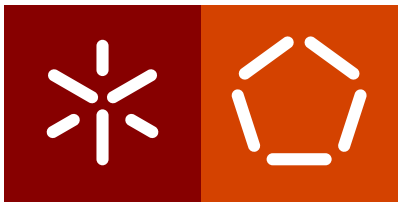
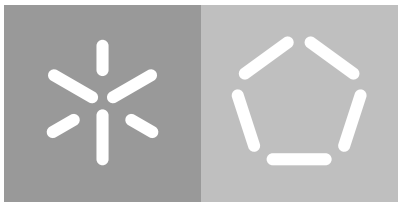


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Damien da Silva Vaz

Implementing an Integrated Syntax Directed Editor for LISS.

September 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Damien da Silva Vaz

Implementing an Integrated Syntax Directed Editor for LISS.

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor Pedro Rangel Henriques
Professor Daniela da Cruz

September 2016

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor Pedro Rangel Henriques and co-supervisor Daniela da Cruz. They are the most who supported me throw this ambitious project and took me to the final stage of my university career.
Thank you also to my family and friends (Ranim, Bruno, Chloé, Tiago, Tamara, Saozita, Nuno, David) for supporting me.
And last but not least, I would like to dedicate this thesis to my, particularly, most beautiful mother. Despite you couldn't be here to watch me conclude my studies. Wherever you are, I hope that you are proud of me. None of this could have been made without their unconditional help.

ABSTRACT

The aim of this master work is to implement LISS language in ANTLR compiler generator system using an attribute grammar which create an abstract syntax tree (AST) and generate MIPS assembly code for MARS (MIPS Assembler and Runtime Simulator) . Using that AST, it is possible to create a Syntax Directed Editor (SDE) in order to provide the typical help of a structured editor which controls the writing according to language syntax as defined by the underlying context free grammar.

RESUMO

O tema desta dissertação é implementar a linguagem LISS em ANTLR com um gramática de atributos e no qual, irá criar uma árvore sintática abstrata e gerar MIPS assembly código para MARS (MIPS Assembler and Runtime Simulator). Usando esta árvore sintática abstrata, criaremos uma SDE (Editor Dirigido a Sintaxe) no qual fornecerá toda a ajuda típica de um editor estruturado que controlará a escrita de acordo com a gramática.

CONTENTS

LIST OF FIGURES

1 http://www.biiet.org/blog/wp-content/uploads/2013/07/img028.jpg
2 http://www.alauda.ro/wp-content/uploads/2011/04/XCode-interface-e1302035068112.png

LIST OF TABLES

List of Tables

1

INTRODUCTION

In informatics, solving problems with computers is related to the necessity of helping the end-users, facilitating their life. And all these necessities pass through developers who creates programs for this purpose.

However, developing programs is a difficult task; analyzing problems, and debugging software takes effort and time.

And this is why we must find a solution for these problems.

Developing a software package requires tools to help the developers to maximize their programming productivity. These tools are: on one hand, compilers to generate lower-level code (machine code) from the high-level source code (the input program written in an high-level programming language); on the other hand, editors to create that source code. And to make easier and safer the programmers work, high-level programming languages were created for facilitating their work.

This is not enough to overcome all the difficulties for creating a program in a safety way and having a high level productivity!

This is why we need to have fresh ideas and to implement more features to help on solving these problems.

1.1 OBJECTIVES

In this work, this project aims to develop an editor with the concept of a SDE (Syntax Directed Editor).

It is intended that the editor works with language designed by the members of the Language Processing group at UM which is called LISS.

LISS language will be specified by an attribute grammar that will be passed, as input, to ANTLR. The compiler generated by ANTLR will generate MIPS assembly code (lower-level source code).

The front-end and the back-end of that compiler will be explained and detailed along the next pages.

1.2 RESEARCH HYPOTHESIS

1.3 THESIS OUTCOMES

1.4 DOCUMENT STRUCTURE

In this section, the project planned for this master thesis will be explained.

First, create an ANTLR version of the CFG grammar for LISS language.

Second, extend the LISS CFG to an AG in order to specify throw it the generation of MIPS assembly code. To verify the correctness of the assembly code generated, a simple MIPS simulator, named MARS, will be selected to provide all the tools for checking it.

Third, the desired Structure-Editor, SDE, will be developed based on ANTLR. It will be implemented in with Java SWING because ANTLR has always been implemented via Java and it is said, also, to use Java target as a reference implementation mirrored by other targets. SWING is a GUI widget toolkit for Java which provides all the API for creating an interface with Java. At this phase, we will create an IDE similar to other platforms but with the capacity of being a syntax-directed editor.

Fourthly, to complete the SDE functionality, an incremental compiler shall be included. Incremental compilation (Reps et al., 1983; Holsti, 1986; Vogt et al., 1990) means that only the part that was changed must be processed again. And like that, both tasks (edition and compilation) are done synchronously at the same time and having an editor which compiles cleverly.

Finally, exhaustive and relevant tests will be made with the tool created and, the outcomes will be analyzed and discussed.

2

LANGUAGES AND GRAMMAR: CONCEPT & TOOLS

A grammar (Chomsky, 1962; Gaudel, 1983; Waite and Goos, 1984; Aho et al., 1986; Kastens, 1991b; Muchnick, 1997; Hopcroft et al., 2006; Grune et al., 2012) is a set of derivation rules (or production) that explains how words are used to build the sentences of a language.

A grammar (Deransart et al., 1988; Alblas, 1991; Kastens, 1991a; Swierstra and Vogt, 1991; Deransart and Jourdan, 1990; Räihä, 1980; Filè, 1983; Oliveira et al., 2010) is considered to be a language generator and also a language recognizer (checking if a sentence is correctly derived from the grammar).

The rules describe how a string is formed using the language alphabet, defining the sentences that are valid according to the language syntax.

One of the most important researchers in this area was Noam Chomsky. He defined the notion of grammar in computer science's field.

He described that a formal grammar is composed by a finite set of production rules

$$(\text{left hand side} \mapsto \text{right hand side})$$

where each side is composed by a sequence of symbols.

These symbols are split into two sets : non terminals, terminals; the start symbol is a special non-terminal.

There is, always, at least one rule for the start symbol (see Figure 1) followed by other rules to derive each non-terminal. The non terminals are symbols which can be replaced and terminals are symbols which cannot be.

One valid sentences (Example in Figure 1), could be : bbebee .

In the compilers area two major classes of grammars are used : CFG (Context-free grammar) and AG (Attribute Grammar).

The difference between these two grammars are that a CFG is directed to define the syntax (only) and, AG contains semantic and syntax rules.

An AG is , basically, a GFC grammar extended with semantic definitions. It is a formal way to define attributes for the symbols that occur in each production of the underlying grammar. We can associate values to these attributes later, after processed with a parser; the evaluation will occur applying those semantic definition to any node of the abstract syntax tree. These attributes are divided into two groups: synthesized attributes and inherited attributes.

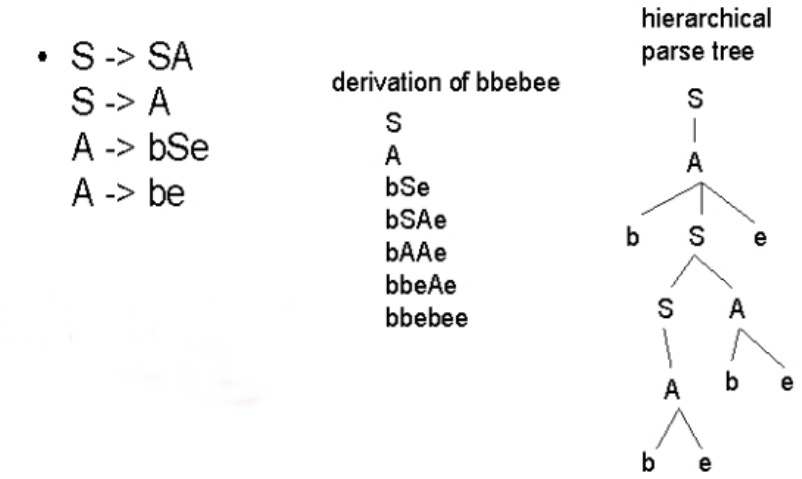


Figure 1.: CFG example [1]

The synthesized attributes are the result of the attribute evaluation rules for the root symbol of each subtree, and may also use the values of the inherited attributes. The inherited attributes are passed down from parent nodes to children or between siblings.

Like that it is possible to transport information anywhere in the abstract syntax tree which is one of the strength for using an AG (as seen on Listing 2.1).

```
facturas : fatura (facturas)*
         ;

fatura : 'FATURA' cabec 'VENDAS' corpo {System.out.println("Total Factura
    : "+$corpo.totOut);}
       ;

cabec : idFat idForn 'CLIENTE' idClie {System.out.println("Factura n: "+
    $idFat.text);}
      ;

idFat : numFat ;

numFat : ID ;

idForn : nome morada 'NIF:' nif 'NIB:' nib
       ;

idClie : nome morada 'NIF:' nif
       ;

nome :  STR ;
```

```
21
22 morada : STR;
23
24 nif : STR;
25
26 nib : STR;
27
28 corpo returns [int totOut]
29     : linha '.' {$totOut += $linha.linhatot;}
30       (linha '.' {$totOut += $linha.linhatot;})*
31     ;
32
33 linha returns [int linhatot]
34       : refProd '|' valUnit '|' quant {$linhatot = $valUnit.val * $quant.
             quan;System.out.println("Ref: "+$refProd.text+" Total linha: "+(
             $linhatot)+" Euros");}
35       ;
36
37   refProd : ID;
38 valUnit returns [int val]
39         : NUM {$val = $NUM.int;}
40         ;
41 quant returns [int quan]
42     : NUM {$quan = $NUM.int;}
43     ;
```

Listing 2.1: Example of an AG

In this way, an AG will be used to specify the translation from syntax tree directly into code for some specific machine or into another intermediate language. For our thesis, the AG will be processed by ANTLR tool in order to build automatically the parser, the attribute evaluator, and the code generation.

2.1 FORMAL GRAMMAR

According to Noam Chomsky, a classic formalization of generative grammars is composed by:

- A finite set N of nonterminals symbols.
- A finite set $\sum$ of terminals symbols.
- A finite set P of production rules.

- A start symbol S $\in$ P

A grammar is formally constructed by that tuple (N,$\sum$,P,S).

Grammar is a set of productions rules which describes the syntax of the language (not semantic). Each grammar has only one start symbol production that defines where the grammar begins. And each production is composed by two things : LHS (Left Hand Side) and RHS (Right and Side). Left Hand Side represents the non terminal and the right hand side represents the behaviour of the rule (composed by non terminal and terminal).

```
1     liss : 'program' identifier body
2          ;
```

Listing 2.2: A rule production

In 2.2, we can see that it is composed by two sides. The left hand side and the right hand side, delimited by ':'. On the LHS, 'liss' is a non-terminal and on the RHS, it is composed by the terminal 'program' followed by two non-terminals. This is the syntax of one production rule of the grammar.

Now let's speak about the entire syntax of the LISS.

3

LISS LANGUAGE

LISS (da Cruz and Henriques, 2007a) -that stands for Language of Integers, Sequences and Sets- is an imperative programming language, defined by the Language Processing members (Pedro Henriques and Leonor Barroca) at UM for teaching purposes (compiler course).

The idea behind the design of LISS language was to create a simplified version of the more usual imperative languages although combining functionalities from various languages.

It is designed to have atomic or structured integer values, as well as, control statements and block structure statements.

Now, let's explain in the next sections the basic statements of the language and its data types, using a context free gammar.

3.1 LISS DATA TYPES

There are 5 types available. From atomic to structured types, they are known as : integer, boolean, array, set and sequence.

Used for declaring a variable in a program, the data type gives us vital information for understanding what kind of value we are dealing with.

Let's obverse a LISS code example:

```
1  a -> integer;
2  b -> boolean;
3  c -> array size 5,4;
4  d -> set;
5  e -> sequence;
```

Listing 3.1: Declaring a variable in LISS

As we can see in Listing 3.1, some variables ('a','b','c','d' and 'e') are being declared each one associated to a type ('integer', 'boolean', 'array', 'set' and 'sequence'). Syntactically, in

LISS, this is done by writing the variable name followed by an arrow and the type of the variable (see Listing 3.2).

```
variable_declaration : vars '->' type ';'
                     ;
vars : var (',' var )*
   ;
var : identifier value_var
  ;
value_var :
        | '=' inic_var
        ;
type : 'integer'
   | 'boolean'
   | 'set'
   | 'sequence'
   | 'array' 'size' dimension
   ;
dimension : number (',' number )*
        ;
inic_var : constant
       | array_definition
       | set_definition
       | sequence_definition
       ;
constant : sign number
       | 'true'
       | 'false'
       ;
sign :
   | '+'
   | '-'
   ;
```

Listing 3.2: CFG for declaring a variable in LISS

Variables that are not initialized, have a default value (according to Table 1).

Table 1.: LISS data types

Type	Default Value
boolean	false
integer	0
array	[0,...,0]
set	{}
sequence	nil

Additionally, we may change the default values of the variables by initializing them with a different value (see an example in Listing 3.3). This can be made by writing an equal symbol after the variable name and, then, inserting the right value according to the type (see example in Listing 3.2).

```
a = 4, b -> integer;
t = true -> boolean;
vector1 = [1,2,3], vector2 -> array size 5;
a = { x | x<10} -> set ;
seq1 = <<10,20,30,40,50>>, seq3 = <<1,2>>, seq2 -> sequence;
```

Listing 3.3: Initialize a variable

Now, let's define which types are, correctly, associated with the arithmetic operators and functions in LISS (see Table 2).

Table 2.: Operations and signatures in LISS

Operators && Functions	**Signatures**
+ (add)	integer x integer -> integer
- (subtract)	integer x integer -> integer
|| (or)	boolean x boolean -> boolean
++ (union)	set x set -> set
/ (division)	integer x integer -> integer
* (multiply)	integer x integer -> integer
&& (and)	boolean x boolean -> boolean
** (intersection)	set x set -> set
== (equal)	integer x integer -> integer; boolean x boolean -> boolean
!= (not equal)	integer x integer -> integer; boolean x boolean -> boolean
<(less than)	integer x integer -> boolean
>(greater than)	integer x integer -> boolean
<= (less than or equal to)	integer x integer -> boolean
>= (great than or equal to)	integer x integer -> boolean
in (contains)	integer x set -> boolean
tail	sequence -> sequence
head	sequence -> integer
cons	integer x sequence -> sequence
delete	integer x sequence -> sequence
copy	sequence x sequence -> void
cat	sequence x sequence -> void
isEmpty	sequence -> boolean
length	sequence -> integer
isMember	integer x sequence -> boolean

So, in Table 2, we list the operators and functions, available in LISS, and their signature. In order to understand the table better, we will explain how to read the table and its signature with one example.

Consider the symbol '+' (Table 2), indicates that both operands must be of type integer. The result of that operation, indicated by the symbol '->', will be an integer. Semantically, operations must be valid according to Table 2; otherwise the operations would be incorrect and throw an error.

Arrays. LISS supports a way of indexing a collection of integer values such that each value is uniquely addressed. LISS also supports an important property of multidimensionality.

Called as 'array', it is considered to be a static structured type due to the fact that its dimensions and maximum size of elements in each dimension is fixed at the declaration time.

The operations defined over arrays are:

1. *indexing*
2. *assignment*

Arrays can be initialized, in the declaration section, partially or completely in each dimension. For example, consider an array of dimension 3x2 declared in the following way:

```
array1 = [[1,2],[5]] -> array size 3,2;
```

Thi is equivalent to the initialization below:

```
array1 = [[1,2],[5,0],[0,0]] -> array size 3,2;
```

Notice that the elements that are not explicitly assigned, are initialized with the value 0 (see Table 1).

The grammar for array declaration and initialization is shown below.

```
array_definition : '[' array_initialization ']'
                 ;

array_initialization : elem (',' elem)*
                     ;

elem : number
```

```
8       | array_definition
9       ;
```

Sets. The type *set*, in LISS, is a collection of integers with no repeated numbers.

It is defined by an expression, in a comprehension, instead of by enumeration of its element. A *set* variable can have an empty value and, syntactically, this is done by writing '{}'.

To define a set by comprehension, the free variable and the expression shall be return between curly brackets. The 'identifier' (free variable) is separated from the expression by an explicit symbol '|'.

The expression is built up from relational and boolean operators to define an integer interval.

The operations defined for sets are :

1. *union*
2. *intersection*
3. *in* (membership)

Let's see an example of its syntax below:

```
1   set1 = {x | x < 6 && x > -7} -> set;
```

This declaration defines a set including all the integers from -7 to 6 (open interval) and others numbers are not included in the set.

The syntax for set declaration and initialization is :

```
1   set_definition : '{' set_initialization '}'
2                  ;
3
4   set_initialization :
5                      | identifier '|' expression
6                      ;
```

Sequences. Considered as a dynamic array of one dimension, the type sequence is a list of ordered integers. But, in opposition to the concept of an array, its size is not fixed; this means that it grows dinamicallly at run time like a linked list. A sequence can have the empty value (syntactically done by writing '<<>>'). If not empty, the sequence value is defined by enumerating its components (integers) in the right order. Let's see deeper with one example:

```
c=<<1,2,3>> -> sequence;
```

Listing 3.4: Example of valid operations using sequence on LISS

In the example of Listing 3.4 the sequence is defined by three numbers (3,2,1).
The operations defined for the sequence are:

1. *tail* (all the elements but the first)
2. *head* (the first element of the sequence)
3. *cons* (adds an element in the head of the sequence)
4. *delete* (remove a given element from the sequence)
5. *copy* (copies all the elements to another sequence)
6. *cat* (concatenates the second sequence at the end of the first sequence)
7. *isEmpty* (true if the sequence is empty)
8. *length* (number of elements of the sequence)
9. *isMember* (true if the number is an element of the sequence)

Those operations will be explained further and deeper.

The grammar below defines how to declare a sequence:

```
sequence_definition : '<<' sequence_initialization '>>'
                    ;

sequence_initialization :
                        | values
                        ;

values : number (',' number )*
       ;
```

3.1.1 *LISS lexical conventions*

Once you've declared a variable of a certain type, you cannot redeclare it again with the same name.

The variable name must be unique (see Listing 3.5).

```
program single_variable_name{
  declarations
    int=1 -> integer;
    int=true -> boolean; //cannot declare this variable with this name
       (already exists)
  statements
}
```

Listing 3.5: Conflicts with variable names

Keywords cannot be used as variable names.

For example, you cannot declare a variable with the name *array* due to the fact that *array* is a keyword in LISS (in this case, a type).

See the example in Listing 3.6.

```
array -> array size 3,4; //variable 'array' cannot be declared as a
   name
integer -> integer;
```

Listing 3.6: Conflicts with keyword names

Variable names contain only letters and numbers, or the underscore sign. However the first character of the variable name must be a letter (lower or upper case). See the example below:

```
My_variable_1
MyVariable1
```

Numbers are composed of digits (one or more). Nothing more is allowed.

See example below:

```
1562
1
```

A string is a sequence of n-characters enclosed by double quotes.

See example below:

```
"This is a string"
```

3.2 LISS BLOCKS AND STATEMENTS

A LISS program is always composed of two parts: declarations and statements (a program block). LISS language is structured with a simple hierarchy. And this is done by structuring LISS code as a block.

Any program begins with a name then appear the declaration of variables and subprograms. After that appear the flow of the program by writing statements.

Let's see one example (see Listing 3.7).

```
program sum{
  declarations
      int=2 -> integer;
  statements
      writeln(int+3);
}
```

Listing 3.7: The structure of a LISS program (example)

So a program in LISS begins by, syntactically, writing 'program' and then the name of the program (in this case, the name is 'sum'). A pair of curly braces delimits the contents of the program; that is done by opening it after the name of the program and closing it at the end of the program. After the left brace, appear the declaration and statement blocks.

As in a traditional imperative language (let's compare 'C language'), if we don't take the habit of declaring the variable always in a certain part of the code, it becomes confusing. This makes the programmer's life harder to understand the code when the code is quite long.

So, in LISS, we always declare variables first (syntactically written by 'declarations') and then the statements (syntactically written by 'statements'). This is due to the fact that LISS wants to help the user to create solid and correct code. And in this case, the user will always know that all the variable declarations will be always at the top of the statements and not randomly everywhere (see grammar in Listing 3.8).

```
liss : 'program' identifier body
   ;

body : '{'
     'declarations' declarations
     'statements'   statements
     '}'
   ;
```

Listing 3.8: CFG for program in LISS

3.2.1 *LISS declarations*

The declaration part is divided into two other parts: variable declarations and subprogram declarations, both optional.

The first part is explained in section 3.1; the subprogram part will be discussed later in section 3.3.

This part is specified by the following grammar (see Listing 3.9).

```
declarations : variable_declaration* subprogram_definition*
             ;
```

Listing 3.9: CFG for declarations in LISS

3.2.2 *LISS statements*

As said previously, under the statements part, we control and implement the flow of a LISS program. In LISS, we may write none or, one or more statements consecutively.

Every statement ends with a semicolon, unless two type of statements (conditional and cyclic statements) as shown in Listing 3.10.

```
statements : statement*
           ;
statement : assignment ';'
          | write_statement ';'
          | read_statement ';'
          | function_call ';'
          | conditional_statement
          | iterative_statement
          | succ_or_pred ';'
          | copy_statement ';'
          | cat_statement ';'
          ;
```

Listing 3.10: CFG for statements in LISS

Let's see one example of a LISS program which shows how the language shall be used (see Listing 3.11).

```
program factorial{
  declarations
      res=1, i -> integer;
  statements
      read(i);
```

```
6         for(j in 1..i){
7             res=res*j;
8         }
9         writeln(res);
10    }
```

Listing 3.11: Example of using statements in LISS

Assignment. This statement assigns, as it is called, values to a variable and it is defined for every type available on LISS. This operation is done by writing the symbol "=" in which a variable is assigned to the left side of the symbol and a value to the right side of the symbol.

Notice that an assignment requires that the variable on the left and the expression on the right must agree in type.

Let's see in Listing 3.12 an example.

```
1  program assignment1{
2    declarations
3      intA -> integer;
4      bool -> boolean;
5    statements
6      intA = -3 + 5 * 9;
7      bool = 2 < 8;
8  }
```

Listing 3.12: Example of assignment in LISS

In Listing 3.12, we can see assignment statements of integers and boolean types. Those assignments are correct, as noticed in the previous paragraphs, because they have the same type on the left and right side of the symbol equals (operations of integers assigned to a variable of integer type and operation of booleans assigned to a variable of boolean type).

The grammar that rules the assignment is shown at Listing 3.13.

```
1  assignment : designator '=' expression
2             ;
```

Listing 3.13: CFG for assignment in LISS

I/O. The input and output statements are also available in LISS.

The *read* operations, called syntactically as 'input' in LISS, assign a value to a variable obtained from the standard input and require to be an atomic value (in this case, only an integer value).

```
1  program input1{
```

```
  declarations
    myInteger -> integer;
  statements
    input(myInteger);
}
```

Listing 3.14: Example of input operation in LISS

Notice that, in Listing 3.14, the variable *myInteger* must be declared and must be integer otherwise the operations fails. The grammar that rules the input statement, is shown in Listing 3.15.

```
read_statement : 'input' '(' identifier ')'
               ;
```

Listing 3.15: CFG for input operation in LISS

The *write* operations, called syntactically as 'write' or 'writeln' in LISS, print an integer value in the standard output. Notice that 'write' operation only prints the value and doesn't move to a new line; instead, 'writeln' moves to a new line at the end.

Listing 3.16 shows some more examples.

```
writeln(4*3);
writeln(2);
writeln();
```

Listing 3.16: Example of output operations in LISS

Note that the write statement may have as assignment, an atomic value as well as an empty value or some complex arithmetic expression (see grammar in 3.17).

```
write_statement : write_expr '(' print_what ')'
                ;

write_expr : 'write'
           | 'writeln'
           ;

print_what :
           | expression
           ;
```

Listing 3.17: CFG for output operation in LISS

Function call. The function call is a statement that is available for using the functions created in the program under the section 'declarations' (as described in Section 3.2.1). This will allow reusing functions that were created by calling them instead of creating duplicated code.

See Listing 3.18 for a complete example.

```
program SubPrg {

 declarations

   a = 4, b= 5, c= 5 -> integer;
   d = [10,20,30,40], ev -> array size 4;


   subprogram calculate() -> integer
   {
     declarations
           fac = 6 -> integer;
           res = -16 -> integer;

       subprogram factorial(n -> integer; m -> array size 4) -> integer
           {
             declarations
                   res = 1 -> integer;
             statements
                   while (n > 0)
                   {
                       res = res * n;
                       n = n -1;
                   }

                   for (a in 0..3) stepUp 1
                   {
                     d[a] = a*res;
                   }
                   return res;
           }
     statements
           res = factorial(fac,d);
           return res/2;
   }


 statements
```

```
39
40      a = calculate();
41      writeln(a);
42      writeln(d);
43  }
```

Listing 3.18: Example of call function in LISS

In Listing 3.18, we can see that the function *calculate()*, called in the main program, and that is created under the declarations section.

The grammar who rules the function call is shown in Listing 3.19.

```
1   function_call : identifier '(' sub_prg_args ')'
2                 ;
3   sub_prg_args :
4                | args
5                ;
6   args : expression (',' expression )*
7      ;
```

Listing 3.19: CFG for call function in LISS

3.2.3 *LISS control statements*

LISS language includes some statements for controlling the execution flow at runtime with two different kind of behaviour.

The first one is called conditional statement and it has only one variant in LISS language (see Listing 3.20).

The second one is called cyclic statement or iterative statement, and it has two variants (see Listing 3.20).

```
1   conditional_statement : if_then_else_stat
2                         ;
3   iterative_statement : for_stat
4                       | while_stat
5                       ;
```

Listing 3.20: CFG for control statement in LISS

These control statements, mimics the syntax and the behaviour of other modern imperative language.

CONDITIONAL The if-statement, which is common across many modern programming languages, performs different actions according to decision depending on the truth value of a control conditional expression: an alternative 'else' block is also allowed (optional).

If the conditional expression evaluates 'true', the content of 'then' block will be executed. Otherwise, if the condition is 'false', the 'then' block is ignored; and if an 'else' block is provided it will be executed alternatively.

Let's see an example in Listing 3.21.

```
1 if(y==x)
2 then{
3   x=x+1;
4 }else{
5   x=x+2;
6 }
```

Listing 3.21: LISS syntax of a if statement

The code shown in Listing 3.21, means that the if-statement evaluates the conditional expression 'y==x'. If the expression, which must be boolean, is true, then every action in the 'then' block will be executed and the block 'else' will be ignored. Otherwise, if the condition is false, every action in the 'else' block is executed ignoring the 'then' block.

If the else-statement is not provided, the if-statement will finish and do not perform any actions.

The syntax of the if-statement in LISS is shown in Listing 3.22.

```
1 if_then_else_stat : 'if' '(' expression ')'
2                     'then' '{' statements '}'
3                     else_expression
4                     ;
5
6 else_expression :
7               | 'else' '{' statements '}'
8               ;
```

Listing 3.22: CFG for iterative statement in LISS

ITERATIVE We should take a look at the behaviour of each iterative control statement to understand it deeper.

The for-statement offers two variants to control the repetition. Normally, in a conventional way, the for-loop has a control variable which takes a value in a given range and step up or step down by a default or an explicit value.

In LISS, the control variable is set in a given integer interval defined by the lower and upper bounds. By default, the step is one, which means that the control variable is incremented by one at the end of each iteration but it is possible to increment or decrement it by a different value, setting it explicitly. Additionally, we may write a condition for filtering the values in the interval. This can be done as shown in the following example:

```
for(a in 1..10) stepUp 2 satisfying elems[a]==1{
  ...
}
```

Listing 3.23: LISS syntax of a for-loop statement

In Listing 3.23, the control variable 'a' is set to a range 1 to 10 and would be increased (due to the 'stepUp' constructor) by 2. Also there is a filter condition (after the 'satisfying' keyword) that restricts the values of 'a' to those that makes the condition 'elems[a]==1' true. Notice that the filter expression must be boolean.

After each cycle, the control variable will be incremented with value 2 and the filter condition tested again.

This is the first way of expressing the control in a for-loop statement. Let's see the second way in the sequel.

There is also the possibility to assign to the control variable the values in an array, like illustrated in the following example:

```
for(b inArray elems){
      ...
}
```

Listing 3.24: LISS syntax of a for-each statement on array

In Listing 3.24, the control variable 'b' is assigned with all of the elements of the array and begins with his lower index (zero) until his upper index (size of the array minus one). Notice that, in this case, we cannot apply an increment or decrement neither a filter condition.

The next grammar fragment describes the cycle 'for' in LISS:

```
for_stat : 'for' '(' interval ')' step satisfy
         '{' statements '}'
       ;
interval : identifier type_interval
       ;
type_interval : 'in' range
            | 'inArray' identifier
            ;
range : minimum '..' maximum
    ;
```

```
minimum : number
        | identifier
        ;
maximum : number
        | identifier
        ;
step :
    | up_down number
    ;
up_down : 'stepUp'
        | 'stepDown'
        ;
satisfy :
        | 'satisfying' expression
        ;
```

Listing 3.25: CFG for for-statement in LISS

Finally, the while-statement consists in a block of code that is executed repeatly until the control condition evaluates 'false'.

Each time that the 'while' block is performed, the conditional expression associated will be evaluated again to decide whether to repeat the execution of the statements in the block or to continue the normal program flow.

Let's see an example in Listing 3.26.

```
while (n > 0)
{
  res = res * n;
  pred n;
}
```

Listing 3.26: LISS syntax of a while-statement in LISS

In Listing 3.26, the while-statement is controled by the conditional expression 'n>0' that is evaluated at the beginning. If the condition is true, then all the actions that are inside the braces will be performed. Later, after executing all the actions, the condition will be evaluated again. If the condition remains 'true', then those actions would be executed again otherwise if the condition is false, the while-statement will be exited.

The syntax that rule the while-statement is shown below:

```
while_stat : 'while' '(' expression ')'
           '{' statements '}'
           ;
```

Listing 3.27: CFG for while-statement in LISS

3.2.4 *Others statements*

LISS language offers other statements to make it more expressive easing the codification of any imperative algorithm.

Succ/Pred. Those statements are available for incrementing or decrementing a variable. This is a common situation in modern programming languages, making life easier for the developers.

The keyword 'succ' means increment (successor) and the syntax 'pred' means decrease (predecessor). Only integer variables can be used with those constructors.

Listing 3.28 illustrates both statements.

```
1 succ int1;
2 pred int1;
```

Listing 3.28: Example of using succ/pred in LISS

As we can see in Listing 3.28, variable 'int1' is, first, incremented by 1 and then it is decremented also by 1.

Grammar of 'succ' and 'pred' in LISS is shown in Listing 3.29.

```
1 succ_or_pred : succ_pred identifier
2               ;
3 succ_pred : 'succ'
4           | 'pred'
5           ;
```

Listing 3.29: CFG for succ and pred in LISS

Copy statement. This statement is applied only to variables of type sequence. Basically, it copies one sequence to another sequence. Let's see an example in Listing 3.30.

```
1 copy(seq1,seq2);
```

Listing 3.30: Example of copy statement in LISS

Notice that 'copy' is a statement and not a function: it modifies the arguments but does not return any value.

In Listing 3.30, the statement 'copy' copies the content of the variable *seq1* to *seq2*.

The grammar for 'copy' statement is in Listing 3.31.

```
copy_statement : 'copy' '(' identifier ',' identifier ')'
               ;
```

Listing 3.31: CFG for copy statement in LISS

Cat statement.

'Cat' statement is simular to 'copy', it only operates with variables of type sequence. The behaviour of this statement is to concatenate a sequence to another sequence. Let's see an example in Listing 3.32).

```
cat(seq1,seq2);
```

Listing 3.32: Example of cat statement in LISS

In Listing 3.32, 'cat' concatenates the content of *seq2* to *seq1*. Again, 'cat' is not a function; it modifies the arguments instead of returning a value.

The grammar for cat-statement is shown in Listing 3.33.

```
cat_statement : 'cat' '(' identifier ',' identifier ')'
              ;
```

Listing 3.33: CFG for cat statement in LISS

3.3 LISS SUBPROGRAMS

In LISS, it is possible to organize the code by splitting the general block of statements into sub-programs. This allows the programmer to reuse or to give more clarity to his code by creating functions or procedures. Also, it is possible to create sub-programs inside sub-programs by using a nesting strategy.

The syntax that defines a sub-program in LISS is shown in Listing 3.34.

```
subprogram_definition: 'subprogram' identifier '(' formal_args ')'
   return_type f_body
                  ;
f_body : '{'
       'declarations' declarations
       'statements' statements
       returnSubPrg
       '}'
     ;
formal_args :
          | f_args
```

```
11              ;
12  f_args   : formal_arg (',' formal_arg )*
13        ;
14  formal_arg : identifier '->' type
15             ;
16  return_type :
17              | '->' typeReturnSubProgram
18              ;
19  returnSubPrg :
20               | 'return' expression ';'
21               ;
```

Listing 3.34: CFG for block structure in LISS

Note that every variable declared inside of a sub-program is local, and it can be accessed only by other nested sub-programs. However, variables declared in the program (not in a sub-program) are considered global and can be accessed by any sub-program. The usual scope rules are applied to LISS.

As can be inferred from the syntax above (Listing 3.34), the body of a sub-program is identical to the body of a program — the same declarations can be made and similar statements can be used.

3.4 EVOLUTION OF LISS SYNTAX

Due to the maturity of the language already done along the years, we have added some few but extra changes for a better experience of the programming language.

One of the first changes was concerned with declarations in order to avoid mixing functions and variable declarations. We, indirectly, teach the programmer by doing it in the right way. So we declare, first, the variables and then the functions.

```
declaration : variable_declaration * subprogram_definition *
            ;
```

Another change was to add punctuation after each statement (see Figure 3.35).

```
statement : assignment  ';'
          | write_statement  ';'
          | read_statement  ';'
          | conditional_statement
          | iterative_statement
          | function_call ';'
          | succ_or_pred  ';'
          | copy_statement  ';'
          | cat_statement  ';'
          ;
```

Listing 3.35: Function statement

Another change was adding also a 'cat_statement' rule which works with only sequences. It concatenates a sequence with another sequence.

Regarding arrays, it was previously possible to use any expression to access elements of the array. So it was possible to index with a boolean expression what does not make any sense. Now only integers are allowed (see in Listing 3.36).

```
elem_array : single_expression (',' s2=single_expression )*
           ;
```

Listing 3.36: Rule element of array

In the previous version of LISS, it was allowed to create a boolean expression associating relational operators, but we decided to change that and not permit associativity; only able to create one boolean expression (see Listing 3.37). It does not make sense to have an expression like that : '3 == 4 == 5 != 6'.

```
expression : single_expression (rel_op single_expression )?
           ;
```

Listing 3.37: Rule for Boolean expression

We added the possibility of using parenthesis on expressions (see Listing 3.38).

```
factor: '(' expression ')'
      ;
```

Listing 3.38: Rule factor

We changed the rules of two pre-defined functions: 'cons' and 'del'. These functions were working both in the same way. Waiting for an expression and a variable as arguments. Now, we decide to change that allowing to expression as arguments giving more expressive power to those functions (see Listing 3.39).

```
cons // integer x sequence -> sequence
     : 'cons' '(' expression ',' expression ')'
     ;

delete // del : integer x sequence -> sequence
       : 'del' '(' expression ',' expression ')'
       ;
```

Listing 3.39: Rule cons and delete

Besides adding some improvements to the grammar, we additionally deleted a rule which we thought not necessary to control the for-statement (see Listing 3.40).

```
type_interval : 'in' range
              | 'inArray' identifier
              //| 'inFunction' identifier
              ;
```

Listing 3.40: Rule type interval

Last but not least, we also added comments to the programming language, giving more power to the programmer.

```
fragment
COMMENT
    : '/*'.*?'*/' /* multiple lines comment*/
    | '//'~('\r' | '\n')* /* single line comment*/
    ;
```

Listing 3.41: Lexical rule for Comment

4

TARGET MACHINE: MIPS

MIPS, from Microprocessor without Interlocked Pipeline Stages, is a Reduced Instruction Set Computer (RISC) developed by MIPS Technologies. Born in 1981, a team led by John L. Hennessy at Stanford University began to work on the first MIPS processor.

The main objective for creating MIPS, was to increase performance with deep pipelines, a main problem back to the 80's. Some instructions, as division, take a longer time to complete; if the CPU needs to wait that the division ends before passing to the next instruction into the pipeline, the total time is greater. If it can be done without that waiting time, the total process will be faster.

As MIPS solved those problems, it was primarly used for embedded systems and video games consoles (which requires a lot of arithmetic computation).

Now, the architecture of MIPS, along the years, has gained maturity and provides different versions of it (MIPS32, MIPS64....) [1].

Figure 2 [2] illustrate the architecture of MIPS.

1 according to https://imgtec.com/mips/architectures (See also wikipedia https://en.wikipedia.org/wiki/MIPS_instruction_set)

2 from https://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/MIPS_Architecture_(Pipelined).svg/300px-MIPS_Architecture_(Pipelined).svg.png

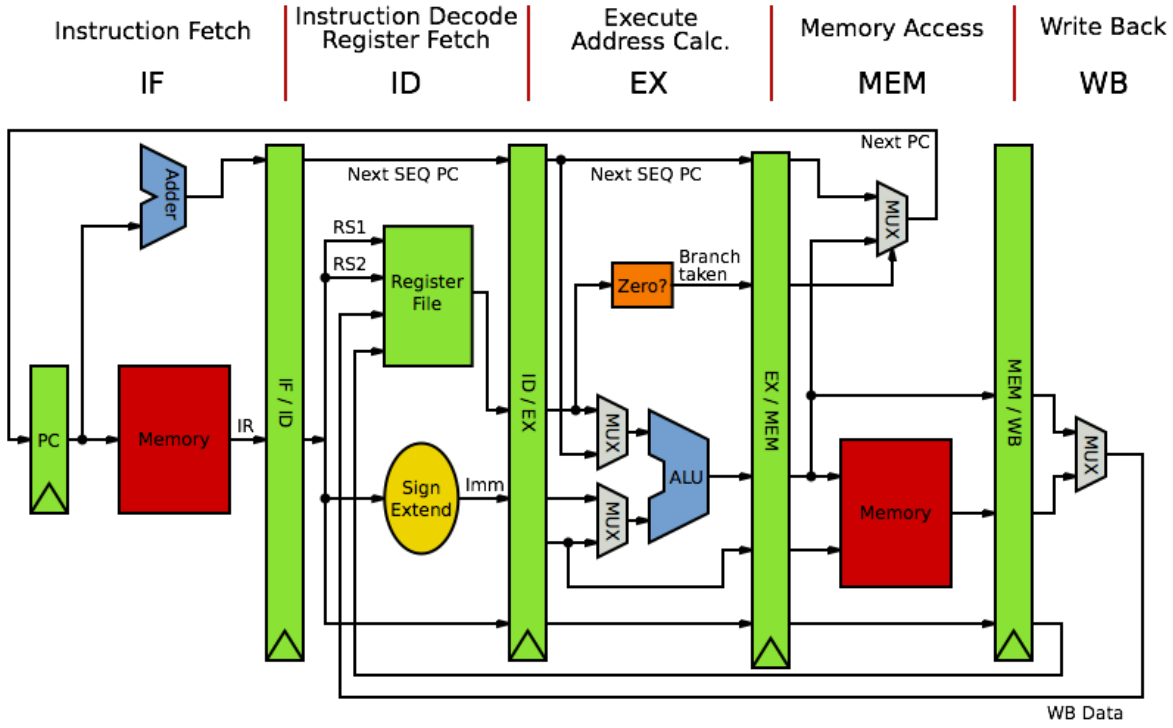


Figure 2.: MIPS architecture

In this chapter, we will talk about the architecture components and assembly of MIPS 32-bit version.

4.1 MIPS COPROCESSORS

MIPS was born for solving complex arithmetic problems by reducing the time consumed in those operations. This is attained through the implementation of coprocessors within MIPS.

MIPS architecture includes four coprocessors respectively, CP0, CP1, CP2 and CP3:

1. Coprocessor 0, denoted by *CP0*, is incorporated in the CPU chip; it supports the virtual memory system and exception handling (also known as the *System Control Coprocessor*).
2. Coprocessor 1, denoted by *CP1*, is reserved for floating point coprocessor.
3. Coprocessor 2, denoted by *CP2*, is reserved for specific implementations.
4. Coprocessor 3, denoted by *CP3*, is reserved for the implementations of the architecture.

Notice that coprocessor *CP0*, translates virtual addresses into physical addresses, manages exceptions, and handles switch between kernel, supervisor and user modes.

4.2 MIPS CPU DATA FORMATS

The CPU of MIPS defines four differents formats:

- *Bit* (1 bit, b)
- *Byte* (8 bits, B)
- *Halfword* (16 bits, H)
- *Word* (32 bits, W)

4.3 MIPS REGISTERS USAGE

MIPS architecture has 32 registers dedicated and there are some conventions to use those registers correctly. Table 3 summarizes those registers, and their usage.

Table 3.: MIPS registers

Name	Number	Use	Callee must preserve?
$zero	$0	has constant 0	No
$at	$1	register reserved for assembler (temporary)	No
$v0 - $v1	$2 - $3	register reserved for returning values of functions, and expression evaluation	No
$a0 - $a3	$4 - $7	registers reserved for function arguments	No
$t0 - $t7	$8 - $15	temporary registers	No
$s0 - $s7	$16 - $23	saved temporary registers	Yes
$t8 - $t9	$24 - $25	temporary registers	No
$k0 - $k1	$26 - $27	register reserved for OS kernel	N/A
$gp	$28	global pointer	Yes
$sp	$29	stack pointer	Yes
$fp	$30	frame pointer	Yes
$ra	$31	return address	N/A

Note: N/A (Not applicable)

Table 3 is composed of 4 columns:

1. *Name* displays the identifier of the registers available in MIPS. Those identifiers will be used as operands of MIPS instructions.

2. *Number* column defines the number of each register. This number can also be used to refer to the register in an instruction.
3. *Use* column refers to the meaning/definition of each register.
4. *Callee must preserve?* column provides information about the volatility of the register (used when a function is called).

Beside those 32 registers, 3 more registers are dedicated to the CPU.
And they are known by:

- *PC* - Program Counter register
- *HI* - Multiply and Divide register higher result
- *LO* - Multiply and Divide register lower result

PC is the register which holds the address of the instruction that is being executed at the current time; *HI* and *LO* registers have different usage according to the instruction that is being executed. In this case, let's see what context they have:

- when there is a multiply (*mul* instruction) operation, the *HI* and *LO* registers store the result of integer multiply.
- when there is a multiply-add (*madd* instruction) or multiply-subtract (*msub* instruction) operation, the *HI* and *LO* register store the result of integer multiply-add or multiply-subtract.
- when there is a division (*div* instruction) operation, the *HI* register store the remainder of the division and the *LO* register store the quotient of the division operation.
- when there is a multiply-accumulate (instruction) operation, the *HI* and *LO* registers store the accumulated result of the operation.

See an overview of the MIPS registers in Figure 3.

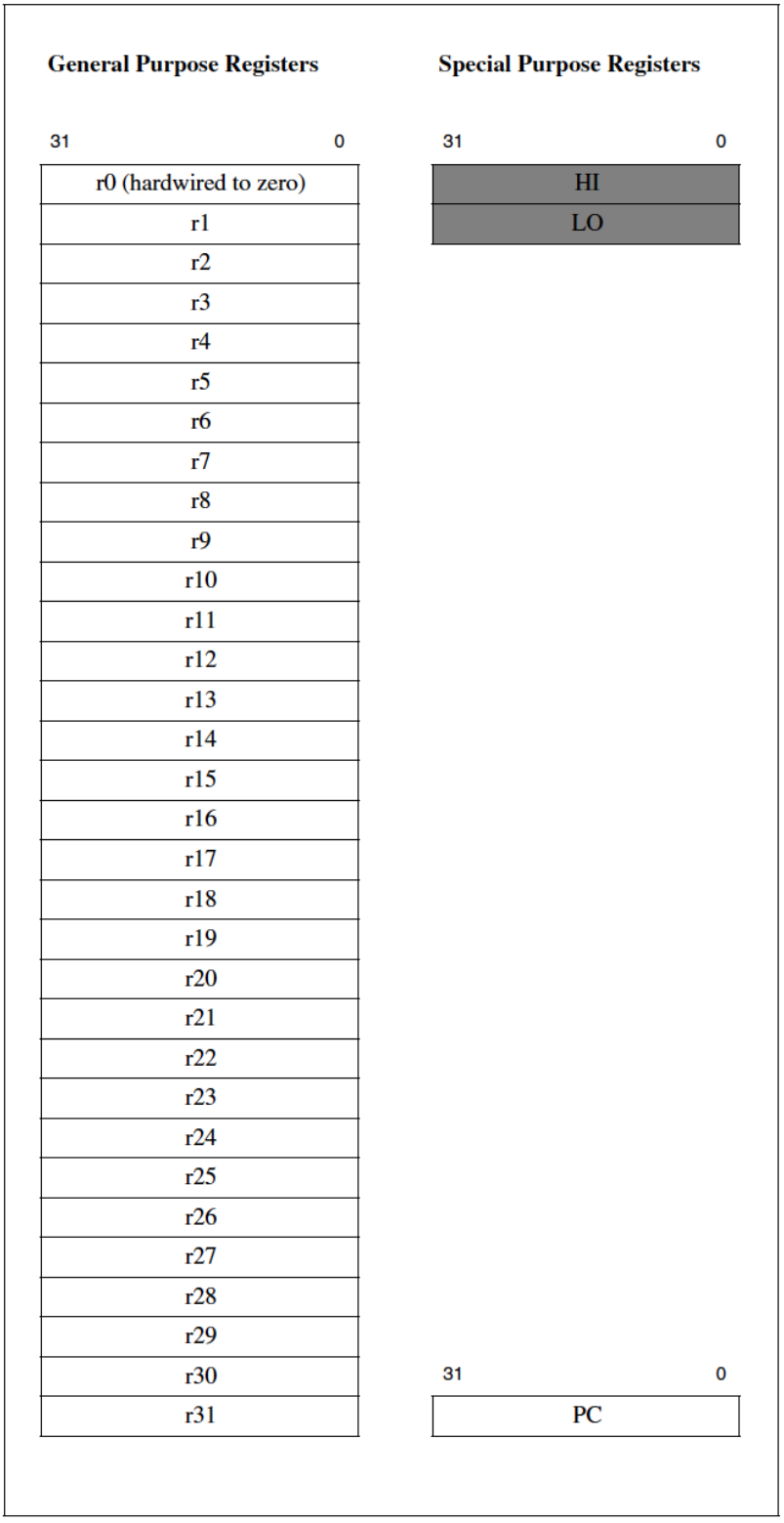


Figure 3.: MIPS register

4.4 MIPS INSTRUCTION FORMATS

Instructions, in MIPS, are divided into three types:

- R-Type
- I-Type
- J-Type

Each instruction is denoted by an unique mnemonic that represents the correspondent low-level machine instruction or operation.

Next sections provide the necessary details.

4.4.1 *MIPS R-Type*

R-Type instruction refers a register type instruction (it is the most complex type in MIPS). The idea behind that instruction is to operate with registers only.

This type has the following format in MIPS (see Listing 4.1).

```
1 OP rd, rs, rt
```

Listing 4.1: R-Type instruction format

In Listing 4.1, the instruction is composed of one mnemonic, denoted by *OP*, and three operands, denoted by *rd* (destination register), *rs* (source register), *rt* (another source register).

The R-Type instruction format as the following mathematical semantics:

```
1 rd = rs OP rt
```

To understand better this instruction, let's see an example of one R-Type instruction in MIPS (see Listing 4.2).

```
1 add $t1, $t1, $t2
```

Listing 4.2: Example of a R-Type instruction

The instruction shown in Listing 4.2 means that register $t1 shall be added (due to *add* mnemonic) to register $t2 and their sum (the result) stored in register $t1.

The following equivalence explains that meaning.

$$
\begin{aligned}
OP\ rd,\ rs,\ rt &\iff rd\ =\ rs\ OP\ rt \\
&\Downarrow \\
add\ \$t1,\ \$t1,\ \$t2 &\iff \$t1\ =\ \$t1\ add\ \$t2 \\
&\Downarrow \\
\$t1\ &=\ \$t1\ +\ \$t2
\end{aligned}
$$

Table 4 defines the bit-structure of a R-Type instruction in a 32-bit machine.

Table 4.: R-Type binary machine code

opcode	**rs**	**rt**	**rd**	**shift (shamt)**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Let's explain each of the columns in Table 4.

- **opcode** defines the instruction type. For every R-Type instruction, *opcode* is set to the value 0. The *opcode* field is 6 bits long (bit 31 to bit 26).
- **rs** this is the first source register; it is the register where it will load the content of the register to the operation.The *rs* field is 5 bits long (bit 25 to bit 21).
- **rt** this is the second source register (same behaviour as *rs* register). The *rt* field is 5 bits long (bit 20 to bit 16).
- **rd** this is the destination register; it is the register where the results of the operation will be stored. The *rd* field is 5 bits long (bit 15 to bit 11).
- **shift amount** the amount of bits to shift for shift instructions. The *shift* field is 5 bits long (bit 10 to bit 6).
- **function** specify the operation in addition to the *opcode* field. The *function* field is 6 bits long (bit 5 to bit 0).

Let's see an example of a R-Type instruction and its transformation to machine code in Table 5.

$$add\ \$t0,\ \$t0,\ \$t1$$
$$\Downarrow$$
$$add\ \$8,\ \$8,\ \$9$$
$$\Downarrow$$
$$(8)_{10} = (01000)_2$$
$$(9)_{10} = (01001)_2$$
$$add\ instruction\ (funct\ field) = (100000)_2$$
$$\Downarrow$$

opcode (6bits)	rs (5bits)	rt (5bits)	rd (5bits)	shift (shamt) (5bits)	funct (6bits)
000000	01000	01001	01000	00000	100000

Table 5.: Transformation of R-Type instruction to machine code

In Table 5, the instruction 'add $to, $to, $t1' will be normalized with the name of the register according to the number associated for the register in MIPS (see Table 3). Then a conversion operation is applied to the two register numbers (*8* and *9*), translating them into their binary number with 5 bits long. Also we give the information for the *add* instruction, which is set for the MIPS architecture (not predictable).

After that, we complete the table for R-Type instruction according to Table 4 with the informations available and the restriction/rules associated to R-Type instruction in MIPS.

Notice that the *opcode* field for R-Type instruction are set to the value 0 (according to the explanation in Table 4).

4.4.2 *MIPS I-Type*

I-Type instruction is a set of instructions which operate with an immediate value and a register value.

Several different Immediate (*I-Type*) instructions formats are available.

Let's see those diferents formats for this type in Table 6.

31 – 26	25 —- 21	20 – 16	15 ——— 11	10 ——- 6	5 —— 0
opcode	rs	rt	immediate		
opcode	rd	offset			
opcode	offset				
opcode	rs	rt	rd	offset	
opcode	base	rt	offset		function

Table 6.: Distinct I-Type instruction formats

In Table 6, there are 5 differents instruction formats which corresponds to different bit structures as illustrated.

The most frequent MIPS I-Type instruction is the first one, denoted as Imm16 (Immediate instruction with 16 bits immediate value), is used for logical operands, arithmetic signed operands, load/store address byte offsets and PC-relative branch signed instruction displacements (see Table 7).

31 — 26	25 — 21	20 — 16	15 ————— 0
opcode	rs	rt	immediate

Table 7.: Immediate (I-Type) Imm16 instruction format

Let's see examples of Imm16 instruction:

```
addi $to, $to, 10 // Arithmetic operation
ori $to, $t1, 5   // Logical operation
beq $to, $t1, 1    // Conditional branch operation
lw $to, array1($to) //Data transfer operation
```

The second instruction, denoted as Immediate Off21 instruction (Immediate instruction with 21bits offset), is used for comparing a register against zero and branch (offset field is larger than the usual 16-bit field (immediate field of the first instruction from the table above)). See Table 8.

31 — 26	25 — 21	20 ————— 0
opcode	rd	offset

Table 8.: Immediate (I-Type) Off21 instruction format

The third instruction, denoted as Immediate Off26 instruction (Immediate instruction with 26 bits offset), is used for PC-relative branches with very large displacements (unconditional branches (BC mnemonic instruction) & branch-and-link (BALC mnemonic instruction) with a 26-bit offset,). See Table 9.

31 — 26	25 ——————————— 0
opcode	offset

Table 9.: Immediate (I-Type) Off26 instruction format

The fourth instruction, denoted as Immediate Off11 instruction (Immediate instruction with 11 bits offset), is used for the newest encodings of coprocessor 2 load and store instructions (LWC2, SWC2, LDC2, SWC2). See Table 10.

31 — 26	25 —— 21	20 ——— 16	15 ——— 11	10 ————— 0
opcode	rs	rt	rd	offset

Table 10.: Immediate (I-Type) Off11 instruction format

Finally, the last one (fifth instruction), denoted as Immediate Off9 instruction (Immediate instruction with 9 bits offset), is used for SPECIAL3 instructions such as EVA memory access (*LBE* mnemonic). Also this is primarly used for instruction encodings that have been moved, such as *LL* menmonic and *SC* mnemonic instruction. See Table 11.

31 — 26	25 —— 21	20 ——— 16	15 ————— 7	6	5 ————— 0
opcode	base	rt	offset	0	function

Table 11.: Immediate (I-type) Off9 instruction format

Notice that, for the project related to the thesis, only the first instruction type (Immediate (I-Type) Imm16 instruction format) was used. The other instruction formats are not really important for this project.

4.4.3 *MIPS J-Type*

J-Type instructions are instructions which jump to a certain address. Let's see his format in Table 12.

31 — 26	25 ——————————————————————— 0
opcode	address

Table 12.: J-Type instruction format

In Table 12, 6 bits are associated to the *opcode* field and 26 bits for the *address* field. But notice that in MIPS, addresses are 32 bits long.

For solving that, MIPS use a technique which leads to shift the address left by 2 bits and then combine 4 bits with the 4 high-order bits of the PC in front of the address.

Examples of J-Type formats can be seen in Listing 4.3.

```
1  jal writeln  // Jump and link instruction
```

```
2  jr $ra    // Jump register instruction
3  j writeln      // Jump instruction
```

Listing 4.3: Examples of J-Type instruction

In Listing 4.3, we see three different types of jump instruction. The first one example, is a *jal* instruction and it means 'jump and link' in an extensive way. Basically, it jump to the branch written in front of the *jal* nomenclature and stores the return address (instantly) to the return address register ($ra; $31). In this way, the programmer don't need to use some instructions for saving the return address and continue the flow of the execution code.

The second example, is a *jr* instruction and it means 'jump to an address stored in a register'. Notice that registers are available in the MIPS architecture.

The third and last example is a *j* instruction and this is a 'jump instruction'. Summing it up, it jumps to the branch written in front of the letter *j*, which is in this case *writeln*.

4.5 MIPS ASSEMBLY LANGUAGE

MIPS language is divided into 2 parts (Data and Text parts).

4.5.1 *MIPS data declarations*

This section is used for declaring variable names used in the program. Variables declared are allocated in the main memory (RAM) and must be identified with a particular nomenclature denoted as *.data*. It is used for declaring global variables, principally.

Then comes the part when the variable names are declared.

Let's see the format for declaring a variable name in Listing 4.4.

```
1  name: storage_type value(s)
```

Listing 4.4: Syntax format of data declarations in MIPS

In Listing 4.4, the *name* field refers to the name of the variable.

The *storage_type* refers to the type of the variable that can be:

- *.ascii* store a string in memory without a null terminator.
- *.asciiz* store a string in memory with the null terminator.
- *.byte* store 'n' bytes contiguously in memory.
- *.halfword* store 'n' 16-bit halfwords contiguously in memory.
- *.word* store 'n' 32-bit words contiguously in memory.

- *.space* store a certain number of bytes of space in memory.

Lastly, the *value(s)* field refers to the value of the type associated.

Let's see some example for declaring some variables in MIPS in Listing 4.5.

```
1  .data  # Tells assembler we're in the data segment
2    val:  .word  10
3    str:  .ascii  "Hello, world"
4    num:  .byte  0x01, 0x02
5    arr:  .space 100
```

Listing 4.5: Examples for declaring variables in MIPS

In Listing 4.5, there are 4 different types under the *data* section.

The variable *val* contains the value '10' and the size of the variable is 32 bits.

The variable *str* contains the string 'Hello World' and the size of the variable is the same size as the string.

The variable *num* stores the listed value(s) (which appears after the *.byte* nomenclature) as 8 bit bytes. In this example, it will be '0x00000201'.

The variable *arr* reserves the next specified number of bytes in the memory, which will be 100 bytes reserved for that variable.

4.5.2 *MIPS text declarations*

This section contains the program code and follows a specific syntax starting with the keyword *.text*.

As all programming languages, there is a starting point in the code that must be designated as *main:*. Each of the assembly language statements in MIPS (written after the *main:* field) are executed sequentially (excepted loop and conditional statements).

Let's see an example in Listing 4.6.

```
1  .text
2    main:
3      li $t0, 5
4      li $t1, 10
5      mul $t0, $t0, $t1
```

Listing 4.6: Example of Text declarations in MIPS

In Listing 4.6, we see the *.text* which begins the code of the program and the *main:* which shows where the code execution must start.

Below the keyword *main:* appears all the instruction of the program code.

In this case, it will load two numbers in different registers and multiply them (see Section 4.6 to understand those instructions).

Notice that the code will execute sequentially.

Also, in the text part beside of the code execution flow, we can write the name of branches for executing some jump instructions. This means that every jump instruction with a name associated, will see if that name is under the text part. Like that when a jump instruction is available it can jump to the name associated.

And for this purpose, we need to add some context to the MIPS jump instruction code and understand it better.

In this case, we need to replicate the same syntax as the *main:* field but with the correct name of the condition or the loop (also inside of the text declarations parts). Like that, MIPS knows where it must jump for the next instruction. Let's look an example in Listing 4.7.

```
1  .data
2  .text
3    main:
4      li $to, 5
5      li $t1, 5
6      mul $to, $to, $t1
7      jal jump_condition #needs to jump to the field jump_condition
8      li $to, 4
9      li $vo, 10
10     syscall
11   jump_condition:  #syntax for jump and conditional instruction in mips
12     li $t1, 5
13     jr $ra
```

Listing 4.7: Example of a loop declaration in MIPS

As we can see in Listing 4.7, we have a *jal* instruction available and a name associated next to the instruction. This name must be included under the *.text* section, because the name is the name of the branch from where the jump instruction will jump. If the name isn't in the MIPS assembly code, then the program cannot execute the assembly code. But in the example case, we can see that the name is available below as *jump_condition:*. So this means that the *jal* instruction will jump to that line and continue the code execution flow there.

Also, in MIPS, there is the possibility to include inline comments in the code using the symbol **#** on a line (see Listing 4.8).

```
1  var1:    .word 3 # create a single integer variable with initial value 3
```

Listing 4.8: Example of a comment in MIPS

Let's see the template for a MIPS assembly language program in Listing 4.9.

```
# Comment giving name of program and description of function
# Template.s
# Bare-bones outline of MIPS assembly language program

    .data       # variable declarations follow this line
                # ...

    .text       # instructions follow this line

      main:     # indicates start of code (first instruction to execute)
                # ...
```

Listing 4.9: Template of a MIPS assembly language

4.6 MIPS INSTRUCTIONS

MIPS has 6 type of instructions :

- instructions for data transfer
- instructions for arithmetic operations
- instructions for logical operations
- instructions for bitwise shift
- instructions for conditional branch
- instructions for unconditional branch

Let's see some examples of those instructions and their meanings.

Table 13.: Example of Data transfer instruction in MIPS

Name	Instruction Syntax	Meaning	Format	Opcode	Funct
Store word	sw $t,C($s)	Memory[$s + C] = $t	I	0x2B	N/A
Load word	lw $t,C($s)	$t = Memory[$s + C]	I	0x23	N/A
Load immediate	li $t, C	$t = C	I	0x9	N/A

Table 14.: Example of Arithmetic instruction in MIPS

Name	Instruction Syntax	Meaning	Type	Opcode	Funct
Add	add $d, $s, $t	$d = $s + $t	R	0x0	0x20
Add immediate	addi $t, $s, C	$t = $s + C (signed)	I	0x8	N/A
Subtract	sub $d, $s, $t	$d = $s - $t	R	0x0	0x22
Move	move $to, $t1	$to = $t1	R	0x0	0x21
Multiply	mul $s, $t, $d	$s = $t * $d LO = $t * $d (upper 32bits) HI = $t * $d (lower 32bits)	R	0x0	0x19
Divide	div $s, $t, $d	$s = $t / $d LO = $t / $d HI = $t % $d	R	0x0	0x1A

Table 15.: Example of Logical instruction in MIPS

Name	Instruction Syntax	Meaning	Format	Opcode	Funct
Set on less than	slt $d,$s,$t	$d = ($s <$t)	R	0x0	0x2A
Or	or $d,$s,$t	$d = $s || $t	R	0x0	0x25
And	and $d,$s,$t	$d = $s & $t	R	0x0	0x24
Set on less than unsigned	sltu $d,$s,$t	$d = ($s <$t)	R	0x0	0x2B
Exclusive or immediate	xori $d,$s,C	$d = $s ^C	I	0xE	N/A

Table 16.: Example of Bitwise Shift instruction in MIPS

Name	Instruction Syntax	Meaning	Format	Opcode	Funct
Shift left logical immediate	sll $d,$t,shamt	$d = $t <<shamt	R	0x0	0x0
Shift right logical immediate	srl $d,$t,shamt	$d = $t >>shamt	R	0x0	0x2
Shift left logical	sllv $d,$t,$s	$d = $t <<$s	R	0x0	0x4
Shift right logical	srlv $d,$t,$s	$d = $t >>$s	R	0x0	0x6

Some explanation must be provided for understanding the tables shown previously:

- **PC** means Program Counter.
- **target** means the name of the target (used for jump instructions).
- **C** means constants.
- `0x..` means a hexadecimal format number.
- **N/A** means Not Applicable.

Table 17.: Example of Conditional Branch instruction in MIPS

Name	Instruction Syntax	Meaning	Format	Opcode	Funct
Branch if equal zero	beqz $s, jump	if($s==0) go to jump address	I	0x4	N/A
Branch on not equal	bne $s, $t, C	if ($s != $t) go to PC+4+4*C	I	0x5	N/A
Branch on equal	beq $s, $t,C	if ($s == $t) go to PC+4+4*C	I	0x4	N/A

Table 18.: Example of Unconditional Branch instruction in MIPS

Name	Instruction Syntax	Meaning	Format	Opcode	Funct
Jump	j target	PC = PC+4[31:28] . target*4	J	0x2	N/A
Jump register	jr $s	goto address $s	R	0x0	0x8
Jump and link	jal target	$31 ($ra) = PC + 4; PC = PC+4[31:28] . target*4	J	0x3	N/A

- **shamt** means the number to shift (used in shift instructions).

Note that the *Format*, *Opcode* and *Funct* are the information of each field for each format instruction as explained in Section 4.4.

Beside those instructions, some others instructions are sequences of instructions and they are called as pseudo instructions (see in Table 19).

Table 19.: Example of Pseudo Instructions in MIPS

Name	Instruction Syntax	Real instruction translation	Meaning
Move	move $d, $s	add $d, $s, $zero	$d=$s
Load Address	la $d, LabelAddr	lui $d, LabelAddr[31:16] ori $d, $d, LabelAddr[15:0]	$d = Label Address
Multiplies and returns only first 32 bits	mul $d, $s, $t	mult $s, $t mflo $d	$d = $s * $t
Divides and returns quotient	div $d, $s, $t	div $s, $t mflo $d	$d = $s / $t
Branch if equal to zero	beqz $s, Label	beq $s, $zero, Label	if ($s==0) PC=Label

Additionally, MIPS includes a number of system services for input and output interaction, denoted as **SYSCALL**. Let's see an example of those services in Table 20.

To understand better Table 20, we need to give some explanation of it. The *service* column gives us the context of the service; the *code* column explains which value must

Table 20.: Example of SYSCALL instruction in MIPS

Service	Code in $vo	Arguments	Result
print integer	1	$ao = integer to print	
print string	4	$ao = address of null-terminated string to print	
read integer	5		$vo contains integer read
sbrk (allocate heap memory)	9	$ao = number of bytes to allocate	$vo contains address of allocated memory
exit (terminate execution)	10		

be set into register $vo (associated to the service wished); the *arguments* column specify the argument values that must be loaded depending on the service and lastly; the *result* column gives some informations about the return value of the service (if available or not).

Let's see an example of one service in Listing 4.10.

```
1 li $to, 3              #adding the number 3 to register to
2 li  $vo, 1            # loading the service number 1 (print integer) to
    register vo
3 add $ao, $to, $zero  # loading the argument value to register ao
4 syscall  #calling the syscall for printing the integer.
```

Listing 4.10: Example of printing integer in MIPS

Notice that every instructions shown in the tables, are instructions which were used for the project.

4.7 MIPS MEMORY MANAGEMENT

MIPS has the possibility to control and coordinate the computer memory by two ways:

1. stack
2. heap

4.7.1 *MIPS stack*

When a program is being executed, a portion of memory is set aside for the program and it is called the **stack**.

The stack is used for functions and it set some spaces for local variables of the functions.

Internally, MIPS doesn't have real instructions for pushing or popping the stack. But this can be made with a sequences of instructions and using the stack pointer register.

Let's see an example in Listing 4.11.

```
1 push:  addi $sp, $sp, -4  # Decrement stack pointer by 4
2        sw   $vo, 0($sp)   # Save register vo to stack
3
4 pop:   lw   $vo, 0($sp)   # Copy from stack to register vo
5        addi $sp, $sp, 4   # Increment stack pointer by 4
```

Listing 4.11: Example of push and pop instructions in MIPS

4.7.2 *MIPS heap*

Beside a stack, we might need to allocate some dynamic memory. And this can be done by using a **Heap**.

For this purpose, in MIPS, we only need to say how much bytes we want to allocate in the heap.

Let's see an example in Listing 4.12.

```
1 .text
2   main:
3     li $ao, 4 #we want to allocate 4 bytes in the heap.
4     li $vo, 9 # we load the value 9 in register vo for calling the heap
          instruction.
5     syscall   # calling the system call instruction for allocating 4
          bytes into the heap. The register vo contains the address of
          allocated memory.
```

Listing 4.12: Example of code for allocating in the heap

4.8 MIPS SIMULATOR

Several simulators are available in the market for executing MIPS assembly code, and some are free.

For this project, we considered two nice free simulators:

- MARS simulator [3]
- SPIM simulator [4]

3 http://courses.missouristate.edu/KenVollmar/MARS/
4 http://spimsimulator.sourceforge.net

Both simulators are for education purposes and built with a GUI.

They execute and debug MIPS assembly code but only MARS simulator has the possibility to write some live-code MIPS assembly code. This explains why MARS was the one selected for this project.

4.8.1 *MARS at a glance*

MARS from *Mips Assembly and Runtime Simulator*, assembles and simulates the execution of MIPS assembly language programs. The strength of MARS comes from the interaction between the user and the program through its integrated development environment (IDE) and the tools available there (program editing, assembling code, interactive debugging...).

Let's see MARS IDE in Figure 4.

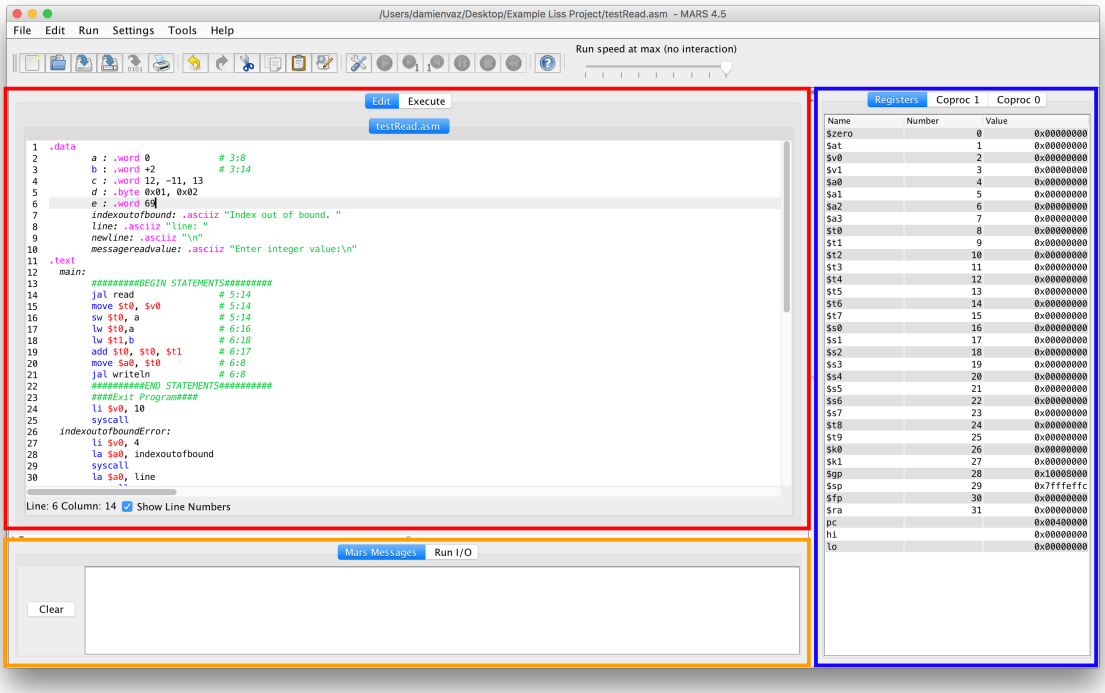


Figure 4.: MARS GUI

In Figure 4, we have 3 different boxes. The red box offers two possible views (two different perspective by switching between the tabs available at the top). In this case, the view is opened for programing some live MIPS assembly code (MIPS assembly code is colored along the left part of the window). But if we open the second tab view, then it will change to the execution mode of the MIPS assembly code (if no syntatic or semantic errors are found).

The orange box also has two possible views (Mars Messages or Run I/O tabs). It is used to display error messages regarding the syntax and semantic of MIPS assembly code, or error messages regarding the execution of the MIPS assembly code.

Lastly, the blue box has three different views: Registers, Co-processor1 and Co-processor 2. In the Figure above, it shows the states of the registers available in MIPS architecture but if we change the view it can show the states of each co-processor (related to division, multiplication).

If the MIPS assembly code typed in (or loaded from a file) is correct (no errors detected), we can assemble it and execute it.

Figure 5 illustrates the new view offered by the IDE after assembling the source program.

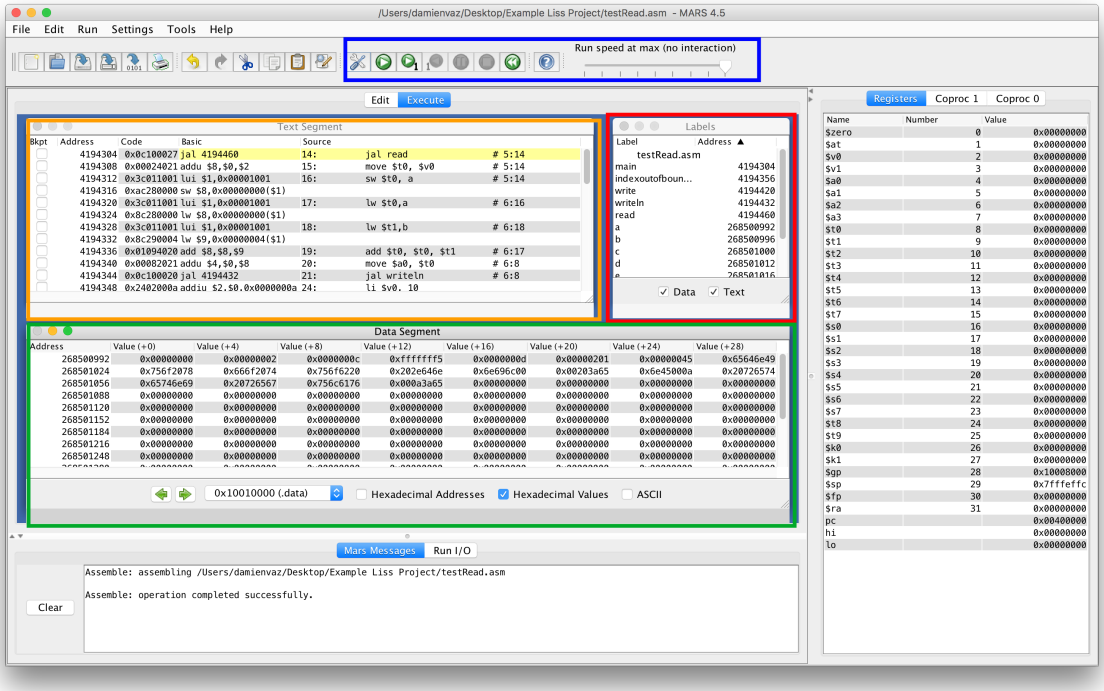


Figure 5.: MARS GUI (Execution mode)

In Figure 5, it is possible to identify the main window (the red one in Figure 4) now split into three subwindows: orange, red and green.

Notice that above the main red window, a small blue box contains buttons to activate tools for assembling MIPS assembly code, executing MIPS assembly code totally or step by step (one instruction at a time) and also the possibility to change the speed execution of the MIPS assembly code if we want to run it completely.

The orange box contains the MIPS assembly code assembled and ready to execute. It shows the MIPS assembly code instructions, the correspondent code in hexadecimal, the respective address in the memory, and eventually some breakpoints associated with cer-

tain MIPS assembly instructions. Also notice that MIPS assembly code has some pseudo-instructions; and in the orange box, there is a part where we can see the translation of the MIPS assembly code to another lower MIPS assembly code (with no pseudo-instruction). The yellow bar, or cursor, displayed in the figure above enhances the next instruction to be executed.

The red box is the identifier table for the MIPS assembly code. It contains the variables existing in the MIPS assembly code and displays their respective address in the memory.

The green box represents the virtual memory of the MIPS architecture. It displays the stack and the heap memory, as well as other informations not relevant in this context. Basically, we see the value being changed throw the iteration of the MIPS assembly code being executed. This mean that if there is a store instruction for a certain variable, it will look up for the identifer table (red box), search the address associated to the variable and store to that address the value associated to the variable.

5

COMPILER DEVELOPMENT

Earlier in the history of computers, software was primarily written in assembly language. Due to the low productivity of programming assembly code, researchers invented a way that add some more productivity and flexibility for programmers; they created the compiler allowing to wire programs in high level programming languages.

A compiler is a software program which converts a high-level programming language (source code) into a lower level programing language for the target machine (known as machine code or assembly language).

The compiler task is divided into several steps (see Figure 6):

1. Lexical analysis
2. Syntactic analysis or parsing
3. Semantic analysis
4. Optimization
5. Code generation

Firstly, the lexical analysis must recognize words; these words are a string of symbols each of which is a letter, a digit or a special character. The Lexical analysis divides program text into ”words” or ”tokens” and once words are identified, the next step is to understand sentence structure (role of the parser). We can think the parsing as an analogy of our world by constructing phrases which requires a subject, verb and object. So, basically, the parser do a diagramming of sentences.

Once the sentence structure is understood, we must extract the ”meaning” with the semantic analyzer. The duty of the semantic analyzer is to perform some contextual checks to catch language inconsistencies and build an intermediate representation to store the meaning of the source text. After that, it may or may not have some optimization regarding the source code.

Finally, the code generator translates the intermediate representation of the high-level programming into assembly code (lower level programming). At this stage, a new opti-

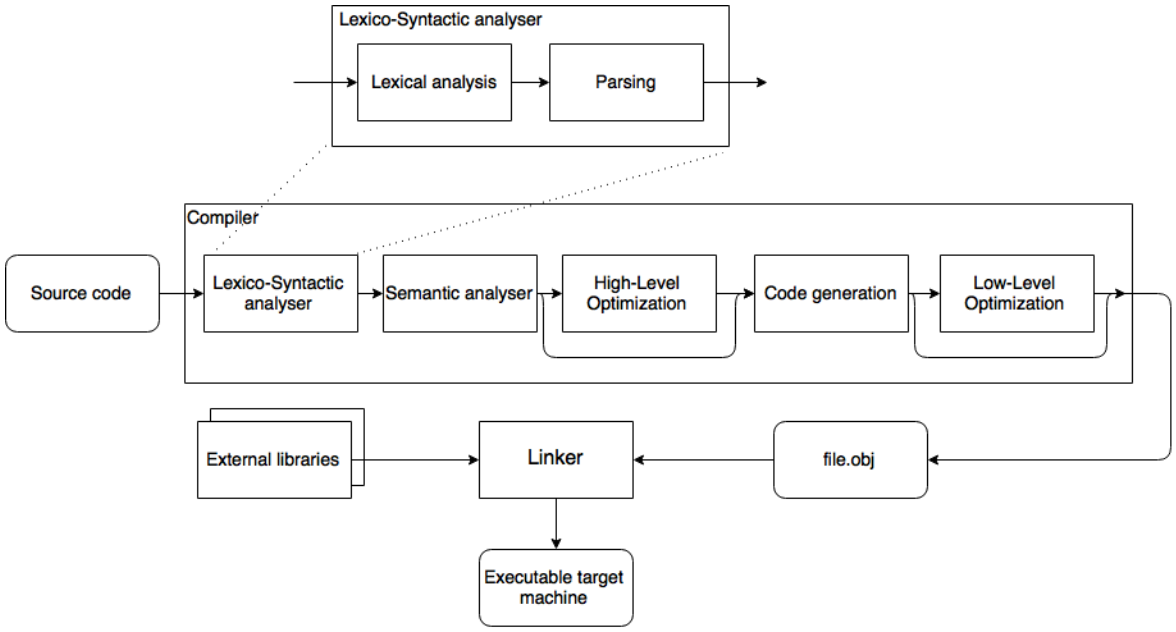


Figure 6.: Traditional compiler

mization phase can occur to deliver an object code shorter and faster than the original one.

Notice that the task of constructing a compiler for a particular source language is complex. To simplify this task, it is usual to resort to a compiler generator that is a system able to build automatically a language processor from the language grammar. In this master project, the compiler generator ANTLR was used, as we will be described in section 5.1.

The first tool steps, lexical and syntatical analysis, will be briefly discussed in section 5.2. Then section 5.3 explains in detail the implementation of LISS semantic analyzer. To conclude the chapter, section 5.4 provides also details about the implementation of the LISS code generator.

5.1 COMPILER GENERATION WITH ANTLR

Terence Parr, the man who is behind ANTLR (ANother Tool for Language Recognition (Parr, 2007, 2005)) made a parser (or more precisely, a compiler) generator that reads a context free grammar, a translation grammar, or an attribute grammar and produces automatically a processor (based on a LL(k) recursive-descent parser) for the language defined by the input grammar.

An ANTLR specification is composed by two parts : the one with all the grammar rules and the other one with lexer grammar.

Listing 5.1 is the one with the grammar rules; in that case it is an example of an AG (Attribute Grammar).

```
facturas : fatura +
         ;
fatura   : 'FATURA' cabec 'VENDAS' corpo
         ;
cabec    : numFat idForn 'CLIENTE' idClie
         { System.out.println("FATURA num: " + $numFat.text);}
         ;
numFat   : ID
         ;
idForn   : nome morada 'NIF:' nif  'NIB:' nib
```

Listing 5.1: AG representation on ANTLR

On the other hand, the lexer grammar defines the lexical rules which are regular expressions as can be seen in Listing 5.2. They define the set of possible character sequences that are used to form individual tokens. A lexer recognizes strings and for each string found, it produces the respective tokens.

```
/*————————————— Lexer ——————————————————————————————*/

ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'-')*
    ;

NUM :   '0'..'9'+
```

Listing 5.2: Lexer representation

5.2 Lexical and syntatical analysis

The parser generator by ANTLR will be able to create an abstract syntax tree (AST) which is a tree representation of the abstract syntactic structure of source code written in a programming language (see Figure 7).

ANTLR will be used to generate MIPS assembly code according to the semantic rule specified in the AG for LISS language.

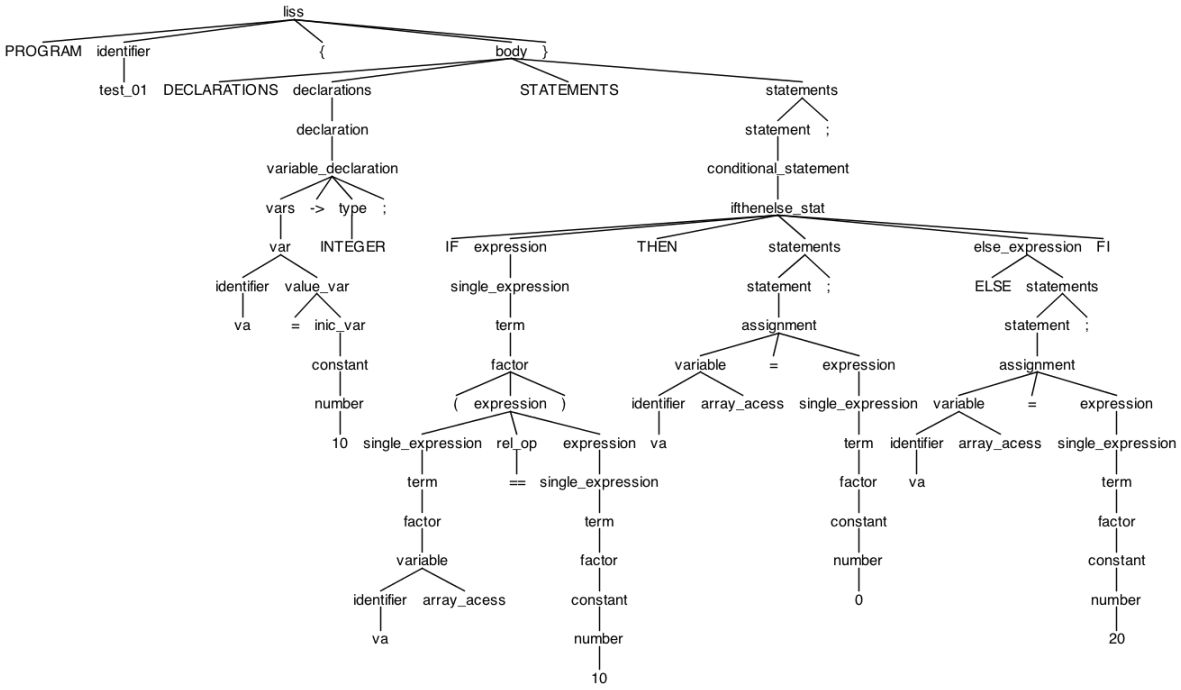


Figure 7.: AST representation

5.3 SEMANTIC ANALYSIS

5.3.1 *Symbol Table*

A symbol table is a data structure used for the compiler, which helps to store some valuable informations for identifiers in a program's source code. Basically, it helps the compiler for finding some semantic errors regarding to the translation of the program which will be done later.

There are a lot of types of data structure for creating a symbol table. From one large symbol table for all symbols or separated, hierarchical symbol tables for different scopes.

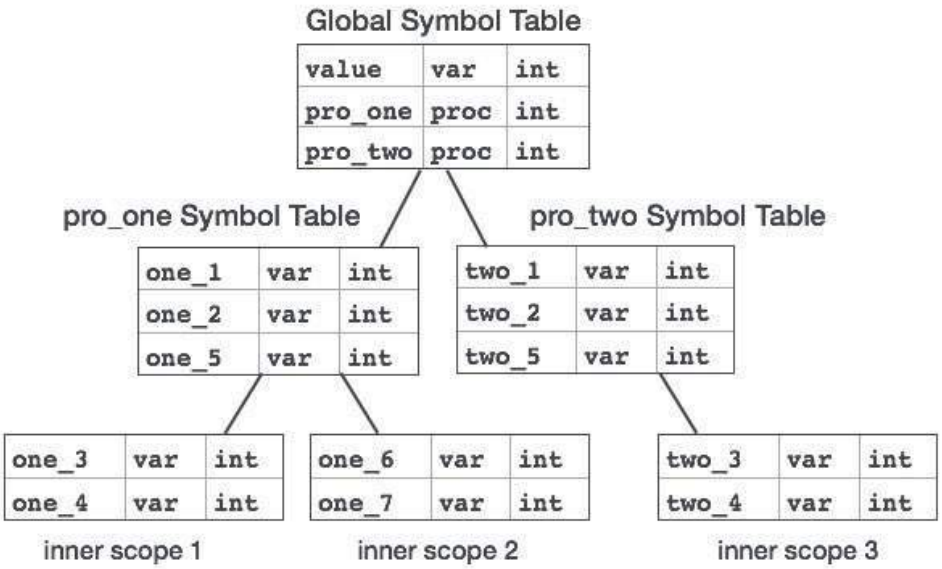


Figure 8.: Example of hierarchical symbol table

Symbol Table in LISS

For this project, we used a one large symbol table for all symbols.

Let's explain throw the Figure 9 how it works the symbol table in LISS.

For our project we implemented the symbol table with a HashMap where the key is an identifier and the value is a LinkedList of some informations related to the identifier.

```
1 HashMap<String, LinkedList<InfoIdentifiersTable>>
```

Listing 5.3: Data structure of the symbol table in LISS

The identifier (related to *String* word in 5.3) must be unique (concept of using a HashMap) and the *LinkedList* must have the idea of an ordered list.

Basically, the identifier is associated to a *LinkedList* of informations related to the identifier and those informations explain which category and type is the identifier.

For our project, we have 3 different categories:

1. TYPE
2. VAR
3. FUNCTION

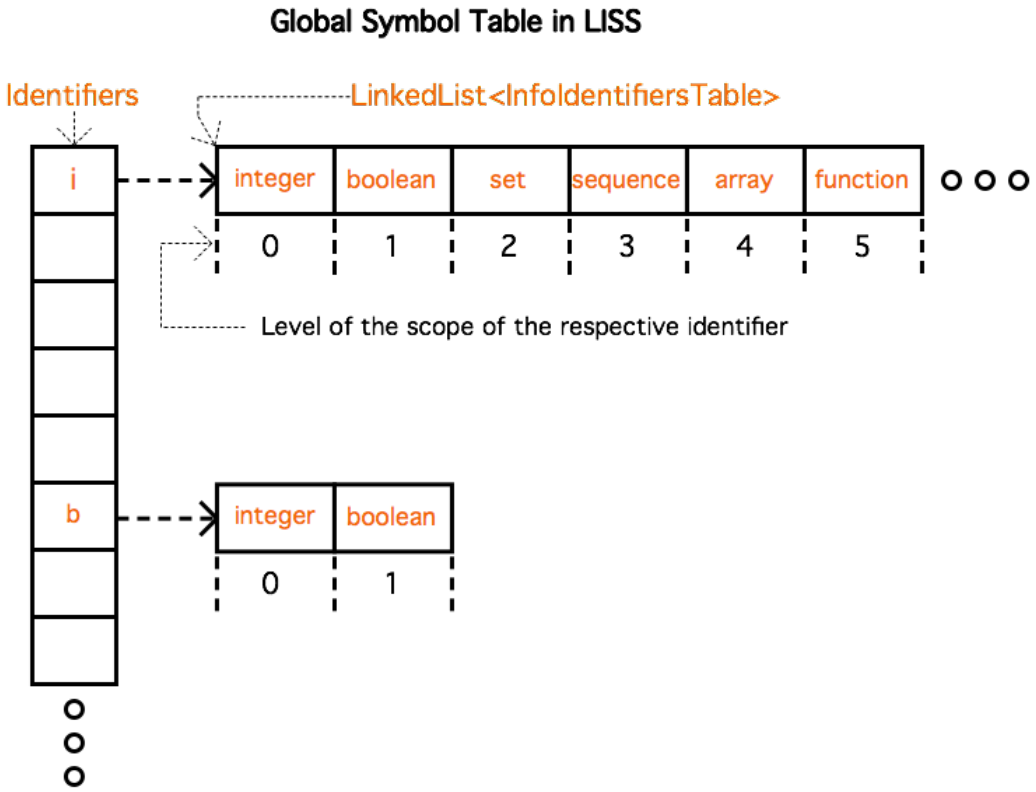


Figure 9.: Global symbol table in LISS

TYPE category

The *TYPE* category contains informations about which primative type is available in LISS. In our cases, they are known by : set, integer, sequence and boolean. It contains information about their fixed size of each type in MIPS as well as their level scope.

Table 21.: TYPE category information

Identifier	Category	Level	Space (Bytes)
set	TYPE	0	0
integer	TYPE	0	4
boolean	TYPE	0	4
sequence	TYPE	0	4

VAR category

The *VAR* category contains informations about the variables declared in LISS code (under *declarations* part). It might be an integer, boolean, array, set or a sequence variable. And each variable have different respective informations for their type associated.

Let's see and explain each type.

Table 22.: Information related for an integer variable

Identifier	Category	Level	Type	Address
int	VAR	0	integer	0

In Table 22, we can see the type of information that an integer variable stores into the symbol table:

- *Identifier* - name of the variable.
- *Category* - the category of the identifier: variable.
- *Level* - the level scope of the variable.
- *Type* - the type of the variable (integer).
- *Address* - the address of the variable in the stack memory.

Table 23.: Information related for a boolean variable

Identifier	Category	Level	Type	Address
bool	VAR	1	boolean	4

In Table 23, this is the type of information that a boolean variable stores in the symbol table.

- *Identifier* - name of the variable.
- *Category* - the category of the identifier: variable.
- *Level* - the level scope of the variable.
- *Type* - the type of the variable (boolean).
- *Address* - the address of the variable in the stack memory.

Table 24.: Information related to an array variable

Identifier	Category	Level	Type	Address	Dimension	Limits
array_1	VAR	0	array	8	2	[2|3]

In Table 24, this is the type of information related to an array.

- *Identifier* - name of the variable.
- *Category* - the category of the identifier: variable.
- *Level* - the level scope of the variable.
- *Type* - the type of the variable (array).
- *Address* - the address of the variable in the stack memory.
- *Dimension* - the number of dimension for the array.
- *Limits* - the limits of each dimension of the array.

Table 25.: Information related to a set variable

Identifier	Category	Level	Type	Address	Tree Allocated
set_1	VAR	0	set	NULL	[x]

In Table 25, this is the type of information for a set.

- *Identifier* - name of the variable.

- *Category* - the category of the identifier: variable.
- *Level* - the level scope of the variable.
- *Type* - the type of the variable (set).
- *Address* - address in the stack memory, but sets doesn't need to. It creates the MIPS assembly code.
- *Tree Allocated* - indicates if the sets was initiated or not. If an 'X' letter appears, it means that the sets was initiated.

Table 26.: Information related to a sequence variable

Identifier	Category	Level	Type	Address	Elements_type
sequence_1	VAR	0	sequence	32	integer

In Table 26, this is the type of information for a set.

- *Identifier* - name of the variable.
- *Category* - the category of the identifier: variable.
- *Level* - the level scope of the variable.
- *Type* - the type of the variable (sequence).
- *Address* - address in the stack memory.
- *Elements_type* - indicates the type of the elements.

FUNCTION category

Lastly, the *FUNCTION* category contains informations about the subprograms created in a LISS code.

Table 27.: Information related to a function

Identifier	Category	Level	Type	Address	Nº Arguments	Type List Arguments
calculate	FUNCTION	0	NULL	32	2	[integer, boolean]

In Table 27, this is the type of information for a function.

- *Identifier* - name of the variable.
- *Category* - the category of the identifier: function
- *Level* - the level scope of the function.

- *Type* - the type of the function which is NULL.
- *Address* - size of the function stack in the stack memory (contains the list arguments, the variables declared in the subprogram and the return address of the function).
- *Nº Arguments* - indicates how many arguments the function does have.
- *Type List Arguments* - indicates the types of each arguments of the function.

Let's see the abstract data structure of InfoIdentifiersTable implemented in Figure 10.

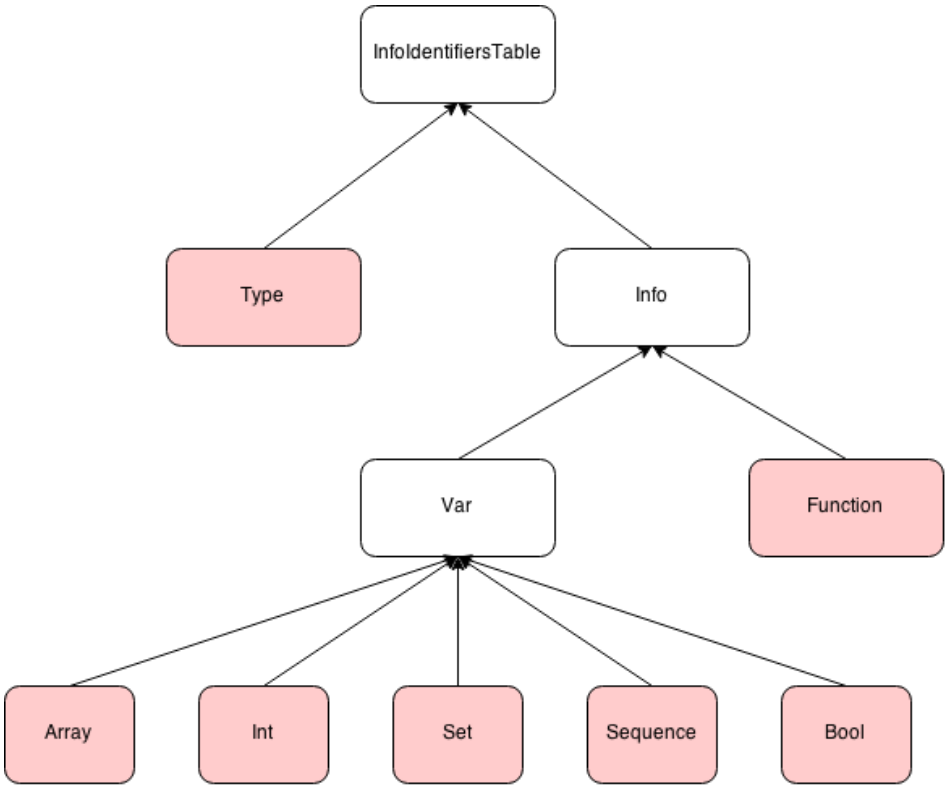


Figure 10.: InfoIdentifiersTable structure

Each time that an identifier is inserted into the HashMap, it inserts the information related to that identifier and in this case, the type of the identifier. For example, in Figure 10 the info that will be stored in the symbol table are the red boxes.

Care that *LinkedList* has the notion of being an ordered list, and this is a very important idea for the symbol table due to the fact that it reveals the level of the scope for the given identifier found.

In Figure 9, the identifier **b** was found in two differents level scope.

- Scope level 0 - Identifier **b** found with type *integer*
- Scope level 1 - Identifier **b** found with type *boolean*

Notice that every time, we look for an identifier and his respective info in the symbol table. It will always take the latest info in the *LinkedList* data structure.

In the case of the identifier example **b** in Figure 9, it will be the **boolean** info.

Also, every time that a function (*subprogram* in LISS) is exited, we remove every information available to the respective level scope of the function in the symbol table. And this is due for having a good consistency regarding to the information available in the symbol table.

Let's see the functions created and available in the project, regarding to the symbol table structure in JAVA.

- getSymbolTable - gets the symbol table.
- doesExist - checks if a certain identifier is available.
- getInfoIdentifier - gets the latest information of a certain identifier.
- removeLevel - removes every information according to the level scope of every identifier available in the symbol table.
- getAddress - gets the latest address (this address is related to the next position of an identifier that will be added in the symbol table).
- setAddress - sets a new address.
- add - adds identifiers into the symbol table.
- toString - gets the representation of the symbol table as a string.

5.3.2 *Semantic system*

In programming language theory, the word *semantics* is concerned by the field of studying the meaning of programming languages. And in this field, it concerns about a lot of area. For our project, every time that we see an inconsistency, we report them to an error table. Let's see which kind of inconsistency we can find for our project.

1. Finding inconsistency in types and their related specifications.
2. Finding inconsistency in variables declared or not.
3. Finding inconsistency regarding to the use of multiple expressions.
4. Finding inconsistency for returning types of functions created.

This will be talked later, let's understand firstly the error table system created.

5.3.3 *Error table in LISS*

The error table let the user to understand the problems that he is having with the code when he is trying to create or making it. In this way, it will facilitate the user to correct the problems found regarding to his code with ease.

Let's see the table error structure made for our project in Figure 11.

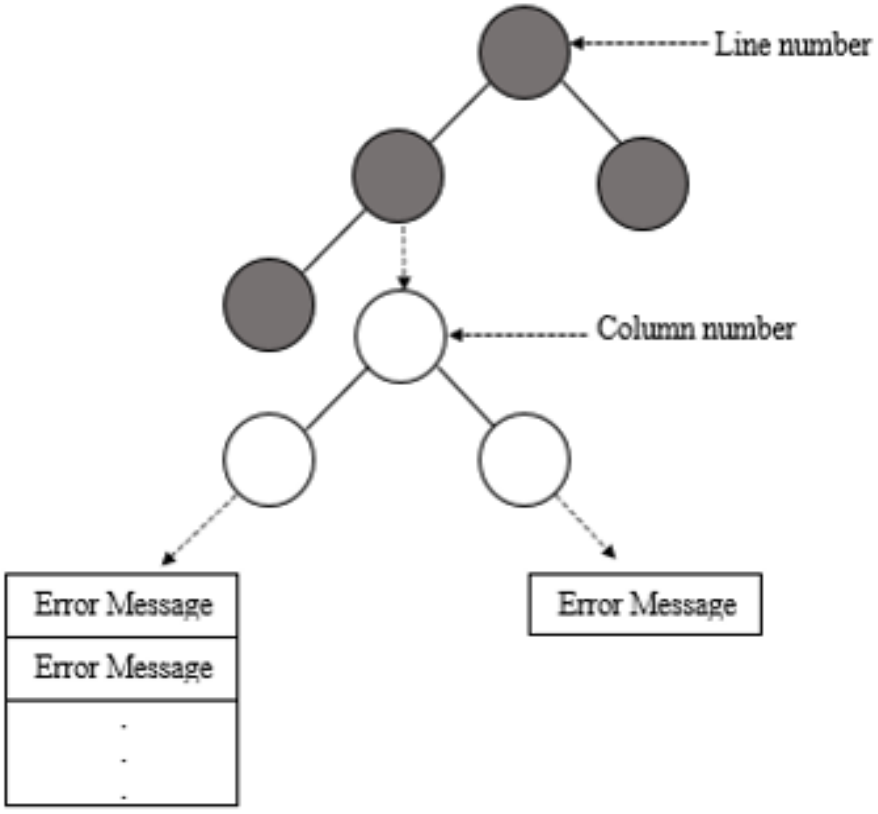


Figure 11.: ErrorTable structure

For our project, we managed to create a data structure which could handle some error messages and could give us also some informations related to the error message (line and column number).

And this was done by creating that data structure in JAVA:

```
1 TreeMap<Integer ,TreeMap<Integer ,ArrayList<String>>>
```

Listing 5.4: Data structure of the error table in LISS

Basically, this data structure is divided into two *TreeMap* (those *TreeMap* can be seen in Figure 11, black and white) and a list (*ArrayList* data structure) of some error messages.

We choosed the *TreeMap* data structure for one reason, the map is sorted according to the natural ordering of its keys. This means that each time we insert in the *TreeMap* data structure, the information is ordered by his key beside that also the key will be unique. In this case, the first *TreeMap* is concerned for ordering the line number of the error message (black tree in Figure 11).

Then when the line number is added and ordered, we add some information linked to the line number and this is the column number of the line (white tree in Figure 11).

Finally, we add the error messages to the list related to a certain line and column.

With that data structure, we are sure that it can have a list of error messages for a certain line and column numbers and that the line and the column number are ordered for an ease interpretation for the user in solving the problem regarding to his code.

Let's see an example of the error table in Figure 5.5.

```
1 ERROR TABLE:
2   line: 5:18  Expression 'b'  has type 'boolean',when It should be '
      integer'.
3   line: 6:11  Expression 'flag'  has type 'integer',when It should be '
      boolean'.
4   line: 7:1 Expression 'array1=[[1,2],[2,3,4,5]],vector' has a problem
      with his limits.
5   line: 8:1 Expression 'vector' already exists.
6   line: 10:4  Expression 'seq1' already exists.
7   line: 14:4  Expression 'b' already exists.
```

Listing 5.5: Example of an error table

5.3.4 *Types of error message*

For our error table structure, we needed to add some informations relatively to the error message thrown. And as we said, previously, we have different kind of types for some error messages.

Let's see in the Table 28.

Table 28.: Types of error message in LISS

Error type number	Error message
1	Variable <name_of_variable >isn't declared
2	Variable <name_of_the_variable >already exists.
3	Variable <name_of_the_array_variable >must be an 'array'.
4	Variable <name_of_the_array_variable >has a problem with his limits.
5	Variable <name_of_the_variable >has type <type_found >, when it should be <type_expected >.
6	Incompatible types in Assignment.
7	Expression <expression_string >has type <type_found >, when it should be <type_expected >.
8	Function <name_of_the_function >has return type <type_found >, when it should be <type_expected >.
9	Variable <name_of_the_function >is not a function.
10	Expression <expression_string >has type <left_type_found > <operator_string ><right_type_found >, required type <left_type_required ><operator_string ><right_type_required >.
11	Expression <name_of_the_array_variable >has dimension <dimension_found >, when it should be equal to <dimension_required >.
12	'stepUp' or 'stepDown' expression, not valid with "inArray" operation.
13	'satisfying' expression, not valid with "inArray" operation.
14	Function <name_of_the_function >does not exist.
15	Expression <name_of_the_array_variable >doesn't have the same limits or dimensions.

Notice that in Table 28, we have every messages used and thrown (when necessary) for the compiler and some of them have a certain semantic structure which we need to explain. For example, every marks:

```
1  <...>
```

found in the table message, means that it must be replaced by the correct name according to the environment that the error was found.

Let's see an example in Listing 5.6 to understand better the usage of the mark.

```
1 program Errors{
2   declarations
3     seq1 =<<1,2,3,4>> -> sequence;
4     seq1 = <<1,4,7>> -> sequence;
5   statements
6 }
```

Listing 5.6: Partial Listing

In Listing 5.6, we have a LISS program which has two variables with the same name in the declarations part. As all programming language have, this behaviour of declaring two variables in the same level scope is prohibited, so the compiler must throw an error.

Let's see the error table in Listing 5.7.

```
1 ERROR TABLE:
2 line: 4:2 Variable 'seq1' already exists.
```

Listing 5.7: Error table related to Listing 5.6

The error table in 5.7, has an error message with the number 2 message type (see Table 28). And we can see that the mark was replaced by the name of the variable (as said in the Table 28 regarding to the message number 2 type). And this is the same behavior regarding to the others messages available in the Table 28.

Now, let's explain the Table 28 for an ease interpretation of those messages.

1. Message for variable that are not declared.
2. Message for variables that already exists.
3. Message for variables that must be an array and they are not.
4. Message for variables that are an array type and their limits doesn't match.
5. Message for variables that have a certain type, but they should have another type.
6. Message regarding to different types when an assignment is found. For example : 'integer' = 'boolean'.
7. Message for expressions that has different types.
8. Message for functions regarding to the return type which might be different.
9. Message for functions which the name of the function doesn't have the type function and does have another type.
10. Message for expressions which has different types according to the operator who is being used. For example: 'integer' + 'boolean'.
11. Message for arrays that has different dimension, according to his declaration.
12. Message for unconditional loop that use 'stepUp' or 'stepDown' expression.

13. Message for unconditional loop that use 'satisfying' expression.
14. Message for functions where the name of the function doesn't exist.
15. Message for arrays where the limits or the dimension isn't equal as the declared array variable.

5.3.5 *Usage of error messages*

In this section, we are going to report where the error message will be thrown in the compiler regarding to the attribute grammar that we have.

Variable declaration

```
variable_declaration : vars '->' type ';'
```

Listing 5.8: Variable declaration rule in LISS

In Listing 5.8, this is the part where we declare some variables with the types chosen under the declarations part. And this also where we add the information into the symbol table too. So, before we add the information into the symbol table, we need to check if every variables does have the correct type regarding to the type chosen. If not, then it will throw the number 5 error message (see Table 28).

Let's see an error that might happen for this case in Listing 5.9.

```
b = boolean -> integer;
```

Listing 5.9: Example of an error message in variable declaration

Notice that in this section, we create the mips code for every variables. And there is one type which can throw an error message too in this section, it is called the *array* type. For this type, we need to check if the limits are correct, if they are not then it throw the number 4 error message (see Table 28).

Regarding to the others type, we don't check in this section. Because the *array* type, is the only one who is the hardest one to deal (needs to calculate the position of the array) for creating the mips code instruction.

Let's see an example of an array type error message regarding to this case in Listing 5.10.

```
array1 = [[1,2],[2,3,4,5]], vector -> array size 4,3;
```

Listing 5.10: Example of an error message in variable declaration for the array type

Vars

```
vars : v1 (',' v2)*
```

Listing 5.11: Vars rule in LISS

In Listing 5.11, this grammar part refers to the declaration of multiple variables in a same line under the declarations section in LISS.

And for this part we need to check if the variables that will be added to the symbol table, they are already created or not. If they are already, then the number 2 error message will be thrown (see Table 28). Let's see an example of LISS that can throw this error in the error table in Listing 5.12.

```
a = 4, a = 5 -> integer;
```

Listing 5.12: Example of an error message in LISS for vars non-terminal

In Listing 5.12, there is a problem regarding that two variables with the same name are being created. This must throw an error as we said previously, and an error message related to the second variable.

Set initialization

```
set_initialization :
                   | identifier '|' expression
```

Listing 5.13: Set initialization rule in LISS

In Listing 5.13, this part refers for declaring a set under the declarations section in LISS. And it has two choices for declaring a set: empty set or some content for the set. If there is some content available, this content must be a boolean expression. In case the expression isn't a boolean, it will throw an error message to the error table and this will be the number 7 error message (see Table 28). Let's see an example of this error with a piece of LISS code related to set initialization in Listing 5.14.

```
set6 = { z | (z+tail(z)) < 5} -> set;
```

Listing 5.14: Example of an error message in LISS for set_initialization non-terminal

In Listing 5.14, we can see that the variable won't be declared due to the fact that the content isn't correct. The function *tail* is a function for sequence and it needs a sequence variable as an argument, not an integer variable (as we can see). The compiler will return the type of that operation as 'null' because he can't execute that operation.

Then, there is an sum operator who needs a variable of type integer. But due to the previous statements we have made, the type that the sum operator will have, are: integer (from *z* variable) and a null (from *tail(z)*). The compiler can't execute it too, so the error is being spread throw the entire operation of the set content.

In the end, after calculating the content of the set initialization of the set, the error table will have that comment regarding to that variable in Listing 5.15.

```
1 line: 20:18 Expression '(z+tail(z))<5' has type 'null',when It should
    be 'boolean'.
```

Listing 5.15: Error message for the set_initialization

Subprogram definition

```
1 subprogram_definition : 'subprogram' identifier '(' formal_args ')'
    return_type f_body
```

Listing 5.16: Subprogram definition rule in LISS

In Listing 5.16, this part refers for declaring any subprogram (functions) under the declaration section in LISS. And for this part we need to check the return type of the subprogram.

If the return type has a different type chosen and type needed (for example, type chosen is boolean, but the return type is an integer) then the number 7 error message (see Table 28) is thrown.

Let's see an example of this case in Listing 5.17.

```
1 subprogram f(amen->boolean)->integer{
2       declarations
3             b -> boolean;
4       statements
5       return b;
6 }
```

Listing 5.17: Example of error message in LISS for subprogram_definition non_terminal

In Listing 5.17, we can see that the return type permitted is an integer. But with this example, the subprogram returns a variable called 'b' and has a boolean type. In this case, an error message will be reported due to the incorrectness type.

Assignment

```
assignment : designator '=' expression
```

Listing 5.18: Assignment rule in LISS

In Listing 5.18, this part refers for assigning some content to a variable or an array under the statement section in LISS. And for this part we need to check some error messages regarding to the context available. Let's explain those differents context below.

If the designator non-terminal in Listing 5.18, is a variable of type *array*. It means that we can store some content for that variable (see example of this case in Listing 5.19).

```
array1 = [1,2,3];
```

Listing 5.19: Example of storing some values to an array variable

In Listing 5.19, we see a variable named *array1* with the type *array* and it will be assigned to some values shown in the example, *[1,2,3]*. For this example, we need to check if the values have the correct dimension and limits regarding the variable. In this case, if it isn't correct then the number 15 error message (see Table 28).

Also the same error message is reported for the case of the next example in Listing 5.20.

```
array[3] = 1;
```

Listing 5.20: Example of storing a value to a certain position in the array

In Listing 5.20, if the variable *array* has only two position available. This means that the access of the fourth position (as shown in the example), it is behind of the limits regarding to the specification of the variable. And, in this case, it must throw the number 15 error message too (see Table 28).

Last case for this part concerns about the types of *designator* non-terminal and *expression* non-terminal not being equals. In this case, the compiler can't execute and procede with the assignment operation.

This will throw the number 6 error message to the error table (see Table 28).

Let's see an example of this case in Listing 5.21.

```
boolean1 = integer1;
```

Listing 5.21: Example of assignment with differents types

In Listing 5.21, the example informs us that assignment operation is trying to store an integer to the *boolean1* variable (which has the boolean type). As we know, if the types aren't equals then the operations cannot be executed and must report an error.

Designator

```
1  designator : identifier array_access
```

Listing 5.22: Designator rule in LISS

In Listing 5.22, this part refers for using a variable or an array variable under the statement section in LISS. And we need to check some errors when it used only for a variable context or in the array variable context.

Let's speak firstly for the variable context.

If it is only the variable context, the *array_access* is not available but, only, the *identifier* non-terminal will be available.

Every variable used must be declared in the symbol table. If it doesn't exist in the symbol table, it means that variable doesn't exist and we need to throw an error in the error table, number 1 error message (see Table 28).

We need to check if the name of the *identifier* isn't the same as the name of a type in LISS (see Table 21). If it is, then it must throw the number 1 error message to the error table (see Table 28).

Now regarding to the array variable context, we need to check a lot of errors.

Firstly, we need to check if the identifier is available in the symbol table otherwise it will throw the number 1 error message to the error table (see Table 28).

Secondly, we need to check if the name of the identifier isn't the same name as the name of a type in LISS (see Table 21). If it is, then it must throw the number 1 error message to the error table (see Table 28).

Thirdly, we need to check the type of identifier. If the type isn't an array then we need to throw the number 3 error message (see Table 28).

And finally, we need to check if the *identifier* and the *array_access* non-terminal have the same dimension. If they don't have then it must be throw the number 11 error message to the error table (see Table 28).

Elem array

```
1  elem_array : s1=single_expression ( ',' s2=single_expression )*
```

Listing 5.23: Elem_array rule in LISS

In Listing 5.23, this part refers to the elements of an array. And for this part we need to check if every elements (represented by the *single_expression* non-terminal) has the correct type.

In an array context, the type of each elements must be an integer. If it isn't then it must throw the number 7 error message to the error table (see Table 28).

Function call

```
function_call : i=identifier '(' sub_prg_args ')'
```

Listing 5.24: Function_call rule in LISS

In Listing 5.24, this part is related to call some functions under the statement section in LISS. For this context, we need to check two things regarding to errors.

Firstly, we need to check if the *identifier* is available in the symbol table. If it isn't then it must throw the number 14 error message to the error table (see Table 28).

Lastly, we need to check if the *identifier* has the correct type (*Function* type). If it doesn't have then it must throw the number 9 error message to the error table (see Table 28).

Expression

```
expression : single_expression (rel_op single_expression )?
```

Listing 5.25: Expression rule in LISS

In Listing 5.25, this part refers to expression that can be used in a various way in LISS. For this case, we need to check if the type of both *single_expression* non-terminal are correct in relation to the type required for the *rel_op* non-terminal. If they are not, then we throw the number 10 error message to the error table (see Table 28).

Notice that it is the *rel_op* non-terminal who tells which type that the left and right *single_expression* non-terminal must have.

Let's see an example in Listing 5.26.

```
2 < true
```

Listing 5.26: Example of an error message in expression rule

In Listing 5.26, we can see the number two (first single_expression annotation) then the minus sign (the rel_op annotation) and finaly the true value (the last single_expression annotation). If we procede to calculate the entire expression, we can see that the types doesn't match at all between them. In that case the minus sign needs an integer in both side (left and right), and actually it does have an integer and a boolean expression which is an error to be thrown in the error table.

Single expression

```
single_expression : term ( add_op term)*
```

Listing 5.27: Single_expresion rule in LISS

In Listing 5.27, this part refers to the expansion of the *single_expression* non-terminal rule of the *expression* rule in LISS. And the behaviour is the same as the *expression* rule, this means that we need to check the types required for the *add_op* non-terminal. If *single_expression* non-terminal type don't coincide with the required type regarding to the *add_op* non-terminal, then it must throw the number 10 error message to the error table.

Term

```
1 term : factor ( mul_op factor)*
```

Listing 5.28: Term rule in LISS

In Listing 5.28, this part refers to the expansion of the *term* non-terminal rule of the *single_expression* rule in LISS. And the behaviour is the same as the *expression* rule, this means that we need to check the types required for the *mul_op* non-terminal. If *factor* non-terminal type don't coincide with the required type regarding to the *mul_op* non-terminal, then it must throw the number 10 error message to the error table.

Factor

```
1 factor : inic_var
2          | designator
3          | '(' expression ')'
4          | '!' factor
5          | function_call
6          | specialFunctions
```

Listing 5.29: Factor rule in LISS

In Listing 5.29, this part refers to the expansion of the *factor* non-terminal rule of the *term* rule in LISS. And as it can be seen, there is a lot of options for this rule.

But we will do a particular attention to a simple one choice of that rule:

```
1 '!' factor
```

Notice that there is an exclamation mark sign and then a *factor* non-terminal. In programming langues, the exclamation mark means that it negate the expression. And this

type of negation requires a boolean type in order to work correctly. So, if the type of the *factor* non-terminal is not a boolean. Then the number 7 error message will be added to the error table.

Print_what

```
print_what :
           | expression
           | string
```

Listing 5.30: Print_what rule in LISS

In Listing 5.30, this part refers for printing in the output with the use of LISS language.

And we need to check the type of the *expression* non-terminal. If the type is a *set* then it must throw the number 7 error message in the error table (see Table 28).

Notice that the type allowed for the *expression* non-terminal are :

- integer
- boolean
- sequence
- array

Read

```
read_statement : 'input' '(' identifier ')'
```

Listing 5.31: Read rule in LISS

In Listing 5.31, this part refers for reading the input of the user which will be stored to the *identifier* non-terminal in LISS. And we need to check some errors for that non-terminal.

If the *identifier* non-terminal doesn't exist in the symbol table, then we must thrown the number 1 error message (see Table 28). If the *identifier* exists in the symbol table, we must check the type of it. If the type isn't an integer, then we must throw the number 5 error message in the error table (see Table 28).

If_then_else_stat

```
if_then_else_stat : 'if' '(' expression ')'
                  'then' '{' statements '}'
                   else_expression
```

Listing 5.32: If_then_else_stat rule in LISS

In Listing 5.32, this part refers in the use of a conditional expression and particularly, an 'if' statement.

As all programming languages, the behaviour of an 'if' statement is the same. And we do also the same in LISS, it means that the *expression* non-terminal must be a boolean type. Because as it is a condition with an 'if' statement it will say if it will enter to that branch or to another branch regarding to the value of the condition. If the *expression* non-terminal type isn't a boolean, then it must throw the number 7 error message to the error table (see Table 28).

For_stat

```
for_stat : 'for' '(' interval ')' step satisfy
           {
             statements
           }
```

Listing 5.33: For_stat rule in LISS

In Listing 5.33, this part refers to the use of a 'for-loop' statements in LISS.

For this particular case, we test an error regarding to the use of a 'for-each' context or not.

Remember that in LISS context, we are able to use a 'for-loop' which can access to every elements of an array, also called 'for-each'. And with that 'for-each' context, we cannot use *step* non-terminal and *satisfy* non-terminal. So we must check if we use a 'for-each' loop, if it is used then we need to check if the *step* or *satisfy* non-terminal are available.

If the *step* non-terminal is available then we must throw the number 12 error message to the error table (see Table 28).

If the *satisfy* non-terminal is available then we must throw the number 13 error message to the error table (see Table 28).

Let's see an example of those cases in Listing 5.34.

```
for(b inArray vector) stepDown 1 satisfying vector[0] == a
```

Listing 5.34: Example of an error message in for_stat rule

In Listing 5.34, the fact that there is an *inArray* word means that the statement is a 'for-each' loop. And for this case we cannot use a *step* or a *satisfying* rule. But, for this case, there is.

For the compiler, it means that it must throw both the number 12 and 13 error message to the error table due to the incorectness of the language LISS.

Interval

```
interval : identifier type_interval
```

Listing 5.35: Interval rule in LISS

In Listing 5.35, this part refers to the expansion of the *interval* non-terminal rule of the *for_stat* rule in LISS.

We need to check if the *identifier* non-terminal is available to the symbol table, if it isn't then it must throw the number 1 error message to the error table (see Table 28).

If the *identifier* is on the symbol table, we need to check the type of the variable.

As we said before, the 'for-each' statement refers to the use of *array*. And the type *array*, uses only *integer* elements. In this case, the variable *identifier* must also have the same type, *integer*.

So we need to check the type of the variable *identifier* and see if it is an *integer*. If it isn't, then it must throw the number 5 error message to the error table (see Table 28).

Type_interval

```
type_interval : 'in' range
              | 'inArray' identifier
```

Listing 5.36: Type_interval rule in LISS

In Listing 5.36, this part refers to the expansion of the *type_interval* non-terminal rule of the *interval* rule in LISS. It tells us which kind of operation can be a 'for-loop' in LISS. And as we can see there is two choices, the normal behaviour of the 'for-loop' statement (represented by *in range*) and the 'for-each' statement (represented by *inArray identifier*). For this case, we make only an attention to the 'for-each' statement, and particularly to the *identifier* terminal.

We need to check if the *identifier* variable is on the symbol table firstly, if it isn't the number 1 error message will be throw in the error table (see Table 28). Then we need to check the type that *identifier* variable has. If the variable isn't an *array* then it must throw the number 5 error message to the error table (see Table 28).

Minimum

```
minimum : number
        | identifier
```

Listing 5.37: Minimum rule in LISS

In Listing 5.37, this part refers to the expansion of the *range* non-terminal rule of the *type_interval* rule in LISS.

For this case, we have two options; the first one is to give a number, the second one is to give a variable. And regarding for the variable, we need to check if it is avaible in the symbol table. If it isn't then it means that it isn't declared which will be throw the number 1 error message to the error table (see Table 28). But if the variable belongs to the symbol table, we need to check the type of it. If the type isn't an *integer* then it must throw the number 5 error message to the error table (see Table 28).

Maximum

```
maximum : number
        | identifier
```

Listing 5.38: Maximum rule in LISS

In Listing 5.38, this part refers to the expansion of the *range* non-terminal rule of the *type_interval* rule in LISS. And the behaviour is the same as the *minimum* rule, this means that we need to check the *identifier* terminal (variable) in the symbol table. If it isn't available then it means that the variable isn't declared and must throw the number 1 error message to the error table (see Table 28).

If the variable exists, we need to check his type. If the type isn't an *integer* then it must throw the number 5 error message to the error table (see Table 28).

Satisfy

```
satisfy :
        | 'satisfying' expression
```

Listing 5.39: Satisfy rule in LISS

In Listing 5.39, this part refers to the expansion of the *for_stat* non-terminal rule of the *satisfy* rule in LISS. Basically, the *satisfying* word means that there is a condition who must be calculated and should be 'true' in order to proceed.

In this case, the *expression* is the condition and it must have a boolean type. If the *expression* type isn't a boolean then it must throw the number 7 error message in the error table (see Table 28).

While_stat

```
1 while_stat : 'while' '(' expression ')'
2              '{' statements '}'
```

Listing 5.40: While_stat rule in LISS

In Listing 5.40, this part refers to the use of a condition loop and particularly named as a *while* statement.

For this case, we need to check if the condition (also known as *expression* non-terminal), has the correct type.

Notice that a condition must have a *boolean* type and we need to check if the *expression* type is a boolean. If it isn't then it must throw the number 7 error message to the error table (see Table 28).

Succ_or_pred

```
1 succ_or_pred : succ_pred identifier
```

Listing 5.41: Succ_or_pred rule in LISS

In Listing 5.41, this part refers for incrementing or decrementing a variable in LISS and it is used as a statement. Regarding to this case, we need to check two things.

First, we need to check if the *identifier* (also known as variable) is available in the symbol table. If it isn't then it must throw the number 1 error message to the error table (see Table 28).

In case that the variable is in the symbol table, we need to check his type associated. If the type isn't an *integer*, then it must throw the number 5 error message to the error table (see Table 28).

Tail

```
1 'tail' '(' expression ')'
```

Listing 5.42: Tail rule in LISS

In Listing 5.42, this part refers to the use of a *sequence* function (also known by *Tail* function).

For this case, we need to see if the *expression* type have the correct type in order to work properly.

If the *expression* type doesn't have the *sequence* type, then it must throw the number 7 error message to the error table (see Table 28).

Head

```
'head' '(' expression ')'
```

Listing 5.43: Head rule in LISS

In Listing 5.43, this part refers to the use of a *sequence* function (also known by *Head* function).

For this case, we need to see if the *expression* type have the correct type in order to work properly.

If the *expression* type doesn't have the *sequence* type, then it must throw the number 7 error message to the error table (see Table 28).

Cons

```
'cons' '(' expression ',' expression ')'
```

Listing 5.44: Cons rule in LISS

In Listing 5.44, this part refers to the use of a *sequence* function (also known by *Cons* function).

For this case, we need to see if both of the *expression* non-terminal have the correct type in order to work properly.

If the first *expression* non-terminal (the most left one) doesn't have an *integer* type, then it must throw the number 7 error message to the error table (see Table 28).

Also if the second *expression* non-terminal (the most right one) doesn't have a *sequence* type, then it must throw the number 7 error message to the error table (see Table 28).

Delete

```
delete : 'del' '(' expression ',' expression ')'
```

Listing 5.45: Delete rule in LISS

In Listing 5.45, this part refers to the use of a *sequence* function (also known by *delete* function).

For this case, we need to see if both of the *expression* non-terminal have the correct type in order to work properly.

If the first *expression* non-terminal (the most left one) doesn't have an *integer* type, then it must throw the number 7 error message to the error table (see Table 28).

If the second *expression* non-terminal (the most right one) doesn't have a *sequence* type, then it must throw the number 7 error message to the error table (see Table 28).

Copy_statement

```
'copy' '(' identifier ',' identifier ')'
```

Listing 5.46: Copy_statement rule in LISS

In Listing 5.46, this part refers to the use of a *sequence* function (also known by *copy* function).

For this case, we need to see if both of the *identifier* terminal are in the symbol table and have the correct type in order to work properly.

If one of the *identifier* aren't available in the symbol table, then it must throw the number 1 error message to the error table (see Table 28).

After seeing the availability of both *identifier* in the symbol table, we need to check their type. If both of the variables aren't a *sequence* type, then it must throw the number 5 error message in the error table (see Table 28).

Cat_statement

```
cat_statement : 'cat' '(' identifier ',' identifier ')'
```

Listing 5.47: Cat_statement rule in LISS

In Listing 5.47, this part refers to the use a *sequence* function (also known by *cat* function) and it has the same behaviour as the *copy* statement rule.

This means that we need to check if both of the *identifier* terminal are in the symbol table and have the correct type in order to work properly.

If one of the *identifier* aren't available in the symbol table, then it must throw the number 1 error message to the error table (see Table 28).

Then, after checking the availability of both *identifier* in the symbol table, we need to check their type. If both of the variables aren't a *sequence* type, then it must throw the number 5 error message in the error table (see Table 28).

Is_empty

```
is_empty : 'isEmpty' '(' expression ')'
```

Listing 5.48: Is_empty rule in LISS

In Listing 5.48, this part refers to the use of a *sequence* function (also known by *is_empty* function).

For this case, we need to check, only, the type of *expression* non-terminal.

In order to proceed the correctness of the *is_empty* function, the *expression* non-terminal type must be a *sequence*. If it isn't, then it must throw the number 7 error message to the error table (see Table 28).

Length

```
length : 'length''(' expression ')'
```

Listing 5.49: Length rule in LISS

In Listing 5.49, this part refers to the use of a *sequence* function (also known by *length* function) and it is the same behaviour as *is_empty* function.

We need to check, only, the type of *expression* non-terminal. If the *expression* non-terminal type isn't a *sequence*, then it must throw the number 7 error message to the error table (see Table 28).

Member

```
member : 'isMember''(' expression ',' identifier ')'
```

Listing 5.50: Member rule in LISS

In Listing 5.50, this part refers to the use of a *sequence* function (also known by *member* function). For this case, we need to check firstly the *identifier* and then the *expression* non-terminal.

If the *identifier* terminal is not available in the symbol table, then we must throw the number 1 error message to the error table (see Table 28). Otherwise, if the variable is in the symbol table, we need to check the type of both (*identifier* and *expression*).

Identifier terminal type must be an *integer*. If it isn't then it must throw the number 5 error message to the error table (see Table 28).

If *expression* non-terminal type isn't an *integer*, then it must throw the number 7 error message to the error table (see Table 28).

5.4 CODE GENERATION

In the compiler process, after adding informations to the *symbol table* and searching some inconsistencies to the LISS language (semantic). It is now time to convert the LISS language representation (higher level language) to MIPS assembly code (lower level language).

In the process of converting the language, there is a lot of tasks who will be operated:

- Instruction selection : choosing which type of instruction to use.
- Register allocation : choosing the right register to use for a certain instruction.
- Instruction scheduling : choosing the right time for the instruction to be added in the code.

Let's talk it in the next section every operations and strategy used for the code generation.

5.4.1 *Strategy used for the code generation*

We talked previously about the MIPS architecture and for this section, we will talk about the strategy used for generating the MIPS assembly code regarding to the specific limitation of his architecture.

Data and text part

It is created two variables called *data* and *text* and those two variable have the type *String*. Each time that it is needed to add some information to the assembly code, it will be added regarding to the context of the LISS language.

For example, each variable created at level 0 scope in LISS language code (declarations statement) will be added to the *data* variable. Beside than that, the information will be added to the *text* variable.

Notice that *subprogram* statement (also known as functions) is the only thing that won't go to the *data* variable string, even if it is available in the *declarations* section.

Compiler register strategy

The MIPS architecture has a limit of registers and it is a necessity to use it wisely for generating the code.

So we created. in our project, an array structure with 8 positions which tells us which position are free (array type is boolean). From 0 to 8, it will represent the state of each register from the MIPS register structure.

In this case, the association of the register with the array are :

- Position 0 : register $t0
- Position 1 : register $t1
- Position 2 : register $t2
- Position 3 : register $t3
- Position 4 : register $t4
- Position 5 : register $t5
- Position 6 : register $t7
- Position 7 : register $t8

Each time that the compiler needs or wants a register, it will always see the state of each register by ascending order. Like that, the compiler knows that the latest register needed or used will always be the latest and not in a random order (for a better searching of the register).

Then we need to apply a strategy for not having an overflow of registers being used. Let's explain this situation with an example in Listing 5.51:

```
1  4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12
```

Listing 5.51: Example of a sum operation with some numbers

In Listing 5.51, we see a complex summing operation being done with 9 numbers available. If we wanted to put those 9 numbers to the available register, it will be impossible due to the limitation of the MIPS architecture (only 8 temporary register). So we need to apply a strategy for solving this situation, and it passes by removing information when they are not needed. For this case, we can put the value 4 to the first register ($t0) and put the value 5 to the second register ($t1). Then we add those two values, available in the registers, and put the result in the register $t0. Notice that by doing that, we set the register $t1 free which will be available for adding the next value 6 there and re-apply the same strategy for continuing the sum.

By using this strategy, we won't have an overflow of registers. Take care that some MIPS instructions used, apply that strategy but might be different due to the complexity of some instructions.

Also, notice that the MIPS architecture has some others registers available (saved temporary registers (reserved across call)) and we could have used them to increase the numbers of registers.

But even if we increase the number of registers, the problem is still there and that is why we need to apply a strategy to solve those cases.

Additionally, those saved temporary registers are reserved for jump instructions and for our case we use them for sequence functions only. Regarding to calling functions which uses also jump instruction, we use a different algorithm. We use the stack for storing the information about the function and that is why we don't need to use the saved temporary registers.

Finally, as we use primarly the temporary registers from MIPS architecture in the project, we have a law that dictate that every line statement who is finished (a statement ends with a semicolon), means that the state of those temporary registers must be set to false. This means that each temporary register are free to use.

Address size

In MIPS architecture, we have the ability to optimize the instruction that will be used regarding to the MIPS architecture. But for our case, we won't optimize anything and we will use a fixed size of address. So, we created an *integer* variable that tells us how many bytes does have an address in the MIPS architecture (4 bytes).

The size of the address will be used for creating variables in the heap or even for the *stack*.

Notice that due to the MIPS architecture, it does a fetch with alignment address of the instructions being executed. And that is why we set up a fixed size address and we don't do optimizations for ease debugging and code generation.

Conditional statement

In LISS language, there are different kinds of conditional statements (if-statement, while-statement and for-statement). And each of them uses in MIPS assembly code, some jump instruction.

Remember that, talked previously in the chapter of MIPS assembly, it has a certain pattern to practice when a jump instruction is being used. And we need to use a strategy for those conditional statement.

So we created two variables, one with the type *LinkedList<Integer>* named as *counterJumpStack* and another one with the type *Integer* named as *counterJump*.

Each time that a conditional statement is new, the variable *counterJump* will be concatenated to the name of the condition statement and incremented also. And this is done for one reason, because in MIPS architecture you can't have the same name (when you will use an inconditional jump instruction in MIPS) in the assembly code. If the name was equal then MIPS won't be able to know which name it should jump. And so, when we concatenate the number with the name of the conditional statement (and then it increments), the name in the assembly code will be different and unique.

Regarding to the *LinkedList<integer>* variable, this is a stack for saving informations about the conditional statement explored when there is a high expressivity of conditional statement inside of them. The stack uses a FILO (First In Last Out) system.

Let's an example in Listing 5.52.

```
program test{
    declarations
        i -> integer;
        array1=[1,2,3] -> array size 3;
    statements
        if(true)
        then{
            for(i inArray array1){
                writeln(i);
            }
        }
}
```

Listing 5.52: Example of conditional statements in LISS language

In Listing 5.52, we can see that we use a lot of conditional statement as a snowball effect. So we need to save the information of each conditional statement anywhere and that is why we use a stack (also known as *LinkedList<Integer>*). Each time that a conditional statement appears, the compiler saves the *counterJumpStack* variable value associated to the conditional statement to the stack. If inside of the conditional statement, there is another conditional statement. The *counterJumpStack* variable, meanwhile incremented, will be associated to the new conditional statement and will be added to the stack.

Like that we don't loose the information and we have a traceability regarding to the conditional statement that the compiler has passed throw. And in this way, making MIPS assembly code will be easier and correct.

Notice that each time, the compiler exits a conditional statement, it removes the information in the stack but the *counterJumpStack* variable won't be decremented.

Subprogram name

For this part, we created three structure:

1. LinkedList<String> **functionName**
2. HashMap<String,String> **mipsCodeFunctionCache**
3. String **functionMipsCode**

So the variable *functionName* is a *LinkedList<String>* structure which adds the name of each subprograms that the compiler finds. It uses a FILO system and act as a stack in this case.

Basically, we created the same system as the use of conditional statements. This means that subprograms uses also a jump instruction regarding to the MIPS assembly code and the name of the function must be also unique in the MIPS assembly code.

Each time, that the compiler finds a subprogram name in the LISS code, it adds to the *LinkedList* structure the information. If there is, also, the snowball effect by having a multiplicity of subprograms inside of each one. Then it will add all those informations to the stack.

When we need to add some MIPS assembly code, we just need to take the entire string available in the stack by using the concatenate method and associate the MIPS assembly code to that name.

The variable *mipsCodeFunctionCache* is a *HashMap<String,String>* structure and the key of the HashMap refers to the name of a subprogram (it is the name that is catched in the *LinkedList* structure explained before) and the value is the MIPS assembly code associated to the name of the function.

Basically, that structure save the information of each subprogram with their MIPS assembly code associated. The fact that we used a *HashMap* structure is for the requirements that the name of a subprogram must be unique. And those standards are perfect with a *HashMap* because the key is always unique.

Finally, the variable *functionMipsCode* is a String which hold the MIPS assembly code of a subprogram. When the compiler is creating the MIPS assembly code of a subprogram, it will add to that variable. At the end, it will be added to the HashMap structure whenever it is necessary.

Notice that when the compiler will finish to pass the entire LISS code, it will remove all the informations available in the *HashMap* structure and add it to the string variable *text* (talked previously in the subsection **Data and text part**).

State of functions

In LISS language, there are some MIPS assembly code that are not automatically generated but instead, they are already defined. This is the case for functions like:

- Sequence functions (tail, head, etc...)
- Printing function (write, writeln)
- Read function (input)
- Index out of bound function (related to the array type)

And instead of adding those defined function in the MIPS assembly code, when they are not used in the LISS code. We created a structure which tells us if those functions will be used or not.

- HashMap<String, Integer> **functionStateUsedOrNot**

Basically, the idea behind that structure is that if a function was used, it will put the function (available in the *HashMap* structure) set to 1(1 means true). When the compiler finishs to see the entire LISS language code, it will check the variable *functionStateUsedOrNot* and see if some functions are set to 1. If it is, then it will add at the end of the generated MIPS assembly code, the appropriated and defined MIPS assembly code of that function to the end.

Notice that the variable will put the state of every function available to 0 before the compiler begins in generating the code (0 means false).

Stack

For this project, we created a structure which behaviour a stack.

This structure is basically for searching some informations regarding to variables when they are created and not available in the level scope 0. Those variables are normally variables who were created on function with a level scope greater than 0 and they will be stored to the stack of the MIPS architecture. So, this structure is here for one reason, instead of using a lot of MIPS assembly code for searching some informations (in this case, those variables). The application will use an algorithm which will find the position directly of the variable by calculating it and using only one instruction.

Let's see the structure of the stack in Figure 12.

In Figure 12, we see two rectangle objects. One is named *levelStackSP* and the other is named *stackSP*.

1. ArrayList<Integer> **levelStackSP**
2. ArrayList<Integer> **stackSP**

The *levelStackSP* is a stack (*ArrayList type*) which contains information about the other stack *stackSP* (*ArrayList type*) and the main objective of this stack is to add informations related with subprogram created in the LISS code (uses a FILO system). Each **position** of the stack *levelStackSP* refers to a certain level scope, so we have in this example 3 levels (level scope 0 (position 0), level scope 1 (position 1) and level scope 2 (position 2)). And each position of the stack *levelStackSP* contains information about the position of the other stack *stackSP*.

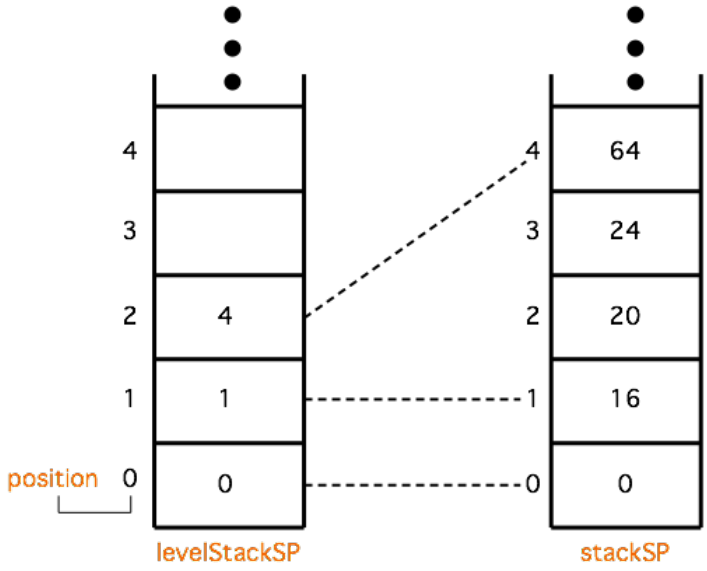


Figure 12.: Stack structure

The stack *stackSP* is the main stack which holds every *push* instruction in the MIPS assembly code and behaves as a FILO system also. Each time that the compiler adds some information into the stack (regarding to the MIPS assembly code), it will also be added into the stack *stackSP* (last free position available). Also, every *pop* instruction, in the MIPS assembly code, will be removed as well as in the stack too (it removes the last position who has information).

To make a summary, we have two structure. One stack (*levelStackSP*) who gives us information about the position of any subprogram that the compiler have found to the other stack *stackSP*. And one stack (*stackSP*) who behaves the stack of MIPS architecture.

Notice that each time that the stack *stackSP* will add some information, it will add in the next free position available and will add the amount to store with the previous position. In this example, we can see that the compiler wanted to add 40 bytes of memory. So he just added to the next free position available (position 4), 24 plus 40 bytes which is equals to 64 (see the position 4 value in the stack *stackSP*).

Regarding to removing informations in the stack *stackSP*, it just removes the last position who have information. When the last position of the stack *stackSP* is linked to the stack *levelStackSP* (in the example, position 2 of stack *levelStackSP* is linked to position 4 of stack

stackSP). By removing the last information of the stack *stackSP*, it will also remove the last information of the stack *levelStackSP*.

The algorithm for finding variables in the stack

The algorithm of searching the position of a certain variable is simple. Whenever we need a variable greater than the level scope 0 (variable only available in the stack MIPS architecture), we need to check at the symbol table his availability. If the variable is there, we need to check the level scope of the variable. If the level scope is greater than 0, then we need to check at our stack structure where is the position in the stack.

So, in this case, we need to check firstly the stack *levelStackSP*. With the knowledge of the level scope of the variable, we access to the right position of that stack. Then by accessing it, we will know which position is in the other stack *stackSP*. By knowing the position of the stack *stackSP*, we just need to calculate the position of the variable in the stack. And the calculation is done by this way :

1. Take the last value added to the stack *stackSP*.
2. Take the value found regarding to the variable in the stack *stackSP*.
3. Do a substracton of those values (step 1 - step 2).
4. Get the address value regarding to the variable in the symbol table.
5. Add the address value to step 3.
6. Position found in the stack.

After that, we just need to create one mips instruction which gets the right value of the position in the stack MIPS architecture regarding to the variable.

5.4.2 *LISS language code generation*

This part will talk about the code generation of every statements feasible in the LISS language. It will be divided by sections for each statements and then divided by the level scope equally to 0 or greater for a better view regarding to the complexity done with the MIPS architecture and his requirements.

Creating a variable in LISS

- Level scope equals to zero

Let's see an example of LISS code in Listing 5.53.

```
1  program liss{
2    declarations
3      a, b = 4, c = -1, d = +2 -> integer;
4      flag, flag1 = false, flag2 = true -> boolean;
5      array1, array2 = [2,1,1], array3 = [1] -> array size 3;
6      array4 = [[1,2],[3]] -> array size 3,3;
7      set1, set2 = { y | y+1 < y+4}, set3 = {} -> set;
8      seq1, seq2 = <<1,2>> -> sequence;
9    statements
10 }
```

Listing 5.53: Example of creating variables in LISS

In Listing 5.53, we can see some variables being declared in the LISS code in the level scope 0. In this case, the compiler needs to take those informations and generate them to MIPS assembly code. Let's go line by line and explain them each one.

In line 3 of Listing 5.53, we see 4 different named variables being declared with the type *integer*. The compiler adds them to the symbol table and do some checkings regarding to the semantic system implemented. Then, if everything is all right, it associate them (each variable) with a certain address to each one. Remember that the type *integer* cost 4 bytes in the memory as explained before. So, in this case, it will generate the address 0, 4, 8 and 12 for those variables.

Also, notice that the association of those addresses are not set in the same order as the variables were declared. This is due to the fact that those variables are stored in a *HashMap* structure where the key is the name of the variable and the value are informations regarding to the variable. And we implemented a *HashMap* structure for the case that the line (which the compiler will process entirely) will check if names of variables are differents.

In the end, when the compiler will take the information of those variables for putting them into the symbol table and regarding to the *HashMap* structure in JAVA which doesn't have the notion of ordering keys. It will simply take the keys in any orders and the compiler will associate those keys (the name of the variable) with an address created. Take care that there is no problem in doing that, because the compiler always knows the address of each variables (symbol table structure holds the information).

Later, we need to generate the assembly code if everything worked as planned. And this is done by declaring them in the data section of the MIPS assembly code (see in Listing 5.54).

```
1  .data
2  a : .word 0    # 3:4
3  b : .word 4    # 3:11
4  c : .word -1   # 3:19
```

```
5  d : .word +2     # 3:27
```

Listing 5.54: Code generation of integer variables in MIPS assembly code

So creating variables in the level scope 0, basically means that those variables are globals. And in this case, we add them to the *data* section otherwise it will be in the stack.

By creating those variables, in the MIPS assembly code, the way of declaring them is different than if it was in a greater level scope. In Listing 5.54, we do by associating the name of the variables (*a*) following by the size type of the variable (*.word* (4 bytes)) and the value that the variable will store.

Notice that a variable who isn't declared will store the value 0.

In line 4 of Listing 5.53, we create the boolean variables and this is done in the same way as an *integer* variable. Remember that *boolean* types cost 4 bytes in the memory. So we just need to do the same way as if it was an *integer* type.

We declared the name of the variable (*flag*), then we say the size type (*.word* (4 bytes)) and we write the value of the boolean (true is 1, false is 0) (see in Listing 5.55).

```
1  flag : .word 0    # 4:4
2  flag2 : .word 1   # 4:33
3  flag1 : .word 0   # 4:18
```

Listing 5.55: Code generation of boolean variables in MIPS assembly code

Notice that boolean variable who are not initialized, the default value is false.

In line 5 and 6 of Listing 5.53, we declare some array type variables. The idea of an *array* type is a fixed-size sequential collection of elements with the same type. And in MIPS assembly code, there is a certain way of creating those types by doing with this way (see in Listing 5.56).

```
1  array2 : .space 12    # 5:12
2  array1 : .space 12    # 5:4
3  array3 : .space 12    # 5:30
4  array4 : .space 36    # 6:4
```

Listing 5.56: Code generation of array variables in MIPS assembly code

In Listing 5.56, we declare the name of the variable, then the size type (sequence of memory, *.space*) (due to the fact that it is an *array* type) and finally, how much space that the array will store. Take care that *array* type in LISS, only store integer values and for calculating the space regarding to the *array* variable, we need to do some calculation.

The calculation is done by multiplying all the limits of the *array* variable and with that result we multiply by the number 4 (space of an *integer* variable). Regarding to the line 5 in

Listing 5.53, the calculation for the variables *array1*, *array2* and *array3* is done by taking the limits 3 and multiply it by 4 (space of an integer), which is equal to 12. However regarding to line 6 in Listing 5.53, the calculation for the variable *array4* is done by multiplying all the limits associated to the variable (3x3 which is 9) and then, multiplying by 4 (space of an integer), which is equal to 36. And the strategy is the same if the dimension of the *array* variable is greater regarding to the calculation of generating the space of the *array*.

Now that we declared the space of those variables in MIPS assembly code, we need to declare the values associated to those variables.

So we implemented a system which takes the information of each position of the array regarding to the value that was declared in the array.

For example, if we have a multidimensional array with 3 dimensions like that :

```
1  array1 = [[[12]],[[5,6],[7]]] -> array size 2,2,3;
```

Listing 5.57: Example of an array with 3 dimensions

We need to create a system which will take the informations regarding to the array declared and this pass by taking the informations of some values who are declared in the array (see Figure 13).

So we created a system which has this structure (see in Listing 5.58).

```
1  ArrayList<ArrayList<Integer>> accessArray
```

Listing 5.58: Structure of saving informations of each index in JAVA

Basically, it is a structure where one *ArrayList* holds the informations of one index of the array processed and add it to the other *ArrayList* whenever it have completed to process the information (it behaves like a stack).

So, in Figure 13, we can see clearly that the left rectangle is the stack where each position of it, holds informations (a *ArrayList* of integer informations) regarding to one index declared in the array.

And this *LinkedList* of informations has a certain architecture which must be explained. The size of that *ArrayList* is equal to the dimension of the *array* plus one (refers to the value available in the index processed). Then, the first positions of the *ArrayList* are reserved for each dimension of the array and the last position of the *ArrayList* is the value which needs to be stored in that index of the array. Each dimension will inform us which position has the value.

For example, regarding to the example in Figure 13.

The first information available in the stack is in the position 0 and this information is telling us that there is a value to be stored at the position [0,0,0] with the value 12. The

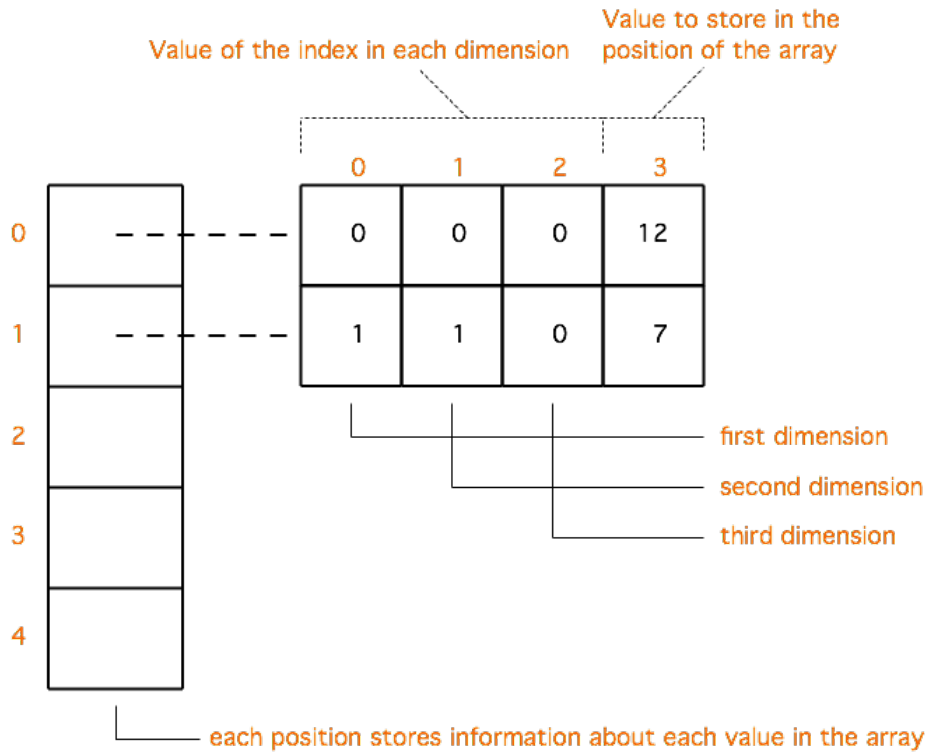


Figure 13.: Structure for saving information of each value declared in a array

second information, available in the position 1 of the stack, is telling us that there is another value to be stored at the position [1,1,0] with the value 7.

After getting all those informations, we need to generate the instructions and for that we need to calculate the right position of each value with the information that it was processed.

The calculation is done with the next formula (see Equation 1).

$$p(l,a) = \sum_{i=0,\ i\neq n-1}^{n-2} (a[i] \times \prod_{j=i+1}^{n-1} l[j]) + a[n-1] \tag{1}$$

In Equation 1, the equation needs two inputs:

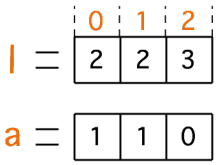
- **l** - array variable which has the informations about the limits of the array in question.
- **a** - array variable which has the information of the position of the array that need to be processed.

Notice that the variable **m**, in the equation, is equal to the dimension of the array.

Then after getting those inputs variables, it calculates the position of the array in question for any n-dimensional size. Also, take a note that if the dimension of the array is equal to 1, the equation doesn't compute the first part (due to the restriction of the equation).

And to understand the formula, let's explain it with an example.

Imagine that we have those input variables for the formula (examples taken from Figure 13 and Listing 5.57):



By using the equation above with that example, let's unroll it.

$$\begin{aligned} p(l,a) &= a[0] \times l[1] \times l[2] + a[1] \times l[2] + a[2] \\ p(l,a) &= 1 \times 2 \times 3 + 1 \times 3 + 0 \\ p(l,a) &= 6 + 3 + 0 \\ p(l,a) &= 9 \end{aligned} \tag{2}$$

So, with that calculation we can see that the position of the array is the number 9.

Let's see throw the next Figure 14 if the calculation was done correctly.

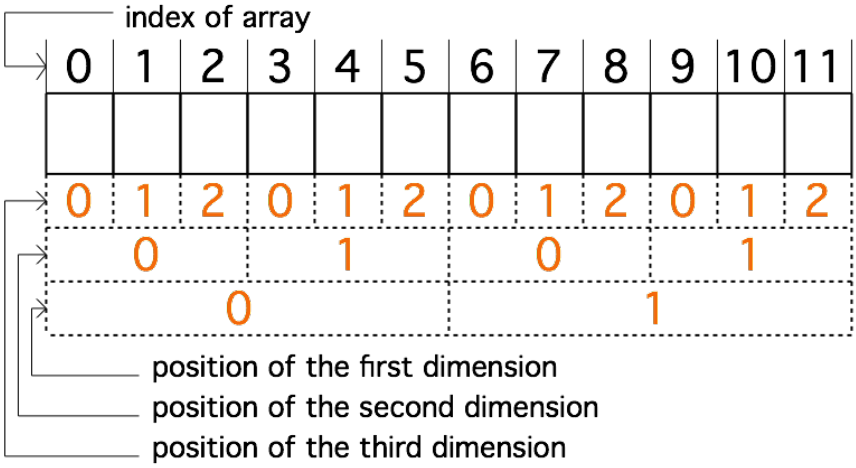


Figure 14.: Array structure with size 2,2,3.

Using the positions of the variable array **l** and using them to find the right position in the array structure in Figure 14, we go firstly to the right position of the first dimension (number 1 (l[0] = 1)). Then, we go to the second dimension of the part where belongs the number 1 in the first dimension, in this case it is the number 1 (l[1] = 1). Finally, we go to the last dimension and go to the number 0 (l[2] = 0). As we can see, it goes directly to the index number 9 of the array.

This proves that the algorithm works very well and also, how it calculates the position for an array with the structure implemented in this project.

Now that we know how the position is calculated in an array; let's continue to unroll the line 5 and line 6 in Listing 5.53.

So, after creating the space of those variables *array2*, *array3* and *array4*, we need to initialize them.

Let's see an example of how it is done the initialization of arrays in Listing 5.59.

```
1  .text
2    main:
3       ##### Initialize Value Array :array2#####
4       li $to,2    # 5:12
5       li $t1,0    # 5:12
6       sw $to, array2($t1)    # 5:12
7       li $to,1    # 5:12
8       li $t1,4    # 5:12
9       sw $to, array2($t1)    # 5:12
10      li $to,1    # 5:12
11      li $t1,8    # 5:12
12      sw $to, array2($t1)    # 5:12
13      ########################################
```

Listing 5.59: MIPS assembly code generated for the variable array2

In MIPS architecture, it is impossible to declare the array with the values associated in the declaration parts. So, we need to fix this situation and this is done by creating MIPS assembly code in the flow of the program execution.

Basically, the idea is that the MIPS assembly code is always in the first place regarding to the flow of the program execution. In this case, we can see in Listing 5.59 that the MIPS assembly code for the initialization of the *array2* variable comes first in the flow of the program execution. Then after that every initialization was made regarding to arrays, comes and begins the flow of the program execution.

In Listing 5.59, let's explain how the code generation works for values regarding to the array:

- line 4 - Loading the value 2, this is the value to be stored in the array.
- line 5 - Loading the position 0, this is the position which the value will be stored (use the algorithm for calculating the position).
- line 6 - Store the value 2 to the position 0 in the array2 memory.
- line 7.... - Continue to use the same strategy with the next values that needs to be added.

Storing one value in an array needs three MIPS instruction assembly code.

Notice that the position calculated is always multiplied, at the end, by the size of an *integer* (number 4).

As we can see in Listing 5.59, the position are :

- line 5 - the value is 0 => position 0 (0/4 = 0)
- line 8 - the value is 4 => position 1 (4/4 = 1)
- line 11 - the value is 8 => position 2 (8/4 = 2)

Also, take care that every other positions in the array have the value 0 and that is why we don't need to create the MIPS instructions for them, because the default value is 0 in an array non-initialized (the story changes when those arrays are created in a level scope greater than 0, but it will be talked further).

In line 7 of Listing 5.53, we see two differents named variables being declared with the type *set*.

That type basically doesn't create any informations in the MIPS assembly code for the declarations parts. Instead it saves the information in a specific structure created for that purpose. The structure is made with the concept of a Tree structure where there are some nodes with branches or not, associated to others nodes.

And this structure is made by two JAVA class:

- Node Class
- Set Class

The Node JAVA class is a class where represents the concept of a node structure in a tree. It is represented by three things:

1. String data
2. Node left
3. Node right

The variable **data** refers to the value represented of that node, the variables **left** and **right** refers to a node who might be to the left or right side of the actual node.

Now, the Set class is a class where it saves the information of the set in a tree structure and the free variable associated with the set. Take care, that the Set class uses and abuses the Node class. It is represented by two things:

1. ArrayList<Node> identifier

2. Node head

The variable **identifer** refers to a list of free variables that are stored in the tree structure. This is done for one particularly reason, instead of browsing the entire tree and looking to those free variables, we have a list where we can change the state of those free variables available in the list and directly, it changes also in the tree. The advantage of that structure is that we don't need to browse in the tree and change or look for those free variable. Otherwise it will be a time consume by doing that.

Then we have the variable **head** which holds the information about the head node of the tree structure.

Let's see the Set structure in Figure 15.

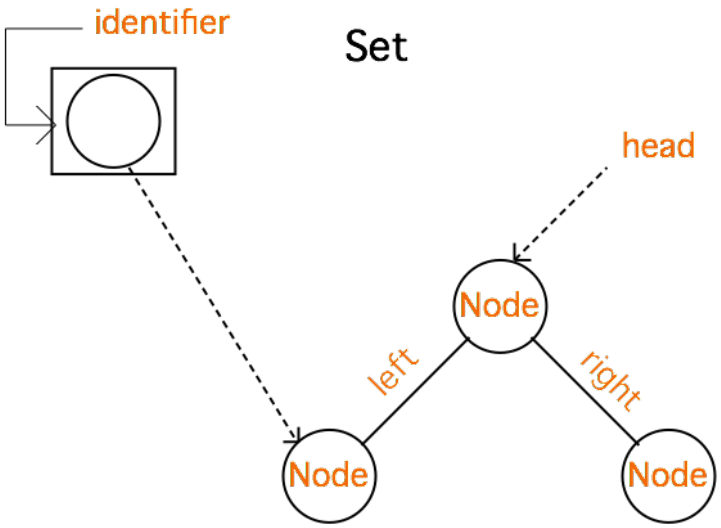


Figure 15.: Set structure in JAVA

Basically, a set in LISS is made with a tree structure where each nodes are a symbol of the expression associated with the set. For our project, we managed to create a variable which points to the head of the tree structure (also named by the variable *head*) and a list of free variables stored in the variable named *identifier*.

We implemented a list for free variables for one reason and this reason comes with the fact that we can join multiple sets. Notice that each set have their own free variable which means that each free variable regarding to each sets does have differents address. And we need to collect all those addresses of each free variable with a list.

Let's see an example of joining two sets in Figure 16.

In Figure 16, we can see that the head of the **Set** is connected with two sets (left and right). This means that we are joining two sets (*Set1* and *Set2*) with the head of one *Set*. And, in this case, we are growing the tree structure by doing more complex structure.

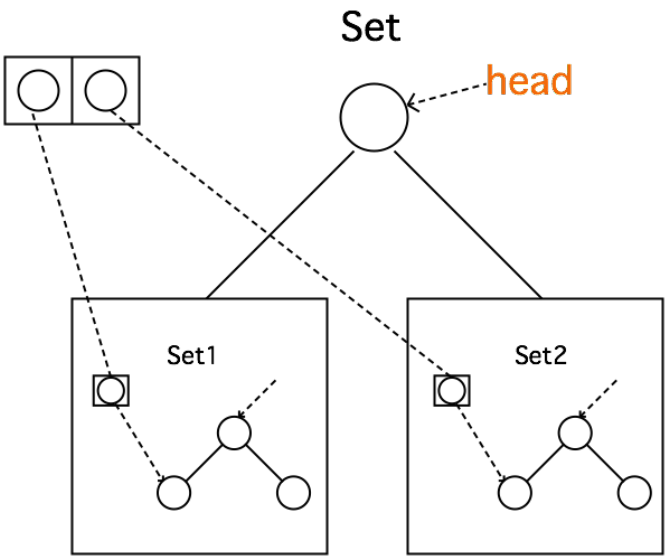


Figure 16.: Set structure in JAVA

Doing those changes will need some attention to fix the free variables available on each sets. And this is why we have created the set class with a list of free variables. Like that, we can get the free variables from the others sets and adds them to the list of the **Set**. By this way, we can change the states of each free variables available in the list of **Set** which will also changes in the other sets too.

After that the compiler, processed the Set class for the variable, it will associate that structure to the variable added in the symbol table.

Notice that the compiler will generate some code only when it will be asked it in the *statements* part of the LISS code.

Lastly, we need to talk about the different states that a set can have regarding to the declaration part.

Set variable can have three different states:

- Universe set
- Empty set
- Defined set

The **universe** set is a set which basically represent the whole number system and it is represented by this syntatic in LISS code:

```
1  set1 -> set;
```

The **empty** set is a set which basically represents nothing in the number systems, also considered *null* and it is represented by this syntatic in LISS code:

```
1  set1 = {} -> set;
```

The **defined** set is a set which basically represents the numbers expressed with the expression associated to the set and they are defined by this way in LISS code:

```
1  set1 = {y | y+1 < y+4} -> set;
```

Let's talk finally with the last line 8 in Listing 5.53, it represents the declaration of variables with the type *sequence*. And the idea of generating the code for that type, is almost the same as the type *array*.

Basically, the type *sequence* creates one position of memory for each variable declared and will store the value -1 in there (value -1 is equal to NULL). By setting the value to -1, it means that the variable sequence is empty, no values are associated to that variable (the same as saying that the sequence wasn't intialized).

Let's see the code generated for the example available on line 8 of Listing 5.53:

```
1  seq2 : .word -1    # 9:10
2  seq1 : .word -1    # 9:4
```

As we can see, we create the name of the variable firstly, then the type of the variable (*.word)* and the value associated to that variable (NULL value (-1)). Notice that the type of the variables is a 4 bytes size and this is for one main raison.

The *sequence* type is a linkedlist of integer numbers and those numbers will be stored in the *heap* section. So, in this case, we need to know in which address the first element of the sequence is stored and this pass by knowing the address of the first element of the sequence. However this address must be stored at one place that can be known and this goes by storing that address to the variable name associated.

The size of an address in the MIPS architecture is 4 bytes, so in this case the variable must be 4 bytes long and that is why we choosed the type *.word*.

After creating the variables, we need to do the same thing as the *array* type, check if the sequence is initialized or not.

And if it is initialized, we need to generate some MIPS assembly code.

For generating the code, we need to take the values that are associated to the sequence and generate each MIPS assembly code for each values.

Notice also, that the MIPS assembly code that will be generated, will also be placed in the same area as the initialization of an array (before the program execution code).

Let's explain it throw the example of Listing 5.53, the code generated for the variable **seq2**.

```
1 ##### Initialize Sequence :seq2#####
2 lw $so, seq2     # 9:10
3 li $s1, 1    # 9:10
4 jal cons_sequence    # 9:10
5 move $so, $vo    # 9:10
6 li $s1, 2    # 9:10
7 jal cons_sequence    # 9:10
8 sw $vo, seq2     # 9:10
9 ######################################
```

Listing 5.60: Code generated for the sequence variable

Basically, in our project we need some inputs in order to add some values to a sequence and those informations are:

1. the name of the variable (for having the knowledge of which sequence that the number must be associated).
2. the value that needs to be added to the sequence.
3. the function who will do the work of adding the number to the heap and linking it to the variable *sequence*.

So, regarding to the variable **seq2** in Listing 5.53, the compiler needs to take the value 1 and 2 and generate code for them to the sequence.

In Listing 5.60, the compiler firstly creates an instruction which puts the address of the sequence variable to a saved registers ($ *so*), then it loads the value 1 (first number that must be added to the sequence) to the next saved registers ($ *s1*) and finally it calls the function that will add to the sequence (*cons_sequence*).

Notice that those steps are always the same and regarding to the use of those saved registers, the reason is that we don't need to use the stack for storing the information. Instead we use some profits of the MIPS architecture, and we use those saved registers.

Care that normally we use the saved registers for calling some functions due to the fact that those registers won't be modified within that transition of jumping to those functions. And for our case, we call the function that will add the number to the sequence.

After that the function will finish to process (*cons_sequence* function), it will return to the register $vo the return value and in this case, this is the address of the first element of the sequence. Then in line 5, it will move the address of the register $vo to the register $so, due to the fact that it needs to add the second number (number 2) to the sequence. And finally, at the end it will store the address of the first element of the sequence to the sequence variable name (line 8).

- Level scope greater than zero

Creating variables with a level scope greater than 0 means that those variables are created in a function. Let's see how they are created in Listing 5.61.

```
1  program liss{
2    declarations
3      subprogram test(){
4        declarations
5          a, b = 4, c = -1, d = +2 -> integer;
6          flag, flag1 = false, flag2 = true -> boolean;
7          array1, array2 = [2,1,1], array3 = [1] -> array size 3;
8          array4 = [[1,2],[3]] -> array size 3,3;
9          set1, set2 = { y | y+1 < y+4}, set3 = {} -> set;
10         seq1, seq2 = <<1,2>> -> sequence;
11       statements
12     }
13   statements
14 }
```

Listing 5.61: Example of creating variables in a level scope greater than 0

As we can see in Listing 5.61, they are created in the same way as if it is with a level scope equals to 0. The only thing that is different is that they are created in a different area (subprogram area) and this means that the every declaration of variables will be declared and stored in the stack memory.

Basically, when the compiler process a function within the LISS code, it will process firstly the arguments that the function has and secondly, every variable declaration under the *declarations* part.

When the compiler have added those informations to the symbol table, he will also calculate the total size that needs to be allocated to the stack memory in MIPS architecture.

Notice that before calculating the total size of the stack memory, the compiler has already processed the information of every variables and arguments into MIPS code instruction.

Let's see how the stack is organized in Figure 17.

So, in Figure 17, the stack is organized by this way:

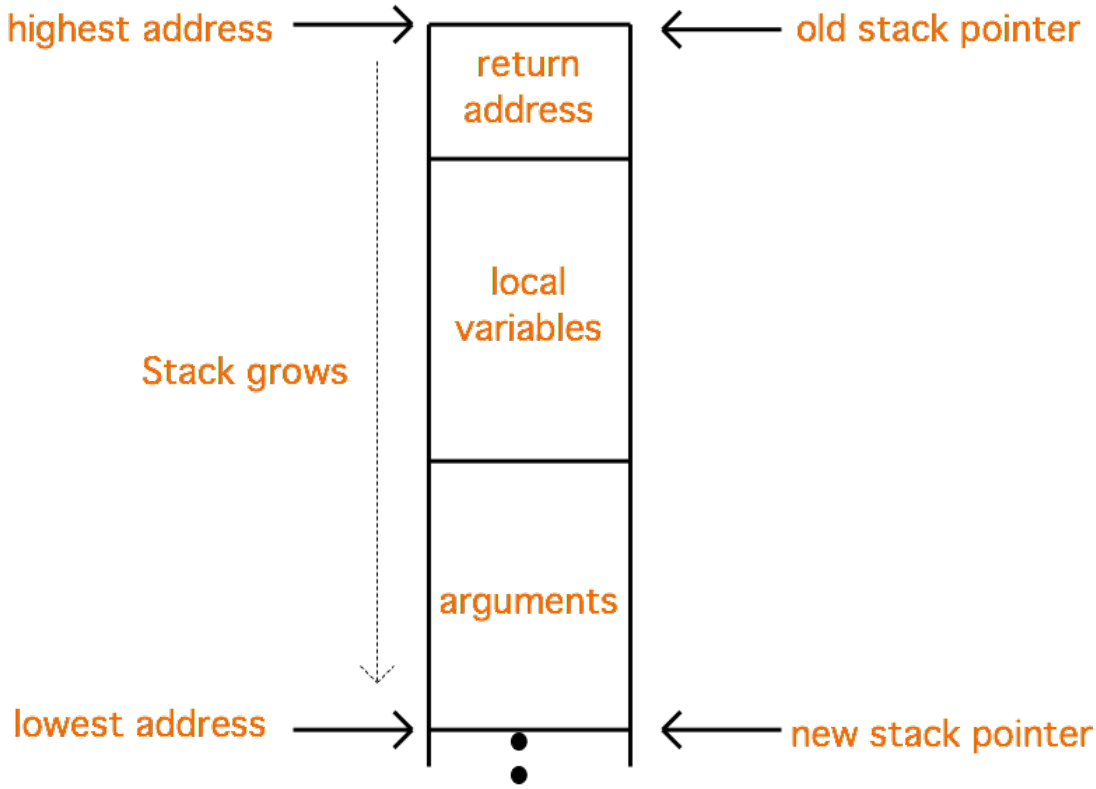


Figure 17.: Architecture of the stack relatively to a function in LISS

- the variables of the arguments function next to the new position of the stack pointer
- the local variables relatively to the variables declared under the declarations parts.
- the return address of the function.

Notice that the stack grows from the highest address to the lowest one and that there is also a reason regarding to this chosen architecture.

Remember that each variables that the compiler finds and needs to add to the symbol table, an address is also associated to the variable. When the compiler enters to a function statement (declaration of a subprogram in LISS), the address is set to zero. And in this case, each variables who are found, will begin by the address zero and then incremented relatively to their types.

Now, if we want to access to a variable in that stack, we just need to get the address of the variable throw the symbol table and get the position of the *new stack pointer* (if the level scope of the compiler, when he is processing the access, is the same as the level scope of the variable). Then we add both of them and get the position for accessing the value of the variable in the stack.

Care that, in Figure 17, it is possible that the arguments section or the local variables section might not appear in the stack (depending to the LISS code) and also that those informations are added to the stack created under the project.

Now, let's explain the code generated in MIPS relatively to the Listing 5.61.

The fact that it is a subprogram (function), every code generated will be added relatively to the branch associated of the function name in MIPS and the first thing that must be done under that branch is to increase the stack relatively to the amount of informations that needs to be allocated (see in Listing 5.62).

```
1   test:
2     addi $sp, $sp, -160
3     sw $ra, 156($sp)
```

Listing 5.62: Initialization of MIPS code generated for a function

In Listing 5.62, we see the name of the branch associated to the function in first. Then comes the part of adding some informations about the stack that the function needs to allocate. In this case, we have one instruction *add* (line 2) which explains us that the function needs to allocate 160 bytes in the stack memory regarding to the stack pointer and to refresh the address of the new stack pointer with the new allocation. Finally, we save the information regarding to the return address of the function into the stack (line 3).

Those are always the initialization of MIPS instruction regarding to a function declared in LISS. After that, comes the MIPS assembly code relatively to the variables declared under the *declarations* part of the function.

In line 5 of Listing 5.61, we are declaring integer variables and the code generated for those variables are available in Listing 5.63.

```
1  li $to,0     # 12:12
2  sw $to, 0($sp)
3  li $to,4     # 12:19
4  sw $to, 4($sp)
5  li $to,-1    # 12:27
6  sw $to, 8($sp)
7  li $to,+2    # 12:35
8  sw $to, 12($sp)
```

Listing 5.63: Declaring integer variables in level scope greater than 0 on MIPS.

In Listing 5.63, line 1 and 2 are related for declaring the variable **a**. Basically, the idea of declaring integer variable is done by this way:

1. load the value to store in the variable

2. store that value to the position related to the variable in the stack.

Notice that the position is given by the algorithm that we explained in a previous section (stack structure).

Now, line 3 and 4 are the code generated for the variable **b**; line 5 and 6 are the code generated for the variable **c**; line 7 and 8 are the code generated for the variable **d**.

Line 6 of Listing 5.61, refers to variables declared with the type boolean and the code generated for that type is visible in Listing 5.64.

```
1  li $to,0     # 13:12
2  sw $to, 16($sp)
3  li $to,1     # 13:41
4  sw $to, 20($sp)
5  li $to,0     # 13:26
6  sw $to, 24($sp)
```

Listing 5.64: Declaring boolean variables in level scope greater than 0

In Listing 5.64, the methodology of creating boolean variables is the same as if it was with a level scope equals to zero. The only thing that differs, is the instruction for storing the value that is associated to the variable.

Line 1 and 2 of Listing 5.64 refers to the declaration of the variable **flag**; line 3 and 4 is the declaration of the variable **flag2** and line 5 and 6 is the declaration of the variable **flag1**.

Line 7 and 8 of Listing 5.61, refers to variables declared with the type array and the code generated for the variable **array2** is visible in Listing 5.65.

```
1  ##### Initialize Array :array2#####
2  li $to,0     # 14:20
3  sw $to, 28($sp)
4  li $to,0     # 14:20
5  sw $to, 32($sp)
6  li $to,0     # 14:20
7  sw $to, 36($sp)
8  ########################################
9  ##### Initialize Value Array :array2#####
10 li $to,2     # 14:20
11 li $t1,0     # 14:20
12 li $t2,28    # 14:20
13 add $t1, $t1, $t2 # 14:20
14 add $t1, $t1, $sp
15 sw $to, ($t1)
16 li $to,1     # 14:20
17 li $t1,4     # 14:20
```

```
18  li $t2,28    # 14:20
19  add $t1, $t1, $t2 # 14:20
20  add $t1, $t1, $sp
21  sw $to, ($t1)
22  li $to,1     # 14:20
23  li $t1,8     # 14:20
24  li $t2,28    # 14:20
25  add $t1, $t1, $t2 # 14:20
26  add $t1, $t1, $sp
27  sw $to, ($t1)
28  ######################################
```

Listing 5.65: Declaring array variables in level scope greater than 0

So the creation of a variable with the type array is done almost by the same way as if it was with a variable declared in the level scope equals to zero. But it differs on some points which must be explained.

As we know, the stack grows and decrease by time and the values in the memory of the stack are not removed regarding to the process of the stack by growing up or decreasing it. So, regarding to variables with the type of array, we need to firstly set to zero all the position of the array in the stack (done on line 2 to 7 of Listing 5.65). Then, we need to put the values that were declared in the array to their right position.

Let's see how it is done with the value 2 of the variable **array2** relatively with the code made in Listing 5.65:

1. line 10 : Put the value 2 to register.
2. line 11 : Put the index of the value 2 regarding to the array declared to register.
3. line 12 : Put the address of the array to register.
4. line 13 : Sum the index with the address of the array for getting the address of the index.
5. line 14 : Sum the index address with the stack pointer, for getting the address position in the stack.
6. line 15 : Store the value 2 to the address position in the stack.

And redo the same algorithm for the next values that need to be stored. Also notice that the index of the array is calculated throw the algorithm mentioned before and that the address of the array is associated to the variable and caught with the symbol table.

Line 9 of Listing 5.61, refers to the declaration of variables with the type set and it only gets and creates the tree structure which will be associated to each variables. Nothing more will be generated as code, unless if it is used in the *statements* section.

Line 10 of Listing 5.61, refers to the declaration of variables with the type sequence and the code generated for the variable **seq2** is visible in Listing 5.66.

```
1  ##### Initialize Sequence :seq2#####
2  li $t2,-1   # 18:18
3  sw $t2, 148($sp)    # 18:18
4  lw $so, 148($sp)    # 18:18
5  li $s1, 1   # 18:18
6  jal cons_sequence   # 18:18
7  move $so, $vo   # 18:18
8  li $s1, 2   # 18:18
9  jal cons_sequence   # 18:18
10 sw $vo, 148($sp)    # 18:18
11 ######################################
```

Listing 5.66: Declaring sequence variable in level scope greater than 0

The idea of generating the code for the sequence variable is the same as an array variable. Firstly, we need to put the value NULL in the stack memory (line 2 and 3 of Listing 5.66) and then, if some values are associated to the variable, we need to generate the code (which is almost the same way as if it was on a level scope equals to 0).

So, firstly we go get the address of the variable sequence which needs to add a certain value (line 4 of Listing 5.66), then we load the value to a register (line 5 of Listing 5.66) and finally, we call the function that will process the concatenation of the value to the sequence.

6

SDE: DEVELOPMENT

Before we try to explain the concept of a Syntax-Directed Editor (SDE) (Reps and Teitelbaum, 1989b; Ko et al., 2005; MI-students et al., 2010; Teitelbaum and Reps, 1981; Reps et al., 1986; Reps and Teitelbaum, 1989a; Arefi et al., 1989), let's start defining what is an Integrated Development Environment (IDE).

An IDE is described as a software application that provides facilities to computer programmers for software development. It consists , normally, of a source code editor, a compiler, a debugger, and others tools. IDEs are designed for maximizing the productivity of programmers with visual interface and contains, normally, an interpreter, a compiler or both (see Figure 18).

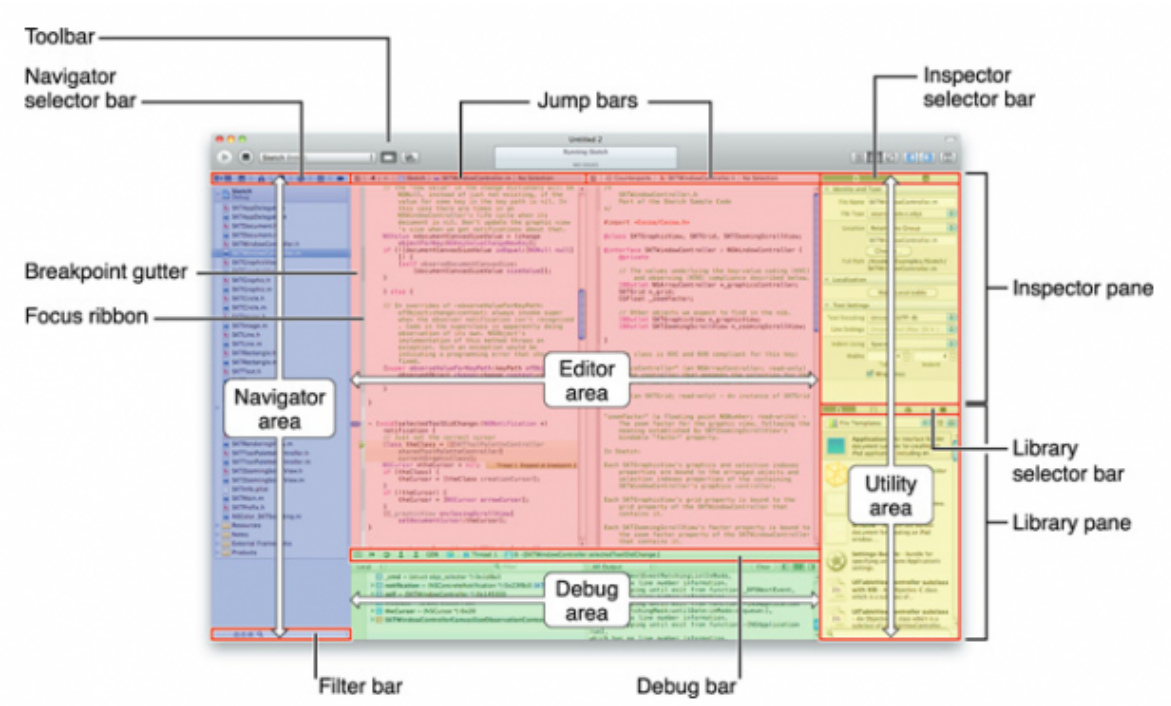


Figure 18.: Example of an IDE visual interface (XCode) [1]

Programs are created top down in the editor sections by inserting statements and expressions at the right cursor position of the current syntactic template and we can, by the cursor, change simply from one line of text to another one.

A SDE has the same approach of an IDE which is (as said above) an interactive programming environment with integrated facilities to create, edit, execute and debugging programs. The difference between them is that SDE encourages the program writing at a high level of abstraction, and promotes the programming based on a step by step refinement process.

It liberates the user from knowing the language syntactic details while editing programs.

SDE is basically guided by the syntactic structure of a programming language in both editing and execution. It is a hybrid system between a tree editor and a text editor.

The notion of cursor is really important in the context of SDE because, when the editing mode is on, the cursor is always located in a placeholder of a correct template (see next section) and the programmer may only change to another correct template at that placeholder or to its constituents.

It reinforces the idea that the program is a hierarchical composition of syntactic objects, rather than a sequence of characters.

6.1 WHAT IS A TEMPLATE?

The grammar of a programming language is a collection of production (or derivation rules) that state how a non-terminal symbol (LHS) is decomposed in a sequence of other symbols (RHS). A template is just the RHS of a grammar rule. Templates cannot be altered, they have placeholders for inserting a phrase or another template and they are generated by editor commands, according to the grammar production.

```
1 IF( condition )
2   THEN statement
3   ELSE statement
```

Listing 6.1: Example of a IF Conditional template

In Listing 6.1 we can see the editor template for the if-statement, where *condition* and *statement* are placeholders.

The notion of template is very important because templates are always syntactically correct for two reasons:

1. First, the command is validated to guarantee that it inserts a template permitted.
2. Second, the template is not typed, so it contains no lexical errors.

So a correct program (i.e., a valid sentence of the programming language) is created by choosing templates and replacing placeholders by others templates or by concrets values (numeric or string constants or identifiers).

To clarify the definition of SDE, we will explain it with the help of an example.

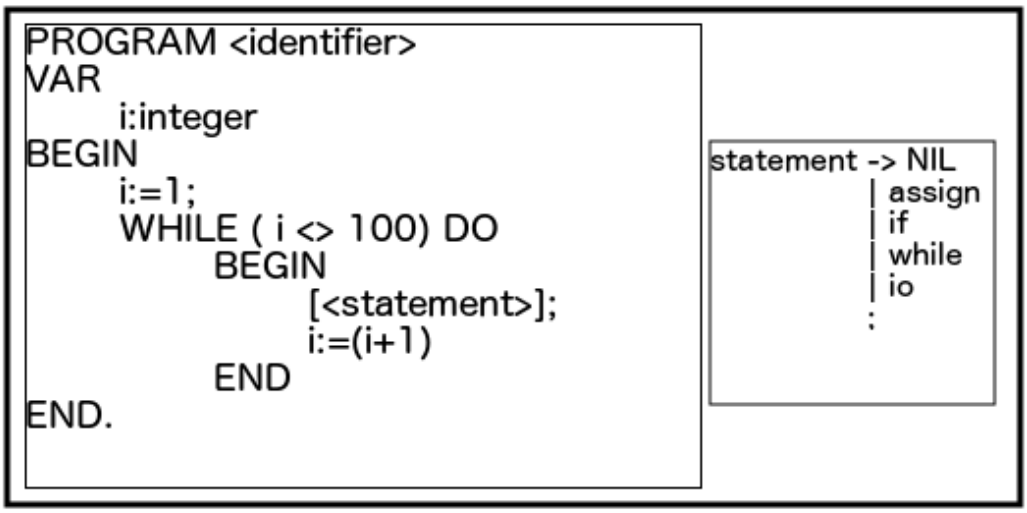


Figure 19.: SDE example

Figure 19 shows the main window of a standard Syntax-Directed Editor. In this figure, two boxes are displayed. The left one is the editor window where we code the program, and the right one exhibits templates choices.

Every <...> tag represents a placeholder, and [...] represents the actual cursor position.

As the cursor changes its position, moving from one placeholder to another placeholder, the right box will be updated according to the grammar rules in the context of the new cursor position. In this example, the cursor in Figure 19 is placed at the placeholder corresponding to a *statement*; at the same time, the right box will be updated with all the possibile templates according to the *statement* derivation rules (RHS).

To sum up, this is how a SDE works.

6.2 CONCEPTION OF THE SDE

7

CONCLUSION

7.1 FUTURE WORK

BIBLIOGRAPHY

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

Henk Alblas. Introduction to attribute grammars. In H. Alblas and B. Melichar, editors, *Int. Summer School on Attribute Grammars, Applications and Systems*, pages 1–15. Springer-Verlag, Jun. 1991. LNCS 545.

F. Arefi, C.E. Hughes, and D.A. Workman. The object-oriented design of a visual syntax-directed editor generator. In *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pages 389 –396, sep 1989. doi: 10.1109/CMPSAC.1989.65112.

Noami Chomsky. Context-free grammars and pushdown storage. RLE Quarterly Progress Report 65, MIT, Apr. 1962.

Daniela da Cruz and Pedro Rangel Henriques. Liss - language of integers, sequences and sets. Talk to the gEPL, Dep. Informática / Univ. Minho, Oct. 2005.

Daniela da Cruz and Pedro Rangel Henriques. Liss – language, compiler & companion. In *Proceedings of the Conference on Compiler Technologies for .Net (CTNET'06 - Universidade da Beira Interior, Portugal)*, Mar. 2006a. (to be published).

Daniela da Cruz and Pedro Rangel Henriques. Liss compiler homepage. http://www.di.uminho.pt/ gepl/LISS, 2006b.

Daniela da Cruz and Pedro Rangel Henriques. LISS — a linguagem e o compilador. Relatório interno do CCTC, Dep.Informática / Univ. do Minho, Jan. 2007a. (to be published).

Daniela da Cruz and Pedro Rangel Henriques. Liss — the language and the compiler. In *Proceedings of the 1.st Conference on Compiler Related Technologies and Applications, CoRTA'07 — Universidade da Beira Interior, Portugal*, Jul 2007b.

P. Deransart and M. Jourdan, editors. *Attribute Grammars and their Applications*, Sep. 1990. INRIA, Springer-Verlag. Lecture Notes in Computer Science, nu. 461.

P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars: Main results, existing systems and bibliography. In *LNCS 341*. Springer-Verlag, 1988.

G. Filè. Theory of attribute grammars. (Dissertation) Onderafdeling der Informatica, Technische Hogeschool Twente, 1983.

M. C. Gaudel. Compilers generation from formal definitions of programming languages: A survey. In *Methods and Tools for Compiler Construction*, pages 225–242. INRIA, Rocquencourt, Dec. 1983.

Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer, New York, Heilderberg, Dordrecht, London, 2nd edition, 2012. ISBN 978-1-4614-4698-9. doi: 10.1007/978-1-4614-4699-6.

Niklas Holsti. Incremental interaction by syntax transformation. In *Compiler Compilers and Incremental Compilation – Proc. of the Workshop, Bautzen*, pages 192–210. Akademie der Wissenschaften der DDR, Institut für Informatik und Rechentechnik, Oct. 1986.

John E. Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 5 – Context-Free Grammars and Languages. Addison-Wesley, 3rd ed. edition, 2006. ISBN 0-321-46225-4.

Uwe Kastens. Attribute grammar as a specification method. In H. Alblas and B. Melichar, editors, *Int. Summer School on Attribute Grammars, Applications and Systems*, pages 16–47. Springer-Verlag, Jun. 1991a. LNCS 545.

Uwe Kastens. Attribute grammars in a compiler construction environment. In H. Alblas and B. Melichar, editors, *Int. Summer School on Attribute Grammars, Applications and Systems*, pages 380–400. Springer-Verlag, Jun. 1991b. LNCS 545.

Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmers text editing. In *CHI '05: HUMAN FACTORS IN COMPUTING*, pages 1557–1560. Press, 2005.

MI-students, Daniela da Cruz, and Pedro Rangel Henriques. Agile - a structured-editor, analyzer, metric-evaluator and transformer for attribute grammars. In Luis S. Barbosa and Miguel P. Correia, editors, *INForum'10 — Simposio de Informatica (CoRTA'10 track)*, pages 197–200, Braga, Portugal, September 2010. Universidade do Minho.

Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4.

Nuno Oliveira, Maria Joao Varanda Pereira, Pedro Rangel Henriques, Daniela da Cruz, and Bastian Cramer. Visuallisa: A visual environment to develop attribute grammars. *ComSIS – Computer Science an Information Systems Journal, Special issue on Advances in Languages, Related Technologies and Applications*, 7(2):266 – 289, May 2010. ISSN ISSN: 1820-0214.

Terence Parr. An introduction to antlr. http://www.cs.usfca.edu/ parrt/-course/652/lectures/antlr.html, Jun. 2005.

Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007. URL http://www.amazon.de/Complete-ANTLR-Reference-Guide-Domain-specific/dp/0978739256.

K. J. Räihä. Bibliography on attribute grammars. *SIGPLAN Notices*, 15(3):35–44, 1980.

Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989a.

Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual*. Texts and Monographs in Computer Science. Springer-Verlag, 1989b.

Thomas Reps, Tim Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 5(3): 449–477, 1983.

Thomas Reps, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language-based editors. *Communications of the ACM*, Sep. 1986.

S.D. Swierstra and H.H. Vogt. Higher order attribute grammars, lecture notes of the Int. Summer School on Attribute Grammars, Applications and Systems. Technical Report RUU-CS-91-14, Dep. of Computer Science / Utrecht Univ., Jun. 1991.

Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9), Sep. 1981.

H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. On the efficient incremental evaluation of Higher Order Attribute Grammars. Research Report RUU-CS-90-36, Dep. of Computer Science / Utrecht Univ., Dec. 1990.

William Waite and Gerhard Goos. *Compiler Construction*. Texts and Monographs in Computer Science. Springer-Verlag, 1984.

A

LISS CONTEXT FREE GRAMMAR

LISS (da Cruz and Henriques, 2007a) is an imperative programming language, defined by the Language Processing members (Pedro Henriques and Leonor Barroca) at UM for teaching purposes. It allows handling integers, sets of integers, dynamic sequences, complex numbers, polynomials, etc., etc (da Cruz and Henriques, 2007b,a, 2006a,b, 2005).

The idea behind the design of LISS language was to create a simplified version of the more usual imperative languages although combining functionalities from various languages.

```
grammar LissGIC;

/* ****** Program ****** */

liss : 'program' identifier body
     ;


body : '{'
       'declarations' declarations
       'statements'   statements
       '}'
     ;

/* ****** Declarations ****** */

declarations : variable_declaration* subprogram_definition*
             ;

/* ****** Variables ****** */

variable_declaration : vars '->' type ';'
                     ;

```

```
25 vars : var (',' var )*
26      ;
27
28 var : identifier value_var
29     ;
30
31 value_var :
32           | '=' inic_var
33           ;
34
35 type : 'integer'
36      | 'boolean'
37      | 'set'
38      | 'sequence'
39      | 'array' 'size' dimension
40      ;
41
42 typeReturnSubProgram : 'integer'
43                      | 'boolean'
44                      ;
45
46 dimension : number (',' number )*
47           ;
48
49 inic_var : constant
50          | array_definition
51          | set_definition
52          | sequence_definition
53          ;
54
55 constant : sign number
56          | 'true'
57          | 'false'
58          ;
59
60 sign :
61      | '+'
62      | '-'
63      ;
64
65 /* ****** Array definition ****** */
66
67 array_definition : '[' array_initialization ']'
```

```
68                  ;
69
70  array_initialization : elem (',' elem)*
71                       ;
72
73  elem : number
74       | array_definition
75       ;
76
77  /* ****** Sequence definition ****** */
78
79  sequence_definition : '<<' sequence_initialization '>>'
80                      ;
81
82  sequence_initialization :
83                          | values
84                          ;
85
86  values : number (',' number )*
87         ;
88
89  /* ****** Set definition ****** */
90
91  set_definition : '{' set_initialization '}'
92                 ;
93
94  set_initialization :
95                     | identifier '|' expression
96                     ;
97
98  /* ****** SubProgram definition ****** */
99
100 subprogram_definition: 'subprogram' identifier '(' formal_args ')'
        return_type f_body
101                    ;
102
103 f_body : '{'
104          'declarations' declarations
105          'statements' statements
106          returnSubPrg
107          '}'
108        ;
109
```

```
110 /* ****** Formal args ****** */
111
112 formal_args :
113             | f_args
114             ;
115
116 f_args  : formal_arg (',' formal_arg )*
117         ;
118
119 formal_arg : identifier '->' type
120            ;
121
122 /* ****** Return type ****** */
123
124 return_type :
125             | '->' typeReturnSubProgram
126             ;
127
128 /* ****** Return ****** */
129
130 returnSubPrg :
131              | 'return' expression ';'
132              ;
133
134 /* ****** Statements ****** */
135
136 statements : statement*
137            ;
138
139 statement : assignment ';'
140           | write_statement ';'
141           | read_statement ';'
142           | conditional_statement
143           | iterative_statement
144           | function_call ';'
145           | succ_or_pred ';'
146           | copy_statement ';'
147           | cat_statement ';'
148           ;
149
150 /* ****** Assignment ****** */
151
152 assignment : designator '=' expression
```

```
153             ;
154
155 /* ****** Designator ****** */
156
157 designator : identifier array_access
158             ;
159
160 array_access :
161               | '[' elem_array ']'
162               ;
163
164 elem_array : single_expression (',' single_expression )*
165             ;
166
167 /* ****** Function call ****** */
168
169 function_call : identifier '(' sub_prg_args ')'
170                ;
171
172 sub_prg_args :
173               | args
174               ;
175
176 args : expression (',' expression )*
177      ;
178
179 /* ****** Expression ****** */
180
181 expression : single_expression ( rel_op single_expression )?
182             ;
183
184 /* ****** Single expression ****** */
185
186 single_expression : term ( add_op term )*
187                   ;
188
189 /* ****** Term ****** */
190 term : factor ( mul_op factor )*
191      ;
192
193 /* ****** Factor ****** */
194
195 factor : inic_var
```

```
196         | designator
197         | '(' expression ')'
198         | '!' factor
199         | function_call
200         | specialFunctions
201         ;
202
203 specialFunctions : tail
204                   | head
205                   | cons
206                   | member
207                   | is_empty
208                   | length
209                   | delete
210                   ;
211
212 /* ****** add_op, mul_op, rel_op ****** */
213
214 add_op : '+'
215        | '-'
216        | '||'
217        | '++'
218        ;
219
220 mul_op : '*'
221        | '/'
222        | '&&'
223        | '**'
224        ;
225
226 rel_op : '=='
227        | '!='
228        | '<'
229        | '>'
230        | '<='
231        | '>='
232        | 'in'
233        ;
234
235 /* ****** Write statement ****** */
236
237 write_statement : write_expr '(' print_what ')'
238                 ;
```

```
239
240 write_expr : 'write'
241           | 'writeln'
242           ;
243
244 print_what :
245           | expression
246           ;
247
248 /* ****** Read statement ****** */
249
250 read_statement : 'input' '(' identifier ')'
251               ;
252
253 /* ****** Conditional & Iterative ****** */
254
255 conditional_statement : if_then_else_stat
256                       ;
257
258 iterative_statement : for_stat
259                     | while_stat
260                     ;
261
262 /* ****** if_then_else_stat ****** */
263
264 if_then_else_stat : 'if' '(' expression ')'
265                     'then' '{' statements '}'
266                     else_expression
267                   ;
268
269 else_expression :
270                 | 'else' '{' statements '}'
271                 ;
272
273 /* ****** for_stat ****** */
274
275 for_stat : 'for' '(' interval ')' step satisfy
276           '{' statements '}'
277         ;
278
279 interval : identifier type_interval
280         ;
281
```

```
type_interval : 'in' range
              | 'inArray' identifier
              ;

range : minimum '..' maximum
      ;

minimum : number
        | identifier
        ;

maximum : number
        | identifier
        ;

step :
     | up_down number
     ;

up_down : 'stepUp'
        | 'stepDown'
        ;

satisfy :
        | 'satisfying' expression
        ;

/* ****** While_Stat ****** */
while_stat : 'while' '(' expression ')'
             '{' statements '}'
           ;

/* ****** Succ_Or_Predd ****** */

succ_or_pred : succ_pred identifier
             ;

succ_pred : 'succ'
          | 'pred'
          ;

/* ****** SequenceOper ****** */

```

```
325 tail // tail : sequence -> sequence
326      : 'tail' '(' expression ')'
327      ;
328
329 head // head : sequence -> integer
330      : 'head' '(' expression ')'
331      ;
332
333 cons // integer x sequence -> sequence
334      : 'cons' '(' expression ',' expression ')'
335      ;
336
337 delete // del : integer x sequence -> sequence
338        : 'del' '(' expression ',' expression ')'
339        ;
340
341 copy_statement // copy_statement : seq x seq -> void
342                : 'copy' '(' identifier ',' identifier ')'
343                ;
344
345 cat_statement //cat_statement : seq x seq -> void
346               : 'cat' '(' identifier ',' identifier ')'
347               ;
348
349 is_empty // is_empty : sequence -> boolean
350          : 'isEmpty' '(' expression ')'
351          ;
352
353 length // length : sequence -> integer
354        : 'length' '(' expression ')'
355        ;
356
357 /* ****** set_oper ****** */
358
359 member // isMember : integer x sequence -> boolean
360        : 'isMember' '(' expression ',' identifier ')'
361        ;
362
363
364
365 /*
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     */
```

```
366
367 string : STR
368         ;
369
370 number : NBR
371         ;
372
373 identifier : ID
374            ;
375 /*
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     */
376
377
378 /* ****** Lexer ****** */
379
380 NBR : ('0'..'9')+
381     ;
382
383 ID : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')* //removi o uso
    do signal '-' conflitos com os valores do signal
384    ;
385
386 WS  :   ( [ \t\r\n] | COMMENT) -> skip
387     ;
388
389 STR :  '"' ( ESC_SEQ | ~('"') )* '"'
390     ;
391
392
393 fragment
394 COMMENT
395     : '/*'.*?'*/' /* multiple comments*/
396     | '//'~('\r' | '\n')* /* single comment*/
397     ;
398
399 fragment
400 ESC_SEQ
401     :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
402     ;
```

lissGIC.g4

Auxiliary results which are not main-stream; or

Details of results whose length would compromise readability of main text; or
Specifications and Code Listings: should this be the case; or
Tooling: Should this be the case.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.