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* and the two complex ones *vector* and *struct*, whose syntax is defined later.

```
func program( par1, par2: int; par3: 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, even if 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 flieds 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, 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, whose definition 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. Note that an assignment separated by the variable's definition follows the same syntax but without the indication of the variable's type (that is, something like *variable_ name = value;*).

```
const
  MAX: int = 100;
  name: T2 = "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, 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).

// TODO add tests

1.1 A full sol program

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

// TODO GoL

Chapter 2

SOL language syntax specification

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

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

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

// TODO extend?

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

$const\_decl \rightarrow decl = expr$ **;**
$func\_list\_opt \rightarrow func\_list \,|\, \boldsymbol{\epsilon}$
$func\_list \rightarrow func\_decl\, func\_list \,|\, func\_decl$
$func\_body \rightarrow$ **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** **(** $expr\_list$ **)**

$expr\_list \rightarrow expr\,,\, expr\_list \,|\, expr$
$vector\_construction \rightarrow \textbf{vector}\ (\, expr\_list\,)$
$func\_call \rightarrow \textbf{id}\ (\, expr\_list\_opt\,)$
$expr\_list\_opt \rightarrow expr\_list \,|\, \boldsymbol{\epsilon}$
$cond\_expr \rightarrow \textbf{if}\ expr\ \textbf{then}\ expr\ elsif\_expr\_list\_opt\ \textbf{else}\ expr\ \textbf{endif}$
$elsif\_expr\_list\_opt \rightarrow \textbf{elsif}\ expr\ \textbf{then}\ expr\ elsif\_expr\_list\_opt \,|\, \boldsymbol{\epsilon}$
$built\_in\_call \rightarrow toint\_call \,|\, toreal\_call$
$toint\_call \rightarrow \textbf{toint}\ (\, expr\,)$
$toreal\_call \rightarrow \textbf{toreal}\ (\, expr\,)$
$dynamic\_input \rightarrow \textbf{rd}\ specifier\_opt\ domain$
$dynamic\_output \rightarrow \textbf{wr}\ specifier\_opt$

Chapter 3

SOL language semantics specification

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

// TODO explain everything in spec.pdf

Listing 3.1: Definitions and assignments

```
```

Part II

The Compiler

Chapter 4

Lexical and Syntactix 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 wether 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 5, which aims at determinining if the written statements (which are correct thanks to the previous analyses) make sense (eg, performing the sum of an integer and a string makes no sense, therefore it is not semantically correct), relying on the data structure produced by the previous analysis;

- The code generation, presented in Chapter 6, which, given that the code is both well-written and semantically correct, translates it in a lower-level and standard code, easier to execute directly (and executed by the virtual machine, of which we talk in Part III). The code is, again, generated starting from the data structure produced by the syntactical analysis, not from the "raw" code.

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

4.1 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, given the input file, produces a data structure (called *symbol table*) containing all the symbols found in the code (constants, ids etc). If, during, the file analysis, an error is encountered (in the file is present something that shouldn't be), the *yylex* function stops and produces an error (calling the *yyerror* function).

The symbol table will be used, along with the source SOL file, 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, lexical elements (a *lexeme*) that need to be defined with a regular expression (such as the id) are defined, in the second part these lexeme are associated to a rule, and the last part contains mostly C functions used in the lex rules of the second part. The lexeme that don't need to be defined are those whose definition is fixed, such as the keywords and the operators.

The definition of lexeme for the SOL language is, for our compiler, the one presented in 4.1.

```
alpha [a-zA-Z]
digit [0-9]
id  {alpha}({alpha}|{digit}|_)*

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

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

%x charconst
```

Listing 4.1: Lex definition of lexical elements

The rules associated to each lexeme must be in the form presented in

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

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

Listing 4.2: Lex rule

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

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

Listing 4.3: Lex rule for a keyword

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

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

Listing 4.4: Lex rules for constants and ids

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

4.2 Syntactical analyzer

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

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

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

identifier determining the nonterminal type if the node is an unqualified nonterminal (the identifiers are represented as possible values of the enumerator *NonTerminal*) or a unique identifier determining the qualifier to be used if the node is a qualified nonterminal (these are represented as possibile 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 4.5: The Node structure

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

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

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

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

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

Listing 4.6: Structure of a translation rule

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

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

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

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

```
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 4.7: Extract of the translation rules for SOL

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

Chapter 5

Semantical analysis

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

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

// TODO IDK how this works

Chapter 6

Code generation

Starting from the tree generated by the syntactic analysis and the table produced by the semantical one, the *yygen* (again, written by us) function proceeds 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).

6.1 S-code specification

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

Everything is done on a global stack. Every instruction has zero to three operands and operates implicitly on the last values present on the stack (generally the last one or two). For example, the instruction to perform a sum of 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”?

6.2 The yygen function

When the *yygen* function is called, it automatically retrieves the root of the Syntax Tree and passes it to *generate_code*. This function consists of a big switch of the node's type and, for every type, it generates an instance of *Code* (a structure pointing to a list of pointers to another structure *Stat*, which in turn contains the actual instructions, see Listing 6.1 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 6.1: 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 6.1, 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

6.2.1 Function problem

The generation of the code for function declarations and calls caused some problem because the call (which is translated as in Listing 6.2) 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 6.2: 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 7

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.

7.1 Stacks. Stacks everywhere

YO BASTARD ALMOND


STACKS
STACKS EVERYWHERE
memegenerator.net

Chapter 8

Graphical interface

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

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

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

// TODO expand

Conclusions

WYNOUNICODEBRO:(