
POP
ITIMES
RETURN
FUNC 7
NEWS 15
NEWS 15
NEWS 15
NEWS 15
NEW 4
NEW 4
NEW 4
LOD 0 1
PUSH 1 2 1
GOTO 95
POP
LDI 1
EQU
JMF 20
LDS "Recursive permutation:"
LOD 0 3
LDI 1
IPLUS
LOD 0 2
LOD 0 1
PUSH 2 9 1
```

```
GOTO 124
POP
CAT 3 27
CAT 1 27
WRITE "[1,(:s,:i,:[15,c])]"
LOD 0 2
LOD 0 1
PUSH 2 9 1
GOTO 124
POP
RETURN
JMP 1
LDI 0
STO 0 9
PUSH 0 3 1
GOTO 69
POP
STO 0 7
LOD 0 9
LOD 0 1
PUSH 1 2 1
GOTO 95
POP
ILT
JMF 75
LOD 0 2
PUSH 1 3 1
GOTO 199
POP
STO 0 5
PUSH 0 3 1
GOTO 69
POP
STO 0 4
LOD 0 1
PUSH 1 2 1
GOTO 95
POP
LDI 1
IMINUS
STO 0 10
LDI 0
STO 0 8
LOD 0 8
LOD 0 10
ILE
JMF 31
LDA 0 7
LDI 0
IXA 1
LDA 0 1
LOD 0 8
IXA 1
EIL 1
IST
LOD 0 8
LOD 0 9
```

```
NEQ
JMF 8
LOD 0 4
LOD 0 7
PUSH 2 9 1
GOTO 124
POP
STO 0 4
JMP 7
LOD 0 5
LOD 0 7
PUSH 2 9 1
GOTO 124
POP
STO 0 5
LOD 0 8
LDI 1
IPLUS
STO 0 8
JMP -33
LOD 0 4
LOD 0 5
LOD 0 3
PUSH 3 7 1
GOTO 248
POP
STO 0 6
LOD 0 3
LOD 0 4
PUSH 1 2 1
GOTO 95
POP
PUSH 1 0 1
GOTO 231
POP
IPLUS
STO 0 3
LOD 0 9
LDI 1
IPLUS
STO 0 9
JMP -80
LOD 0 6
RETURN
FUNC 8
NEW 4
NEW 4
LDI 1
STO 0 2
LOD 0 1
STO 0 3
LDI 2
STO 0 1
LOD 0 1
LOD 0 3
ILE
JMF 10
```

```
LOD 0 2
LOD 0 1
ITIMES
STO 0 2
LOD 0 1
LDI 1
IPLUS
STO 0 1
JMP -12
LOD 0 2
RETURN
FUNC 9
NEW 4
LOD 0 1
TOINT
STO 0 2
LOD 0 2
TOREAL
LOD 0 1
RGT
JMF 6
LOD 0 2
LDI 1
IMINUS
STO 0 2
JMP 1
LOD 0 2
RETURN
FUNC 10
NEW 4
LOD 0 1
TOINT
STO 0 2
LOD 0 2
TOREAL
LOD 0 1
RLT
JMF 6
LOD 0 2
LDI 1
IPLUS
STO 0 2
JMP 1
LOD 0 2
RETURN
FUNC 11
LOD 0 1
LOD 0 1
TOREAL
LOD 0 2
TOREAL
RDIV
PUSH 1 1 1
GOTO 395
POP
LOD 0 2
ITIMES
```

```
IMINUS
RETURN
FUNC 12
LOD 0 1
LDI 2
PUSH 2 0 1
GOTO 429
POP
LDI 0
EQU
RETURN
FUNC 13
LOD 0 1
JMF 4
LOD 0 2
NEG
JMP 2
LDC '0'
JMF 3
LDC '1'
JMP 7
LOD 0 1
NEG
JMF 3
LOD 0 2
JMP 2
LDC '0'
RETURN
FUNC 14
NEWS 19
NEW 4
NEW 4
NEW 4
LDA 0 1
EIL 15
PUSH 1 2 1
GOTO 95
POP
STO 0 3
LDA 0 1
FDA 15
LDA 0 1
FDA 15
EIL 4
LDI 1
IPLUS
IST
LDA 0 1
FDA 15
EIL 4
LOD 0 3
PUSH 1 2 1
GOTO 371
POP
EQU
JMF 19
LDA 0 1
```

```
FDA 15
LDI 0
IST
LDA 0 1
LDA 0 1
EIL 15
LDI 0
LOD 0 3
LDI 1
IMINUS
PUSH 3 3 0
GOTO 612
POP
IST
LOD 0 1
RETURN
JMP 1
LOD 0 3
STO 0 5
LDI 2
STO 0 4
LOD 0 4
LOD 0 5
ILE
JMF 45
LDA 0 1
FDA 15
EIL 4
LOD 0 4
PUSH 1 2 1
GOTO 371
POP
PUSH 2 0 1
GOTO 429
POP
LDI 0
NEQ
JMF 27
LDA 0 1
LDA 0 1
EIL 15
LOD 0 3
LOD 0 4
IMINUS
LDI 1
IPLUS
LOD 0 3
LDI 1
IMINUS
PUSH 3 3 0
GOTO 612
POP
LOD 0 3
LOD 0 4
IMINUS
LOD 0 3
LDI 1
```

```
IMINUS
PUSH 3 3 0
GOTO 569
POP
IST
JMP 7
JMP 1
LOD 0 4
LDI 1
IPLUS
STO 0 4
JMP -47
LOD 0 1
RETURN
FUNC 15
NEW 4
NEW 1
NEW 4
LOD 0 3
LDI 1
IMINUS
STO 0 6
LOD 0 2
STO 0 4
LOD 0 4
LOD 0 6
ILE
JMF 28
LDA 0 1
LOD 0 4
IXA 1
EIL 1
STO 0 5
LDA 0 1
LOD 0 4
IXA 1
LDA 0 1
LOD 0 4
LDI 1
IPLUS
IXA 1
EIL 1
IST
LDA 0 1
LOD 0 4
LDI 1
IPLUS
IXA 1
LOD 0 5
IST
LOD 0 4
LDI 1
IPLUS
STO 0 4
JMP -30
LOD 0 1
RETURN
```

```
FUNC 16
NEW 4
NEW 1
NEW 4
LOD 0 3
LOD 0 2
IMINUS
TOREAL
LDR 2.000000
RDIV
PUSH 1 1 2
GOTO 412
POP
LDI 1
IMINUS
STO 0 6
LDI 0
STO 0 4
LOD 0 4
LOD 0 6
ILE
JMF 32
LDA 0 1
LOD 0 2
LOD 0 4
IPLUS
IXA 1
EIL 1
STO 0 5
LDA 0 1
LOD 0 2
LOD 0 4
IPLUS
IXA 1
LDA 0 1
LOD 0 3
LOD 0 4
IMINUS
IXA 1
EIL 1
IST
LDA 0 1
LOD 0 3
LOD 0 4
IMINUS
IXA 1
LOD 0 5
IST
LOD 0 4
LDI 1
IPLUS
STO 0 4
JMP -34
LOD 0 1
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).

```
SCODE 270
PUSH 0 11 -1
GOTO 4
POP
HALT
FUNC 1
NEWS 229
NEWS 9
NEW 4
NEW 4
NEW 8
NEW 8
NEW 8
NEW 8
NEW 8
NEW 8
LDI 15
STO 0 4
LDS "Welcome to ORZ's Conway's Game of Life!"
STO 0 5
LDS "Thanks for playing with ORZ's Conway's Game of Life!\n\n\tBye!"
STO 0 6
LDS "Your data has been successfully saved in the following file:"
STO 0 7
LDS "Enter the filename of your world and if you'd like to load from a saved
  state."
STO 0 8
LDS "Enter for how many generations would you like to watch your world go by
  ."
STO 0 9
LDS "Your world doesn't exist yet.\nEnter it now."
STO 0 10
NEW 4
LOD 0 5
WRITE "s"
LOD 0 8
WRITE "s"
READ 0 2 "(filename:s,load:b)"
LDA 0 2
FDA 8
EIL 1
JMF 5
LDA 0 2
EIL 8
FREAD 0 1 "(generation:i,world:[15,[15,b]])"
JMP 7
LOD 0 10
WRITE "s"
LDA 0 1
FDA 4
```

```
RD "[15,[15,b]]"
IST
LOD 0 9
WRITE "s"
READ 0 3 "i"
LOD 0 3
STO 0 11
LDI 0
STO 0 3
LOD 0 3
LOD 0 11
ILE
JMF 23
LDA 0 1
FDA 4
LDA 0 1
FDA 4
EIL 225
PUSH 1 15 0
GOTO 96
POP
IST
LDA 0 1
LDA 0 1
EIL 4
LDI 1
IPLUS
IST
LOD 0 1
WRITE "(generation:i,world:[15,[15,b]])"
LOD 0 3
LDI 1
IPLUS
STO 0 3
JMP -25
LOD 0 1
LDA 0 2
EIL 8
FWRITE "(generation:i,world:[15,[15,b]])"
LOD 0 6
LOD 0 7
LDA 0 2
EIL 8
CAT 2 16
CAT 2 24
WRITE "(:s,:(:s,:s))"
LDI 0
RETURN
FUNC 2
NEW 4
NEW 4
NEW 4
NEW 4
NEW 4
NEWS 225
NEWS 12
LDI -1
```

```
LDI 0
LDI 1
CAT 3 12
STO 0 8
NEW 4
NEW 4
NEWS 12
NEW 4
NEW 4
NEWS 12
NEW 4
NEW 4
LOD 1 4
STO 0 9
LDI 0
STO 0 2
LOD 0 2
LOD 0 9
ILE
JMF 145
LOD 1 4
STO 0 10
LDI 0
STO 0 3
LOD 0 3
LOD 0 10
ILE
JMF 132
LDI 0
STO 0 6
LOD 0 8
STO 0 11
LDI 2
STO 0 13
LDI 0
STO 0 12
LOD 0 12
LOD 0 13
ILE
JMF 89
LDA 0 11
LOD 0 12
IXA 4
EIL 4
STO 0 4
LOD 0 8
STO 0 14
LDI 2
STO 0 16
LDI 0
STO 0 15
LOD 0 15
LOD 0 16
ILE
JMF 69
LDA 0 14
LOD 0 15
```

```
IXA 4
EIL 4
STO 0 5
LOD 0 4
LDI 0
NEQ
JMF 3
LDC '1'
JMP 4
LOD 0 5
LDI 0
NEQ
JMF 49
LOD 0 2
LOD 0 4
IPLUS
LDI 0
IGE
JMF 7
LOD 0 2
LOD 0 4
IPLUS
LOD 1 4
ILT
JMP 2
LDC '0'
JMF 7
LOD 0 3
LOD 0 5
IPLUS
LDI 0
IGE
JMP 2
LDC '0'
JMF 7
LOD 0 3
LOD 0 5
IPLUS
LOD 1 4
ILT
JMP 2
LDC '0'
JMF 18
LDA 0 1
LOD 0 2
LOD 0 4
IPLUS
IXA 15
LOD 0 3
LOD 0 5
IPLUS
IXA 1
EIL 1
JMF 6
LOD 0 6
LDI 1
IPLUS
```

```
STO 0 6
JMP 1
JMP 1
JMP 1
LOD 0 15
LDI 1
IPLUS
STO 0 15
JMP -71
LOD 0 12
LDI 1
IPLUS
STO 0 12
JMP -91
LDA 0 7
LOD 0 2
IXA 15
LOD 0 3
IXA 1
LDA 0 1
LOD 0 2
IXA 15
LOD 0 3
IXA 1
EIL 1
JMF 11
LOD 0 6
LDI 2
EQU
JMF 3
LDC '1'
JMP 4
LOD 0 6
LDI 3
EQU
JMP 4
LOD 0 6
LDI 3
EQU
IST
LOD 0 3
LDI 1
IPLUS
STO 0 3
JMP -134
LOD 0 2
LDI 1
IPLUS
STO 0 2
JMP -147
LOD 0 7
RETURN
```

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