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

Listings

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.

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 Listing 1.1.

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

Listing 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 Listing 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
```

Listing 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 Listing 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;
```

Listing 1.3: Types

A list of variables This is similar to that in the definition of parameters, as in Listing 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;
```

Listing 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 Listing 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 ) );
```

Listing 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 Listing 1.6. Note that a double dash starts a comment.

```
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
s.a;
-- Reference to the field c of the field b of the structure s
s.b.c;
-- Indexing of the fourth element of v
v[3];
-- Indexing of the second element of the first element of v
v[0][1];
```

Listing 1.6: Examples of indexing and fielding

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 Listing 1.7.

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

Listing 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 than allows different lists of statements to be executed when different conditions apply.

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

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

Listing 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;
```

Listing 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* (Listing 1.10), the *for* (Listing 1.11) and the *foreach* (Listing 1.12).

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

Listing 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;
```

Listing 1.11: Example of for loop

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

Listing 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 Listing 1.13.

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

read [ filename ] a;
b = rd [ filename ] int;
```

Listing 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 assumes 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.

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
```

Listing 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.

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

func_list_opt → *func_list* | ϵ

func_list → *func_decl func_list* | *func_decl*

func_body → **begin id** *stat_list* **end id**

stat_list → *stat* **;** *stat_list* | *stat* **;**

stat → *assign_stat* | *if_stat* | *while_stat* |
  for_stat | *foreach_stat* | *return_stat* | *read_stat* | *write_stat*

assign_stat → *left_hand_side* = *expr*

left_hand_side → **id** | *fielding* | *indexing*

fielding → *left_hand_side* **. id**

indexing → *left_hand_side* **[** *expr* **]**

if_stat → **if** *expr* **then** *stat_list elsif_stat_list_opt else_stat_opt* **endif**

elsif_stat_list_opt → **elsif** *expr* **then** *stat_list*

elsif_stat_list_opt →

, **else** | ϵ

else_stat_opt → *else stat_list* | ϵ

while_stat → **while** *expr* **do** *stat_list* **endwhile**

for_stat → **for id** = *expr* **to** *expr* **do** *stat_list* **endfor**

foreach_stat → **foreach id in** *expr* **do** *stat_list* **endforeach**

return_stat → **return** *expr*

read_stat → **read** *specifier_opt* **id**

specifier_opt → **[** *expr* **]** | ϵ

write_stat → **write** *specifier_opt expr*

expr → *expr bool_op bool_term* | *bool_term*

bool_op → **and** | **or**

bool_term → *rel_term rel_op rel_term* | *rel_term*

rel_op → == | ! = | > | >= | < | <= | **in**

rel_term → *rel_term low_bin_op low_term* | *low_term*

low_bin_op → + | _

low_term → *low_term high_bin_op factor* | *factor*

high_bin_op → ∗ | /

factor → *unary_op factor* | **(** *expr* **)** | *left_hand_side* |
  atomic_const | *instance_construction* | *func_call* | *cond_expr* |
  built_in_call | *dynamic_input*

unary_op → _ | **not** | *dynamic_output*

atomic_const → **charconst** | **intconst** | **realconst** | **strconst** | **boolconst**

instance_construction → *struct_construction* | *vector_construction*

struct_construction → **struct (** *expr_list* **)**

expr_list → *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
```

Listing 3.1: Lex definition of lexical elements

The rules associated to each lexeme must be in the form presented in Listing 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; }
```

Listing 3.2: Lex rule

For the fixed lexemes (keywords and other simple stuff), the rules are usually as simple the ones in in Listing 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 Listing 3.4.

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

Listing 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 ); }
```

Listing 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 Listing 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;
```

Listing 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 Listing 3.6.

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

Listing 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 Listing 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 );
            }
        ;
```

Listing 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

Starting from the tree, the *yysem* function (this time written entirely by us, as there's no language for generating a semantical analyzer automatically) analyzes the whole code in search for semantical errors. To support itself in this operation, 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 (Listing 4.1):

```c
// 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
{
  char* name;
  int oid;
  ClassSymbol clazz;
  Schema* schema;
  map_t locenv;
  int nesting;
  int last_oid;
  int formals_size;
  struct symtab** formals;
} Symbol;
```

Listing 4.1: Symbol Table structure

The main function of this part is *yysem* in which we look through the Abstract Syntax Tree, generated before, and check if there are any types error between nodes.

Every symbol, depending on its type, has to be enumerated so it could be uniquely found when executing the code. We will keep track of 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 after 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 (Listing 4.3):

```
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
```

Listing 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 schema are compatible each other.

To create a schema we used a recursive approach, descending through the node until its leaves, which have to be *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 **??**

```c
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;
}
```

Listing 4.3: Symbol Table example

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. S-code is a very low level language not dissimilar from Assembly.

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

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

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

5.2 The yygen function

When the *yygen* function is called, it automatically retrieves the root of the Syntax Tree and passes it to *generate_code*. This function consists of a big switch of the node's type and, for every type, it generates an instance of *Code* (a structure pointing to a list of pointers to another structure *Stat*, which in turn contains the actual instructions, see Listing 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;
```

Listing 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 Listing 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 arguments 3) and a pointer to the next instruction.

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

// TODO add other difficult problems w/ solution

5.2.1 Function problem

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

```
PUSH <number of objects in the function's environment> <distance between the
     call environment and the definition one>
GOTO <entry point of function in S-code>
POP
```

Listing 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). 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.

Part III

The Virtual Machine

Chapter 6

Introduction and S-code execution

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. Then, the function *yyvm* is called. This function takes the Code, saved in the global variable *program*, and executes it statement by statement, passing thought a big switch.

6.1 Stacks. Stacks everywhere

YO BASTARD ALMOND

STACKS
STACKS EVERYWHERE
memegenerator.net

Chapter 7

Graphical interface

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

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

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

// TODO expand

Conclusions

WYNOUNICODEBRO:(