Universidade do Minho
Escola de Engenharia
Departamento de Informática

Damien da Silva Vaz

Implementing an Integrated Syntax Directed Editor for LISS.

December 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

December 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 (particularly, my brother Joël and my father Celestino) 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

It is possible to synthesize a complete source program, ready to be compiled and executed, selecting the appropriate alternative language constructors and writing literals in the right positions in a special editor guided by the source language structure, or syntax.

1.3 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. Then verify the correctness of the assembly code generated with a simple MIPS simulator, named MARS, that 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 with Java (JAVAFX) 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. At this phase, we will create an IDE similar to other platforms but with the capacity of being a syntax-directed editor.

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

2

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.

Before explaining the basic statements of the language and its data types using a context free grammar, let's remember briefly the basilar concepts related to formal programming languages and their definition using grammars (context free and attribute grammars).

2.1 FORMAL LANGUAGES AND GRAMMAR

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.

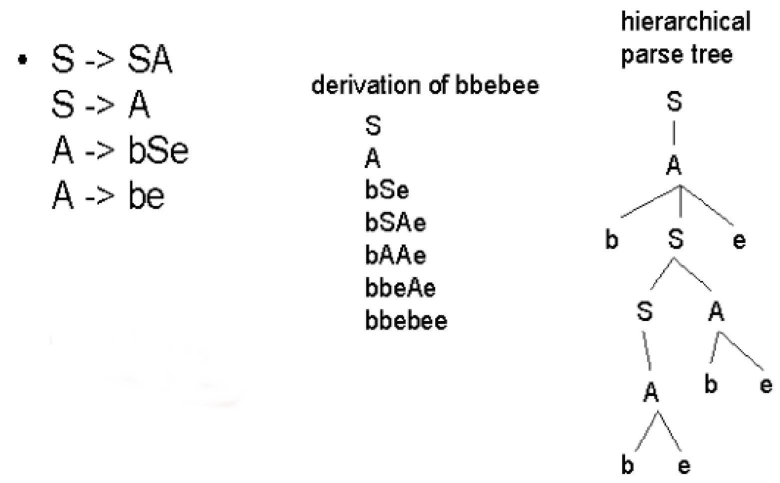


Figure 1.: CFG example [1]

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.

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.

2.2 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:

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

Listing 2.1: Declaring a variable in LISS

As we can see in Listing 2.1, 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 2.2).

```
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
```

Table 1.: LISS data types

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

```
          | 'true'
          | 'false'
          ;
sign :
     | '+'
     | '-'
     ;
```

Listing 2.2: CFG for declaring a variable in LISS

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

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

Additionally, we may change the default values of the variables by initializing them with a different value (see an example in Listing 2.3). 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 2.2).

```
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 2.3: Initialize a variable

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

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
```

```
    | array_definition
    ;
```

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:

```
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 :

```
set_definition : '{' set_initialization '}'
               ;

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

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 2.4: Example of valid operations using sequence on LISS

In the example of Listing 2.4 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)*
       ;
```

2.2.1 *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 2.5).

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

Listing 2.5: 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 2.6.

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

Listing 2.6: 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"
```

2.3 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 2.7).

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

Listing 2.7: 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 2.8).

```
liss : 'program' identifier body
   ;

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

Listing 2.8: CFG for program in LISS

2.3.1 *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 2.2; the subprogram part will be discussed later in section 2.4.

This part is specified by the following grammar (see Listing 2.9).

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

Listing 2.9: CFG for declarations in LISS

2.3.2 *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 2.10.

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

Listing 2.10: CFG for statements in LISS

Let's see one example of a LISS program which shows how the language shall be used (see Listing 2.11).

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

```
        for(j in 1..i){
            res=res*j;
        }
        writeln(res);
}
```

Listing 2.11: 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 2.12 an example.

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

Listing 2.12: Example of assignment in LISS

In Listing 2.12, 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 2.13.

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

Listing 2.13: 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).

```
program input1{
```

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

Listing 2.14: Example of input operation in LISS

Notice that, in Listing 2.14, 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 2.15.

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

Listing 2.15: 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 2.16 shows some more examples.

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

Listing 2.16: 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 2.17).

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

write_expr : 'write'
           | 'writeln'
           ;

print_what :
           | expression
           ;
```

Listing 2.17: 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 2.3.1). This will allow reusing functions that were created by calling them instead of creating duplicated code.

See Listing 2.18 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
```

```

    a = calculate();
    writeln(a);
    writeln(d);
}
```

Listing 2.18: Example of call function in LISS

In Listing 2.18, 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 2.19.

```
function_call : identifier '(' sub_prg_args ')'
              ;
sub_prg_args :
             | args
             ;
args : expression (',' expression )*
     ;
```

Listing 2.19: CFG for call function in LISS

2.3.3 *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 2.20).

The second one is called cyclic statement or iterative statement, and it has two variants (see Listing 2.20).

```
conditional_statement : if_then_else_stat
                      ;
iterative_statement : for_stat
                    | while_stat
                    ;
```

Listing 2.20: 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 2.21.

```
if(y==x)
then{
  x=x+1;
}else{
  x=x+2;
}
```

Listing 2.21: LISS syntax of a if statement

The code shown in Listing 2.21, 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 2.22.

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

else_expression :
                | 'else' '{' statements '}'
                ;
```

Listing 2.22: 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 2.23: LISS syntax of a for-loop statement

In Listing 2.23, 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 2.24: LISS syntax of a for-each statement on array

In Listing 2.24, 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 2.25: 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 2.26.

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

Listing 2.26: LISS syntax of a while-statement in LISS

In Listing 2.26, 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 2.27: CFG for while-statement in LISS

2.3.4 *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 2.28 illustrates both statements.

```
succ int1;
pred int1;
```

Listing 2.28: Example of using succ/pred in LISS

As we can see in Listing 2.28, 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 2.29.

```
succ_or_pred : succ_pred identifier
             ;
succ_pred : 'succ'
          | 'pred'
          ;
```

Listing 2.29: 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 2.30.

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

Listing 2.30: 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 2.30, the statement 'copy' copies the content of the variable *seq1* to *seq2*.

The grammar for 'copy' statement is in Listing 2.31.

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

Listing 2.31: 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 2.32).

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

Listing 2.32: Example of cat statement in LISS

In Listing 2.32, '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 2.33.

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

Listing 2.33: CFG for cat statement in LISS

2.4 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 2.34.

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

```
              ;
f_args   : formal_arg (',' formal_arg )*
        ;
formal_arg : identifier '->' type
             ;
return_type :
              | '->' typeReturnSubProgram
              ;
returnSubPrg :
               | 'return' expression ';'
               ;
```

Listing 2.34: 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 2.34), 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.

2.5 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 2.35).

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

Listing 2.35: 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 2.36).

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

Listing 2.36: 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 2.37). It does not make sense to have an expression like that : '3 == 4 == 5 != 6'.

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

Listing 2.37: Rule for Boolean expression

We added the possibility of using parenthesis on expressions (see Listing 2.38).

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

Listing 2.38: 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 2.39).

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

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

Listing 2.39: 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 2.40).

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

Listing 2.40: 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 2.41: Lexical rule for Comment

3

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 primarily 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`

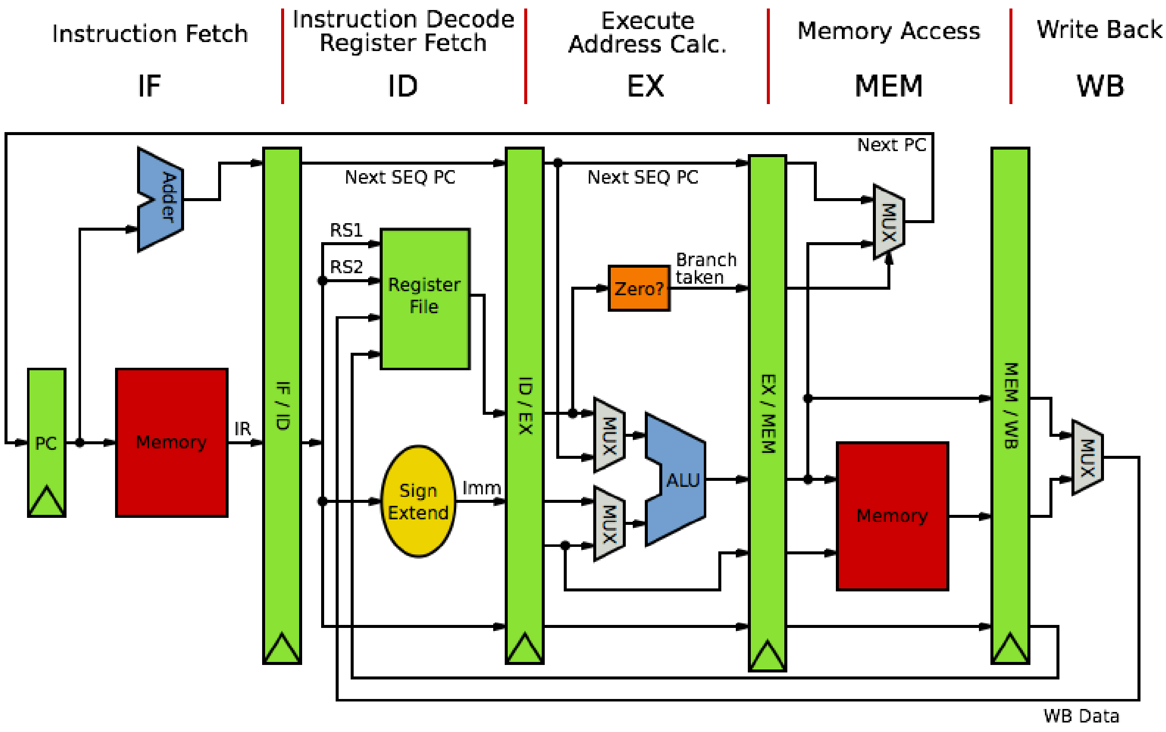


Figure 2.: MIPS architecture

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

3.1 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.

3.2 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)

3.3 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.

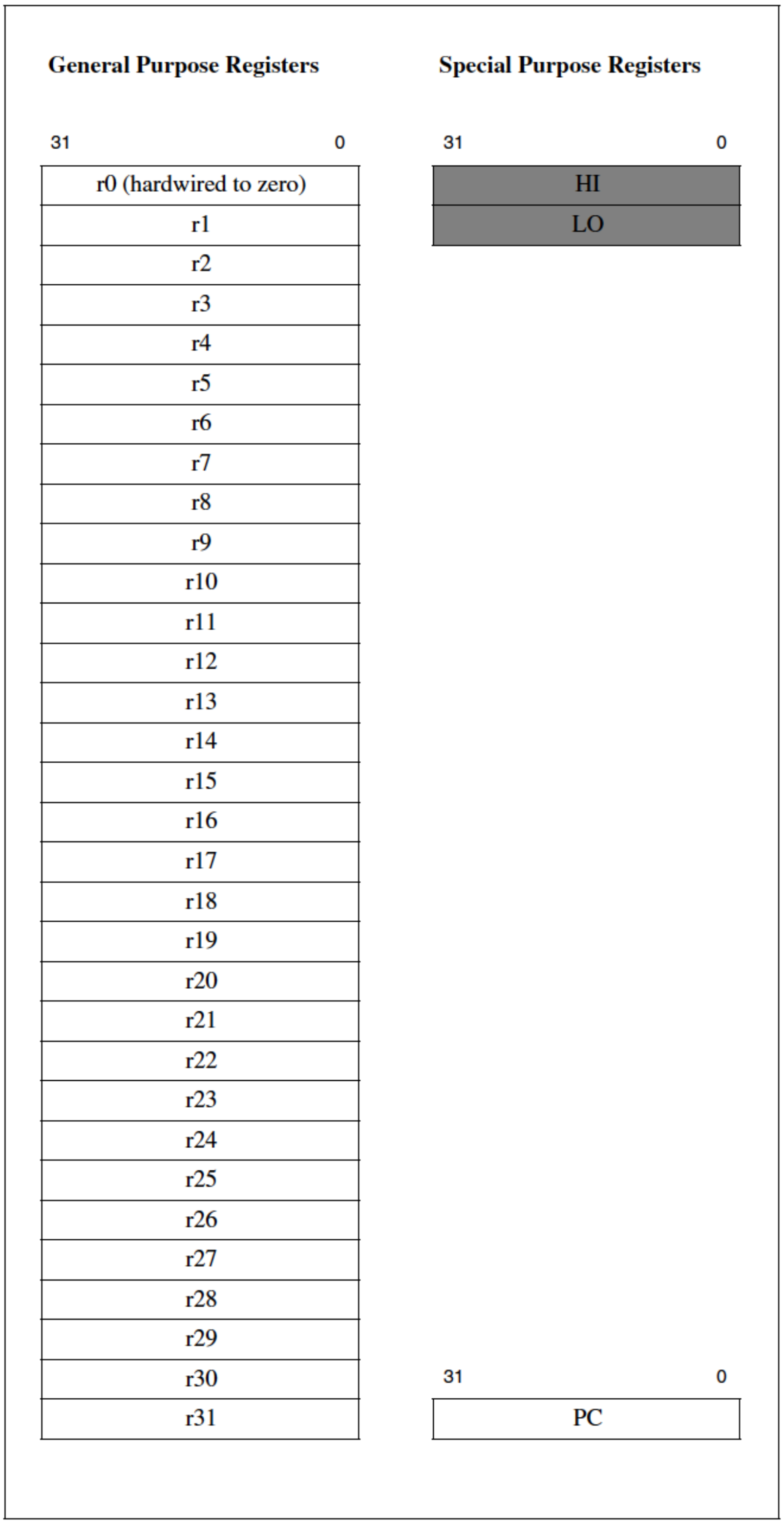


Figure 3.: MIPS register

3.4 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.

3.4.1 *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 3.1).

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

Listing 3.1: R-Type instruction format

In Listing 3.1, 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:

```
rd = rs OP rt
```

To understand better this instruction, let's see an example of one R-Type instruction in MIPS (see Listing 3.2).

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

Listing 3.2: Example of a R-Type instruction

The instruction shown in Listing 3.2 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.

$$OP\ rd,\ rs,\ rt \iff rd\ =\ rs\ OP\ rt$$
$$\Downarrow$$
$$add\ \$t1,\ \$t1,\ \$t2 \iff \$t1\ =\ \$t1\ add\ \$t2$$
$$\Downarrow$$
$$\$t1\ =\ \$t1\ +\ \$t2$$

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.

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

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 $t0, $t0, $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).

3.4.2 *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 $t0, $t0, 10 // Arithmetic operation
ori $t0, $t1, 5   // Logical operation
beq $t0, $t1, 1    // Conditional branch operation
lw $t0, array1($t0) //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.

3.4.3 *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 3.3.

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

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

Listing 3.3: Examples of J-Type instruction

In Listing 3.3, 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*.

3.5 MIPS ASSEMBLY LANGUAGE

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

3.5.1 *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 3.4.

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

Listing 3.4: Syntax format of data declarations in MIPS

In Listing 3.4, 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 3.5.

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

Listing 3.5: Examples for declaring variables in MIPS

In Listing 3.5, 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.

3.5.2 *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 3.6.

```
.text
  main:
    li $t0, 5
    li $t1, 10
    mul $t0, $t0, $t1
```

Listing 3.6: Example of Text declarations in MIPS

In Listing 3.6, 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 3.6 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 3.7.

```
.data
.text
  main:
     li $t0, 5
     li $t1, 5
    mul $t0, $t0, $t1
     jal jump_condition #needs to jump to the field jump_condition
     li $t0, 4
     li $v0, 10
     syscall
  jump_condition:  #syntax for jump and conditional instruction in mips
     li $t1, 5
     jr $ra
```

Listing 3.7: Example of a loop declaration in MIPS

As we can see in Listing 3.7, 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 3.8).

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

Listing 3.8: Example of a comment in MIPS

Let's see the template for a MIPS assembly language program in Listing 3.9.

```
# 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 3.9: Template of a MIPS assembly language

3.6 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 $t0, $t1	$t0 = $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 3.4.

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 $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains address of allocated memory
exit (terminate execution)	10		

be set into register $v0 (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 3.10.

```
li $t0, 3              #adding the number 3 to register t0
li  $v0, 1             # loading the service number 1 (print integer) to register v0
add $a0, $t0, $zero  # loading the argument value to register a0
syscall  #calling the syscall for printing the integer.
```

Listing 3.10: Example of printing integer in MIPS

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

3.7 MIPS MEMORY MANAGEMENT

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

1. stack
2. heap

3.7.1 *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 3.11.

```
push:  addi $sp, $sp, −4  # Decrement stack pointer by 4
        sw   $v0, 0($sp)   # Save register v0 to stack

pop:  lw   $v0, 0($sp)   # Copy from stack to register v0
       addi $sp, $sp, 4   # Increment stack pointer by 4
```

Listing 3.11: Example of push and pop instructions in MIPS

3.7.2 *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 3.12.

```
.text
 main:
   li $a0, 4 #we want to allocate 4 bytes in the heap.
   li $v0, 9 # we load the value 9 in register v0 for calling the heap
       instruction.
   syscall   # calling the system call instruction for allocating 4
       bytes into the heap. The register v0 contains the address of
       allocated memory.
```

Listing 3.12: Example of code for allocating in the heap

3.8 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.

3.8.1 *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.

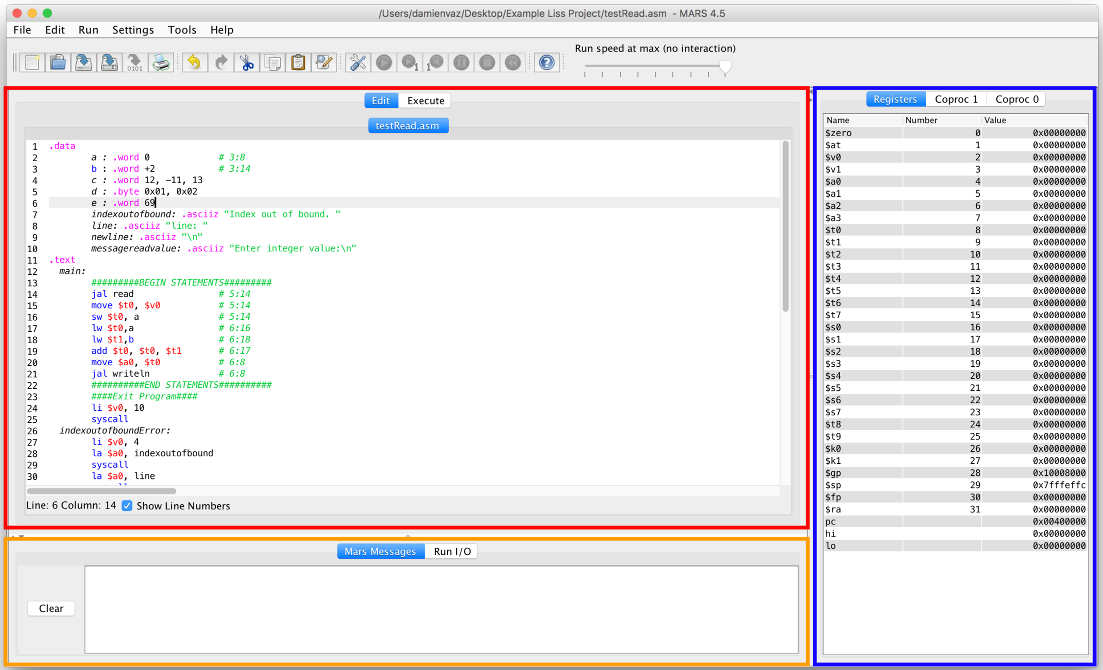


Figure 4.: MARS GUI

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.

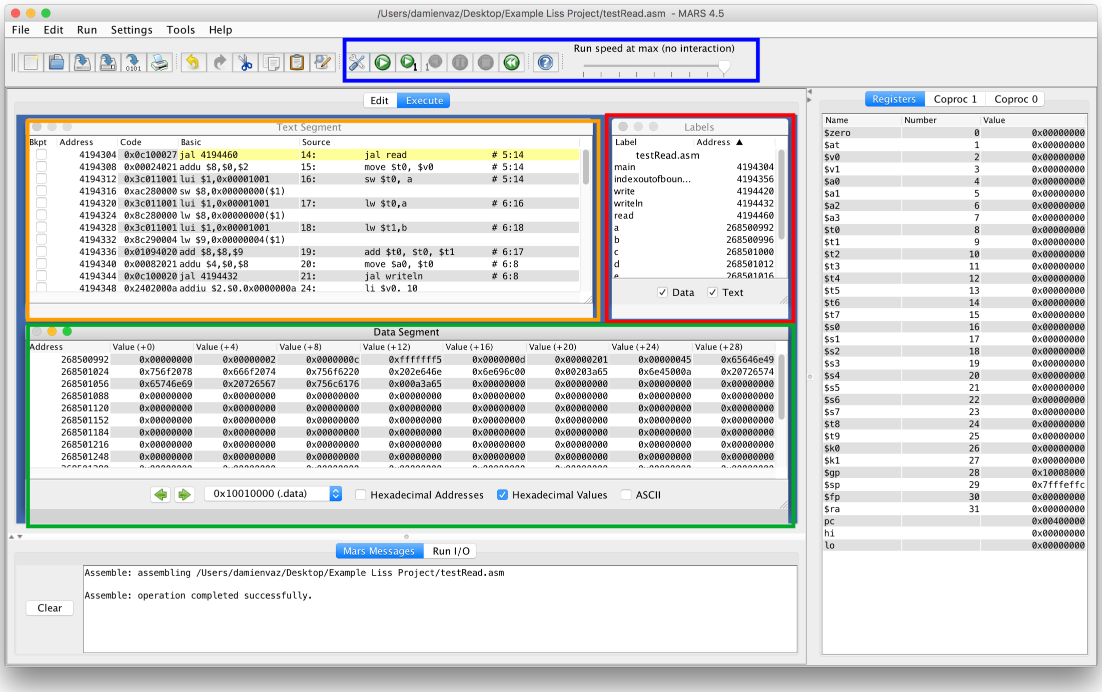


Figure 5.: MARS GUI (Execution mode)

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 certain 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.

4

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-

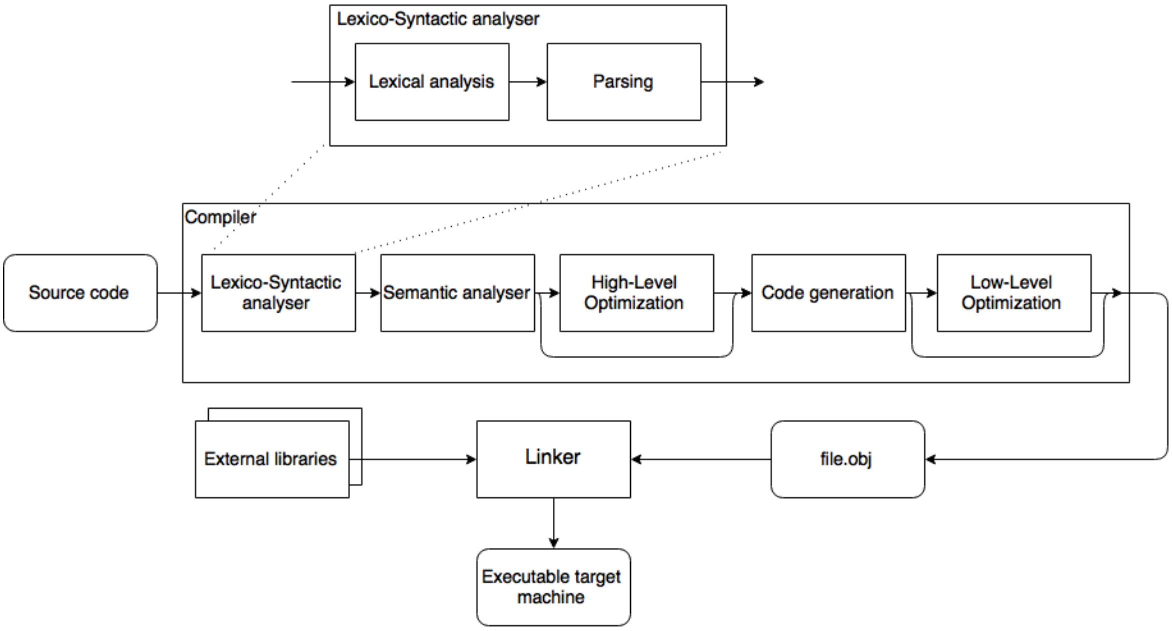


Figure 6.: Traditional compiler

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 4.1.

The first tool steps, lexical and syntatical analysis, will be briefly discussed in section 4.2. Then section 4.3 explains in detail the implementation of LISS semantic analyzer. To conclude the chapter, section 4.4 provides also details about the implementation of the LISS code generator.

4.1 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 4.1 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 4.1: 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 4.2. 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 4.2: Lexer representation

4.2 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.

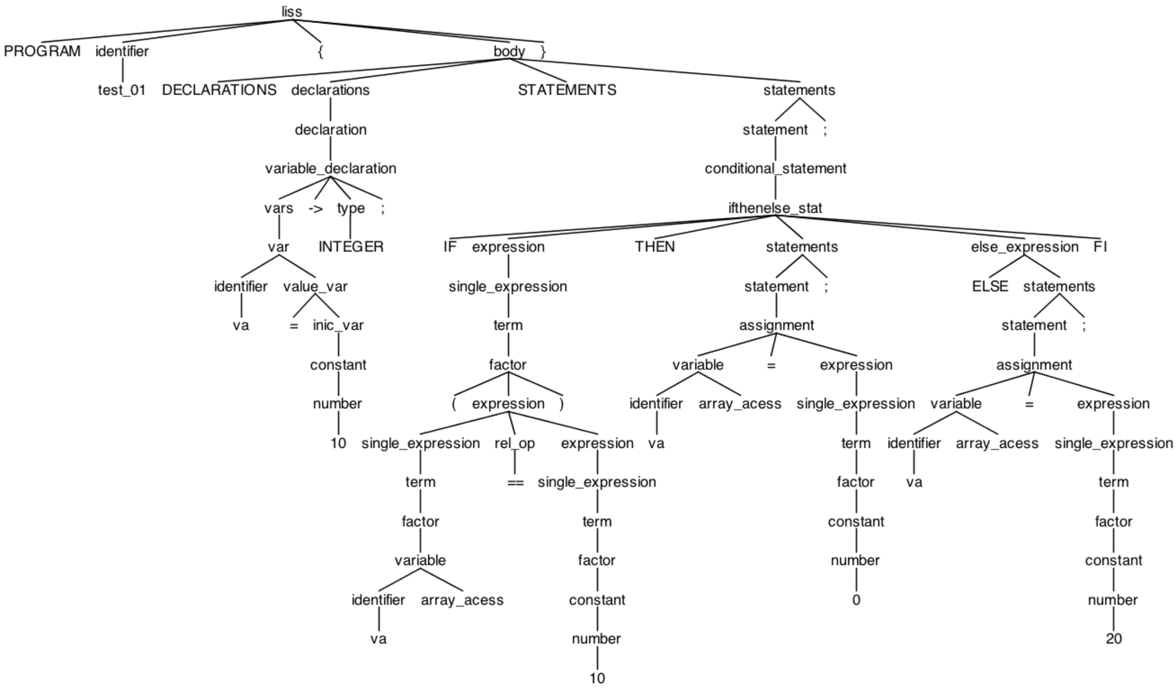


Figure 7.: AST representation

4.3 SEMANTIC ANALYSIS

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 use some structures that helps the compiler for getting those inconsistencies and also informs the user about those inconsistencies.

And the kind of inconsistency that can be found in the project are listed above:

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.

Now, let's talk about the structures that were made for the project.

4.3.1 *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.

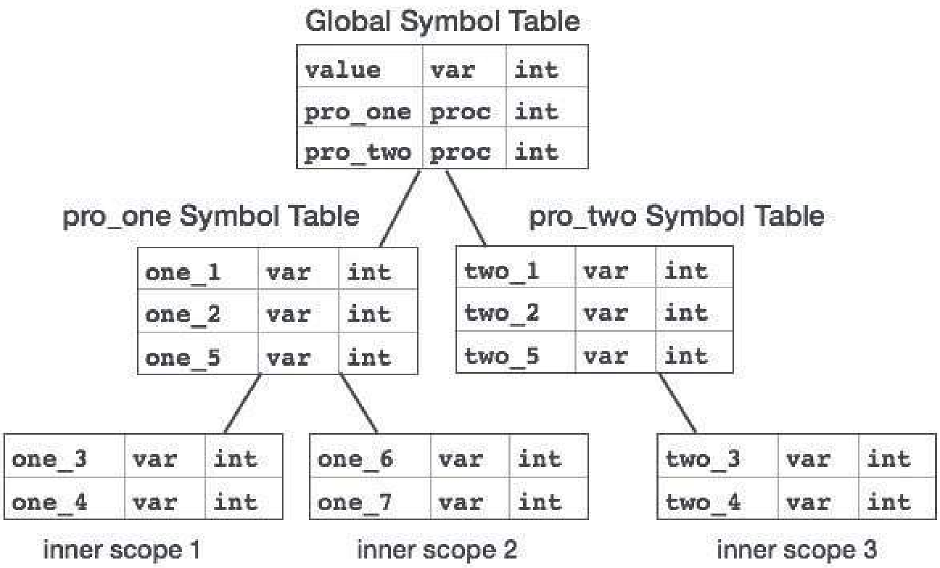


Figure 8.: Example of hierarchical symbol table

Symbol Table in LISS

For this project, we used only one symbol table (ST) for all symbols.

Let's explain throw Figure 9, the usage of the symbol table in LISS.

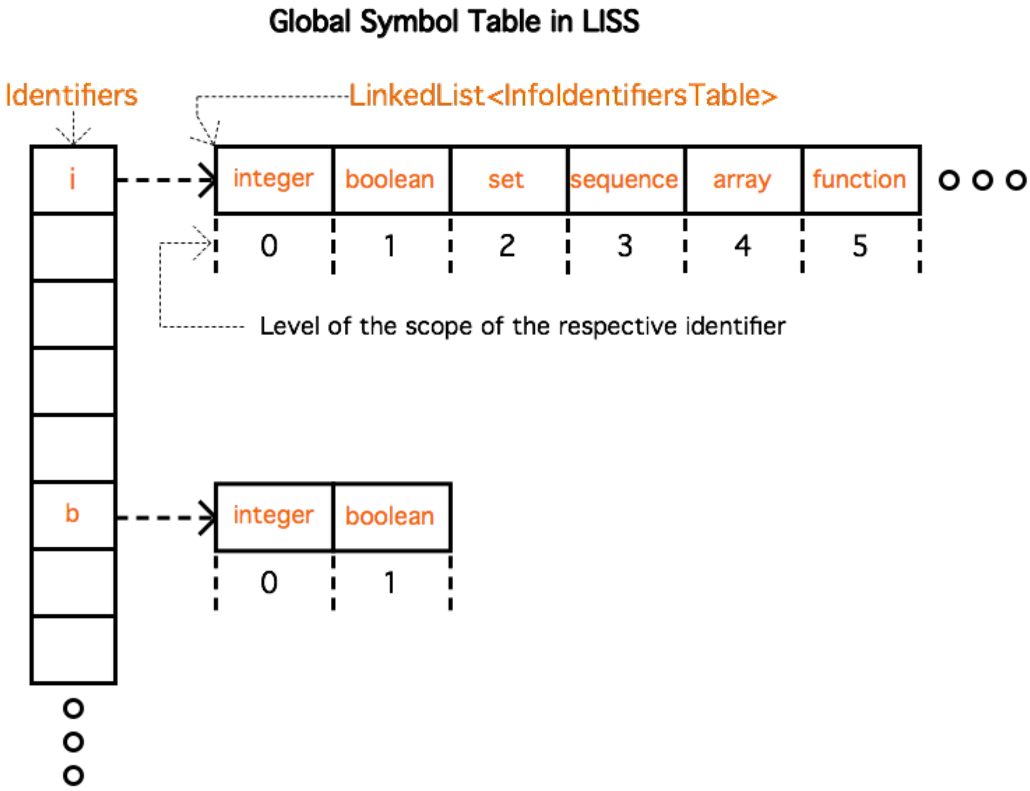


Figure 9.: Global symbol table in LISS

For our project we implemented the symbol table with a HashMap where the key is an identifier and the value is a LinkedList of variables information associated with the identifier.

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

Listing 4.3: Data structure of the symbol table in LISS

The identifier (of type *String*, as shown in Listing 4.3) must be unique (concept of using a HashMap) and the *LinkedList* must be an ordered list.

Basically, the identifier is associated to a *LinkedList* of information related to the identifier that explains among other things the category and type of the identifier.

For our project, we have 3 different categories:

1. TYPE

2. VAR

3. FUNCTION

TYPE category

The *TYPE* category aggregates all the identifiers that denote the primitive (or pre-defined) types available in LISS. In our language, they are: set, integer, sequence and boolean. In this case, the ST contains information about the fixed size of each type in MIPS representation, as well as their scope level as summarized in Table 21.

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 aggregates all the identifiers that denote the variables declared in LISS (in the program *declarations* part). The variable type may be integer, boolean, array, set or sequence. The information associated with each variable depends on its type.

Let's see and explain in detail the information per type.

Table 22.: ST information for an integer variable

Identifier	Category	Level	Type	Address
x	VAR	0	integer	0

In Table 22, we can see the information stored in ST for an integer variable:

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

Table 23.: ST information for a boolean variable

Identifier	Category	Level	Type	Address
bool	VAR	1	boolean	4

In Table 23, it can be seen the information stored in ST for a boolean variable:

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

Table 24.: ST information for an array variable

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

In Table 24, it can be seen information stored in ST for an array.

- *Identifier* - name of the variable.
- *Category* - the category of the identifier: VAR.
- *Level* - the scope level 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.: ST information for a set variable

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

In Table 25, it can be seen the information stored in ST for a set.

- *Identifier* - name of the variable.
- *Category* - the category of the identifier: VAR.

- *Level* - the scope level of the variable.
- *Type* - the type of the variable (set).
- *Address* - not used (no memory address).
- *Tree Allocated* - indicates if the set has an initiate value associated. Letter 'X' means that the set was initialized.

Table 26.: ST information for a sequence variable

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

In Table 26, it can be seen the information stored in ST for a set.

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

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

Table 27.: ST information for a function

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

In Table 27, it can be seen the information stored in ST for a function.

- *Identifier* - name of the function.
- *Category* - the category of the identifier: FUNCTION.
- *Level* - the scope level of the function.
- *Type* - field not used.

- *Address* - size of the function stack in the stack memory (includes the arguments list, the variables declared in the subprogram and the return value).
- *Nº Arguments* - indicates how many arguments the function does have.
- *Type List Arguments* - indicates the type of each argument.

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

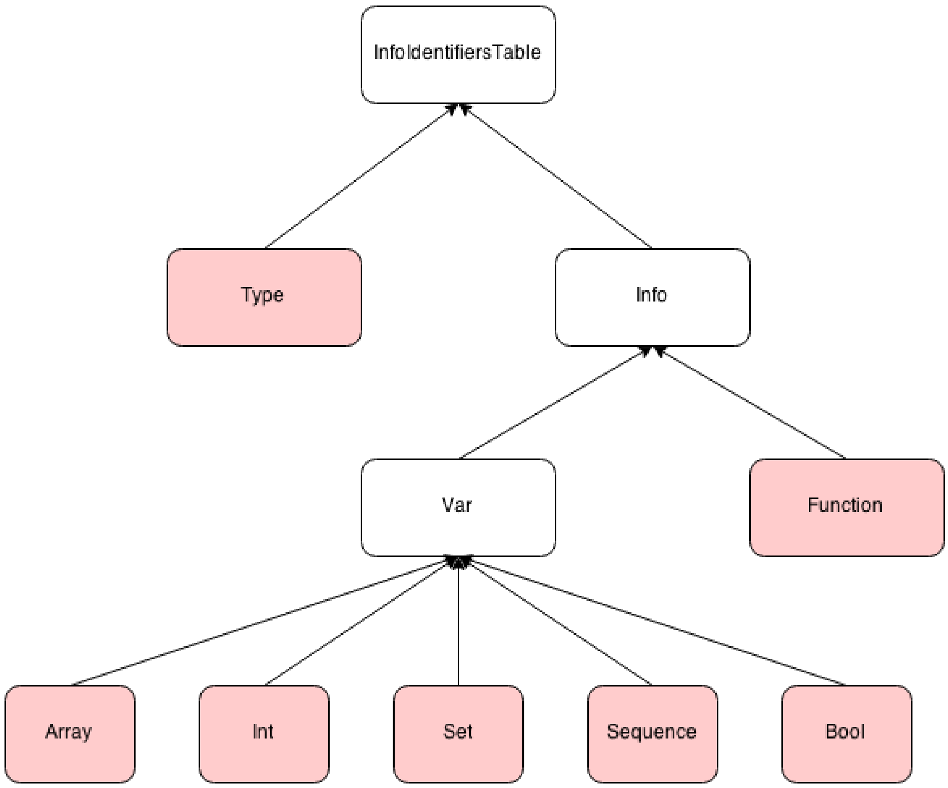


Figure 10.: InfoIdentifiersTable structure

Each time that an identifier is inserted into the HashMap, the information described related to that identifier is inserted according to its category.

The usage of a *LinkedList<InfoIdentifiersTable>* has the notion of being an ordered list, and this is very important due to the fact that it reveals the level of a given identifier found. In Figure 9, the identifier **b** was found in two differents scope level.

- Level 0 - Identifier **b** found with type *integer*
- Level 1 - Identifier **b** found with type *boolean*

Notice that every time, we look for an identifier and its respective info in the symbol table, the most recent (the one that was the latest inserted) info in the *LinkedList* will be inserted.

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

Every time that a function (*subprogram* in LISS) is exited, we remove every information associated with the scope level of the function from the symbol table.

The JAVA functions created and available in the project, regarding the symbol table handling.

- getSymbolTable - gets the symbol table.
- doesExist - checks whether a certain identifier is in ST.
- getInfoIdentifier - gets the most recent information associated with a certain identifier.
- removeLevel - removes from the symbol table every information related to a given scope level.
- getAddress - gets the most recent address (this address is related to the next position of an identifier that will be added to the symbol table).
- setAddress - sets a new address.
- add - adds an identifier into the symbol table.
- toString - gets the representation of the symbol table as a string.

4.3.2 *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 (write) a LISS program. In this way, it will facilitate the user to fix the errors found in his code.

Figure 11 shows the structure of the error table built for our project.

We managed to create a data structure which can handle some error messages and also store some information related to the error message (line and column number).

This was done by creating the following data structure in JAVA:

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

Listing 4.4: Data structure of the error table in LISS

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

We chose 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 a pair <key, value> in the *TreeMap* data structure, the information is ordered by the unique key. In this case, the first

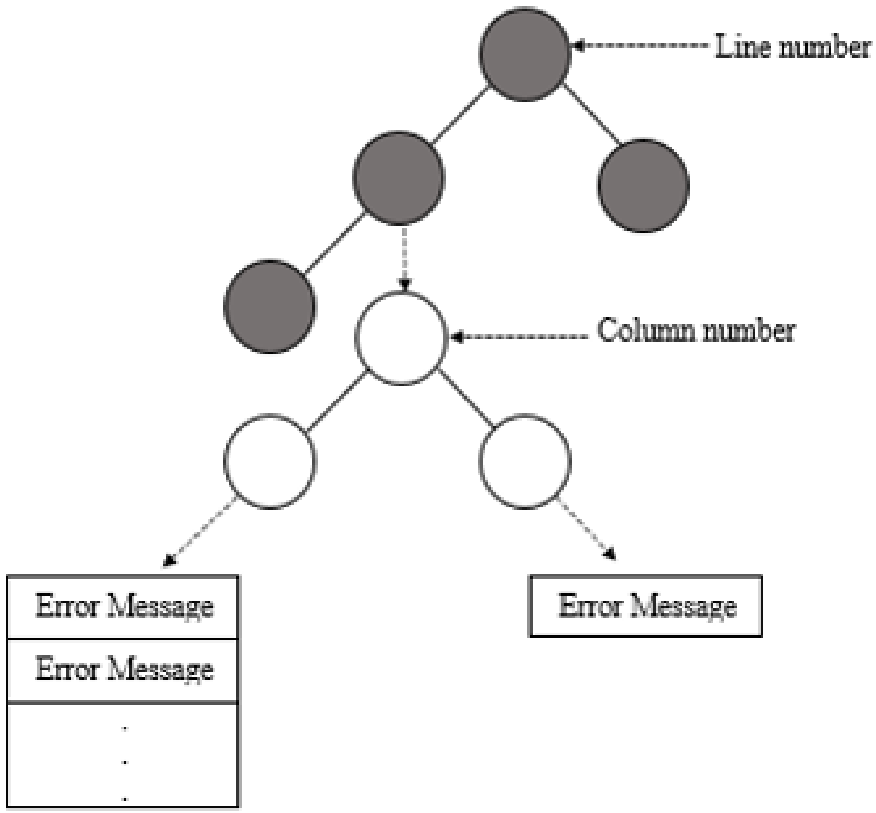


Figure 11.: ErrorTable structure

TreeMap is intended 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 easy reading.

Listing 4.5 shows some error message issued after processing a LISS program.

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

```
    line: 10:4  Expression 'seq1' already exists.
    line: 14:4  Expression 'b' already exists.
```

Listing 4.5: Example of an error table

Regarding the data structure explained above, Figure 12 shows an example of how the error messages are being handled, displaying the storage of the first two messages in Listing 4.5.

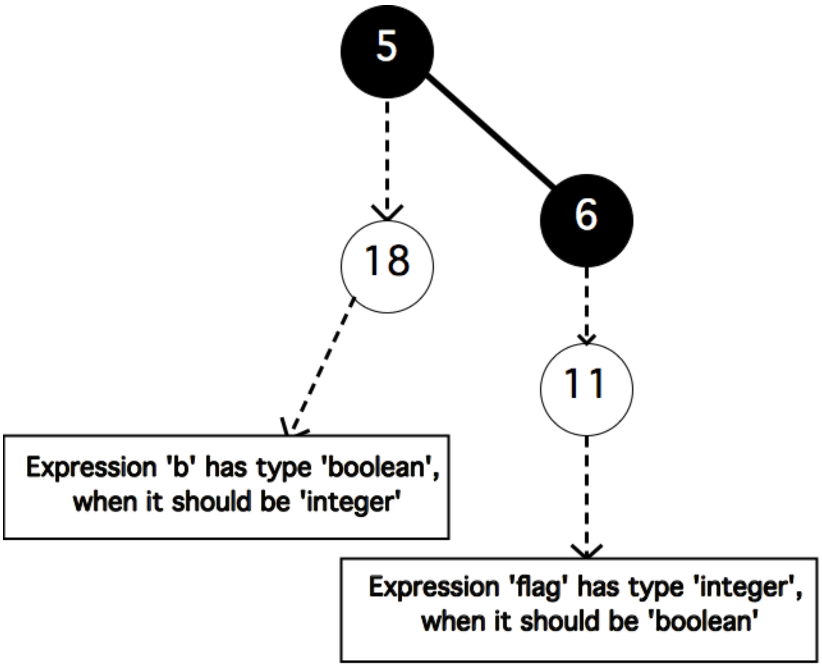


Figure 12.: ErrorTable structure instantiated for example in Listing 4.5

4.3.3 *Types of error message*

As said, previously, we have different kinds of error messages, that can be seen in Table 28.

Notice that in Table 28, we have all the messages used and thrown (when necessary) by the compiler that we need to explain the notation used. For example, the mark:

```
<...>
```

represents a placeholder; it means that it must be replaced by the correct name according to the environment where the error was found.

To understand better the usage of the mark, let's consider the program example in Listing 4.6.

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.

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

Listing 4.6: Partial Listing

The LISS program in Listing 4.6 declares two variables with the same name in the declarations part. As in all programming language, in LISS it is not allowed to declare two variables in the same scope level. So the compiler must throw an error, printing the message in Listing 4.7.

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

Listing 4.7: Error table related to Listing 4.6

The error message in Listing 4.7 has a type with the number 2 message in Table 28. We can see that the mark was replaced by the name of the variable. The same exactly happens regarding the other messages in the Table 28.

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

1. Message for a variable not declared.
2. Message for a variable already declared.
3. Message for a variable that must be an array and is not.
4. Message for a variable that is of type array and its limits that doesn't match.
5. Message for a variable that has a certain type, but should have another type.
6. Message regarding an assignment found with different types. For example : 'integer' = 'boolean'.
7. Message for an expression that contains operand with different types.
8. Message for a function when the return type must be different.
9. Message for a function where 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 an array that has a different number of dimensions, according to its declaration.
12. Message for an unconditional loop that use 'stepUp' or 'stepDown' expression.
13. Message for an unconditional loop that use 'satisfying' expression.
14. Message for a function when its name do not exist.
15. Message for an array where the limits or the dimension do not agree with it.

4.3.4 *Validations Implemented*

In this section, we are going to report where the error message will be thrown by the compiler according to contextual conditions expressed in the attribute grammar that we have.

Variable declaration

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

Listing 4.8: Variable declaration rule in LISS

Listing 4.8, shows the declaration part where the programmer declares variables with the respective type. Processing this part, the compiler adds the information into the symbol table. As variable can be initialized at this stage, before adding the information into the symbol table, we need to check if every variable has the correct type regarding the type of literal value assigned. If not, error message 5 (see Table 28) must be printed.

Let's see in Listing 4.9, an error that may happen in this case.

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

Listing 4.9: Example of an error message in variable declaration

While processing this section, the compiler creates the mips code for each variable. 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 index respects the limits (the index value is in bounds); if not then error message number 4 must be issued (see Table 28).

Regarding to the other types, we don't check values in this section. Notice that the *array* type 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 4.10.

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

Listing 4.10: Example of an error message in variable declaration for the array type

In the left side of a declaration, before the type name the programmer can define one or more variables. Listing 4.11 shows the grammar rule for *Vars*.

Vars

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

Listing 4.11: Vars rule in LISS

The grammar rule in Listing 4.11 refers to the declaration of multiple variables of the same type.

In this case, the compiler needs to check if the variables that will be added to the symbol table have been already declared. If one variable has been previously declared, then an error message type number 2 will be thrown (see Table 28). Let's see an example of a LISS program that illustrates this error in Listing 4.12.

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

Listing 4.12: Example of an error message in LISS for vars non-terminal

In Listing 4.12, 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_definition : '{' set_initialization '}'
               ;
set_initialization :
                   | identifier '|' expression
```

Listing 4.13: Set initialization rule in LISS

In Listing 4.13, we can see how to declare a set under the declarations section in LISS. And it has two choices for declaring a set: empty set or some content in the set. If there is some content available, this content must be defined by a boolean expression. In case the expression is not a boolean, an error message with the number 7 (see Table 28) will be thrown. Let's see an example of this error with a piece of LISS code related to set initialization in Listing 4.14.

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

Listing 4.14: Example of an error in LISS for set_initialization

In Listing 4.14, 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.

Moreover, there is another error concerned with the add operator that defines both operands of type integer. But, due to the previous statements made, the types that the add operator will see: 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, an error message will be printed out, as can be seen in Listing 4.15.

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

Listing 4.15: Error message for the set_initialization

Subprogram definition

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

Listing 4.16: Subprogram definition rule in LISS

In Listing 4.16, we can see how to declare subprogram (function) under the declaration section in LISS. In this case, we need to check the return type of the subprogram.

If the type of the returned expression is different from the type declared (for example, the return expression is of type boolean, but the return type declared is integer) then an error message number 7 (see Table 28) is thrown.

Let's see an example of this case in Listing 4.17.

```
subprogram f(amen->boolean)->integer{
        declarations
                b -> boolean;
        statements
        return b;
}
```

Listing 4.17: Example of error message in LISS for subprogram_definition non_terminal

In Listing 4.17, we can see that the return type declared is integer. However in this example, the subprogram returns a variable called 'b' that has boolean type. In this case, an error message will be reported due to the incompatible types.

Assignment

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

Listing 4.18: Assignment rule in LISS

In Listing 4.18, we can see the syntax for assigning some content to a variable or an array under the statement section in LISS. For this part we need to check some possible error regarding to the context available. Let's explain those differents contexts below.

Suppose the designator non-terminal in Listing 4.18, is a variable of type *array*; In that case, the content that we can assign to that variable is restricted (see an example in Listing 4.19).

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

Listing 4.19: Example of assigning a constant value to an array variable

In Listing 4.19, 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 value has the correct dimensions and limits regarding the variable declaration. In this case, if it isn't correct then the number 15 error message will be outputed (see Table 28).

Also the same error message is reported for the case of the next example in Listing 4.20.

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

Listing 4.20: Example of storing a value to a certain position in the array

In Listing 4.20, suppose that the variable *array* has one dimension with only two position available. In this case, the access to the fourth position (index 3, as shown in the example), is behind the limits regarding to the specification of the variable. And, in this case, it must thrown the number 15 error message too (see Table 28).

The last case for this part concerns in general the types of *designator* and *expression* that must be equal. The compiler can't generate code for the assignment operation.

If they do not conform this general rules, then this will throw the number 6 error message (see Table 28).

Let's see an example of this case in Listing 4.21.

```
boolean1 = integer1;
```

Listing 4.21: Example of assignment with differents types

In Listing 4.21, the example shows us that assignment operation is trying to store an integer value in *boolean1* variable (which has boolean type). As we know, if the types aren't equal then the operations cannot be executed and an error must be reported.

Designator

```
designator : identifier array_access
```

Listing 4.22: Designator rule in LISS

In Listing 4.22, we can see the syntax to refer to an atomic variable or an array variable under the statement section in LISS. In this case, we need to check some errors depending on the context: atomic variable or array variable.

Let's explain first the case of a simple variable.

Every variable used must be declared and so it must be 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 number 1 (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 an error message number 1 must throw (see Table 28).

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

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

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

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

And finally, we need to check if the *identifier* and the *array_access* agreed in number of dimensions. If not, then it must be thrown the number 11 error message (see Table 28).

Elem array

```
elem_array : single_expression ( ',' single_expression )*
```

Listing 4.23: Elem_array rule in LISS

In Listing 4.23, we can see how to handle an element of an array. For this part we need to check if every element (represented by the *single_expression* non-terminal) has the correct type.

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

Function call

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

Listing 4.24: Function_call rule in LISS

In Listing 4.24, we can see the syntax to call a function under the statement section in LISS. In this context, we need to check two error situations.

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

Lastly, the compiler checks if the *identifier* has the correct category (it must belong to the category *Function*). If it doesn't have then it must throw the number 9 error message (see Table 28).

Expression

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

Listing 4.25: Expression rule in LISS

Listing 4.25, defines the syntax to write an expression, that can be used in many different contexts in LISS. In this case, we need to check the type of both operands (the *single_expression* are correct regarding the type required by *rel_op*). If they are not, then we throw the number 10 error message (see Table 28).

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

Let's see an example in Listing 4.26.

```
2 < true
```

Listing 4.26: Example of an error message in expression rule

In Listing 4.26, we can see the number two (left single_expression) then the less-than sign (the rel_op) and finally the true value (the right single_expression annotation). We can see immediately that the operand types doesn't match. In that case the less-than sign requires both expressions (left and right) of type integer, and actually one is integer and the other is boolean — an error number 10 will be generated.

Single expression

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

Listing 4.27: Single_expresion rule in LISS

Listing 4.27 defines how to expand a *single_expression* non-terminal in LISS. In a similar way, we need to check the types of the operands required by the *add_op*. If the *terms* type don't agree with the required type regarding to the *add_op*, then the compiler must throw the number 10 error message.

Term

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

Listing 4.28: Term rule in LISS

In Listing 4.28 is specified the syntax to expand a *term* in LISS. In a similar way, we need to check the types of the operands required by the *mul_op*. If the *factors* type don't agree with the required type regarding *mul_op*, then it must throw the number 10 error message.

Factor

```
factor : inic_var
       | designator
       | '(' expression ')'
       | '!' factor
       | function_call
       | specialFunctions
```

Listing 4.29: Factor rule in LISS

In Listing 4.29, we can see the syntax to expand a *factor* in LISS. As it can be seen, there are a lot of alternative rules; however the majority do not require any special check.

We will only discuss a particular one:

```
'!' factor
```

In this option there is an exclamation mark sign and then a *factor* non-terminal. In programming languages, the exclamation mark represents the negation of the expression. So, the negation operation requires an operand of boolean type in order to work correctly. If the type of the *factor* is not a boolean, then the number 7 error message will be added to the error table.

Print_what

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

Listing 4.30: Print_what rule in LISS

In Listing 4.30, it is shown the syntax for printing a value (numeric or alphanumeric) in the output in LISS language.

In this case, we need to check the type of the *expression* in the context of the first alternative. If the type is a *set* then it must throw the number 7 error message (see Table 28).

Notice that the type allowed for the *expression* are :

- integer
- boolean
- sequence
- array

Read

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

Listing 4.31: Read rule in LISS

In Listing 4.31, it is shown the syntax for reading the input to get the value from the user to be stored in the given *identifier* in LISS. In this case, we need to check some possible errors.

If the *identifier* 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 (see Table 28).

If_then_else_stat

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

Listing 4.32: If_then_else_stat rule in LISS

Listing 4.32 defines the syntax of a conditional statement, in particular, for the 'if' statement.

As for all programming languages, the behaviour of an 'if' statement is the same. It means that the *expression* non-terminal must be of boolean type, in order to be possible to decide if it will enter to the next branch (*then*) or has to jump to the *else* branch. If the *expression* type isn't a boolean, then it must throw the number 7 error message (see Table 28).

For_stat

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

Listing 4.33: For_stat rule in LISS

Listing 4.33 defines the use of a 'for-loop' statement in LISS.

A particular case of this statement is the use of 'for-each' interval which is denoted with the keyword *inArray*.

In LISS, we are able to use a 'for-loop' which can access all the elements of an array, also called 'for-each'. In that 'for-each' context, we cannot use *step* non-terminal nor *satisfy* non-terminal.

Let's see an example of that case in Listing 4.34.

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

Listing 4.34: Example of an error message in for_stat rule

In Listing 4.34, the fact that there is an *inArray* keyword means that the statement is a 'for-each' loop and for this case we cannot use *step* or *satisfying* construct.

If the *step* non-terminal is present then we must throw the number 12 error message (see Table 28); if the *satisfy* non-terminal is present then we must throw the number 13 error message (see Table 28).

However, if the 'for-each' statement isn't used, we can use the known and normal behaviour of a 'for-loop' statement by using the *in* keyword instead of *inArray*.

Range

Listing 4.33 is specified how to expand the *interval* non-terminal in the context of the *for_stat* in LISS.

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

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

The *type_interval* non-terminal of the *interval* rule in LISS (Listing 4.33) tells us which kind of operation can be a 'for-loop' in LISS. As we can see there are two choices, the normal behaviour of the 'for-loop' statement (represented by *in range*) and the 'for-each' statement (represented by *inArray identifier*). In the case of the constructor, we need to check if the *identifier* variable is in the symbol table and if it isn't, the number 1 error message will be thrown (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 (see Table 28).

The *range* rule indicates us the limit bounds of a 'for-loop' statement and it is syntatically represented by: a limit inferior (*minimum*), two dots and a limit superior (*maximum*).

Minimum rule have two options; the first one is to write a constant number, the second one is to write a variable. Regarding the variable, we need to check if it is in the symbol table. If it isn't then it means that it isn't declared and must be thrown the number 1 error message (see Table 28). But if the variable is in the symbol table, we need to check its type. If the type isn't an *integer* then it must throw the number 5 error message (see Table 28).

In a similar way, *maximum* rule has the same behaviour as *minimum* rule. This means that we need to check if the *identifier* (variable) is in the symbol table. If it isn't then it means that the variable isn't declared and the compiler must throw the number 1 error message (see Table 28). However if the variable exists, we need to check its type. If the type isn't an *integer* then it must throw the number 5 error message (see Table 28).

Satisfy

Listing 4.33 is shown the syntax to define a condition *satisfy* in the context of the *for_stat* in LISS. Basically, the *satisfying* keyword means that there is a condition that must be evaluated 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 (see Table 28).

While_stat

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

Listing 4.35: While_stat rule in LISS

In Listing 4.35 is shown the syntax to write the interation control flow statement *while*.

For this case, we need to check if the condition (represented above by the non-terminal *expression*) has the correct type. Notice that a condition must be an expression with *boolean* type. If it isn't then it must throw the number 7 error message (see Table 28).

Succ_or_pred

```
succ_or_pred : succ_pred identifier
```

Listing 4.36: Succ_or_pred rule in LISS

Listing 4.36 defines the syntax of the increment or decrement statement in LISS. In this case, we need to check two things.

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

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

Tail

```
tail : 'tail' '(' expression ')'
```

Listing 4.37: Tail rule in LISS

Concerning the operations of sequences, Listing 4.37 defines the syntax for the tail function for sequences.

For this case, we need to see if the *expression* type has the correct type. If the *expression* doesn't have *sequence* type, then it must throw the number 7 error message (see Table 28).

Head

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

Listing 4.38: Head rule in LISS

Similarly, Listing 4.38 defines the syntax for the *Head* function for sequences.

For this case, we need to see if the *expression* type has the correct type. If the *expression* doesn't have *sequence* type, then it must throw the number 7 error message (see Table 28).

Cons

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

Listing 4.39: Cons rule in LISS

In Listing 4.39, we can see how to write the *Cons* function for sequences.

For this case, we need to see if both *expression* have the correct type. If the first *expression* (the left one) doesn't have *integer* type, then it must throw the number 7 error message (see Table 28). If the second *expression* (the right one) doesn't have *sequence* type, then it must throw the number 7 error message (see Table 28).

Delete

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

Listing 4.40: Delete rule in LISS

In Listing 4.40 is presented the syntax for the *delete* function for sequences.

For this case, we need to see if both *expressions* have the correct type. If the first *expression* (the left one) doesn't have *integer* type, then it must throw the number 7 error message (see Table 28). If the second *expression* (the right one) doesn't have *sequence* type, then it must throw the number 7 error message (see Table 28).

Copy_statement

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

Listing 4.41: Copy_statement rule in LISS

In Listing 4.41 is presented the syntax for the *copy* statement. For this case, we need to see if both the *identifiers* are in the symbol table and have the correct type. If one of the *identifier* is not in the symbol table, then it must throw the number 1 error message (see Table 28). After checking the availability of both *identifiers* in the symbol table, we need to check their type. If a variable is not of *sequence* type, then it must throw the number 5 error message (see Table 28).

Cat_statement

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

Listing 4.42: Cat_statement rule in LISS

In Listing 4.42 is shown the syntax for *cat* statement and it has the same behaviour as the *copy* statement.

This means that we need to check if both *identifiers* are in the symbol table and have the correct type. If one of the *identifiers* is not in the symbol table, then it must throw the number 1 error message (see Table 28). Then, after checking the availability of both *identifiers* in the symbol table, we need to check their type. If a variable is not of *sequence* type, then it must throw the number 5 error message (see Table 28).

Is_empty

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

Listing 4.43: Is_empty rule in LISS

In Listing 4.43, we can see the syntax for function *is_empty* for sequences. In this case, we need to check, only, the type of *expression* that must be a *sequence*. If it isn't, then it must throw the number 7 error message (see Table 28).

Length

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

Listing 4.44: Length rule in LISS

In Listing 4.44 is shown the syntax for function *length*, that is the same behaviour as *is_empty* function.

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

Member

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

Listing 4.45: Member rule in LISS

At last, Listing 4.45 defines the function *member* sequences. For this case, we need to check first the *identifier* and then the *expression*. If the *identifier* terminal is not in the symbol table, then we must throw the number 1 error message (see Table 28). Otherwise, if the variable is in the symbol table, we need to check the type of both (*identifier* and *expression*). *Identifier* must be an *sequence*, if it isn't then it must throw the number 5 error message (see Table 28). If *expression* isn't an *integer*, then it must throw the number 7 error message (see Table 28).

4.4 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.

4.4.1 *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 4.46:

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

Listing 4.46: Example of a sum operation with some numbers

In Listing 4.46, 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 primarily 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 4.47.

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

Listing 4.47: Example of conditional statements in LISS language

In Listing 4.47, 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 13.

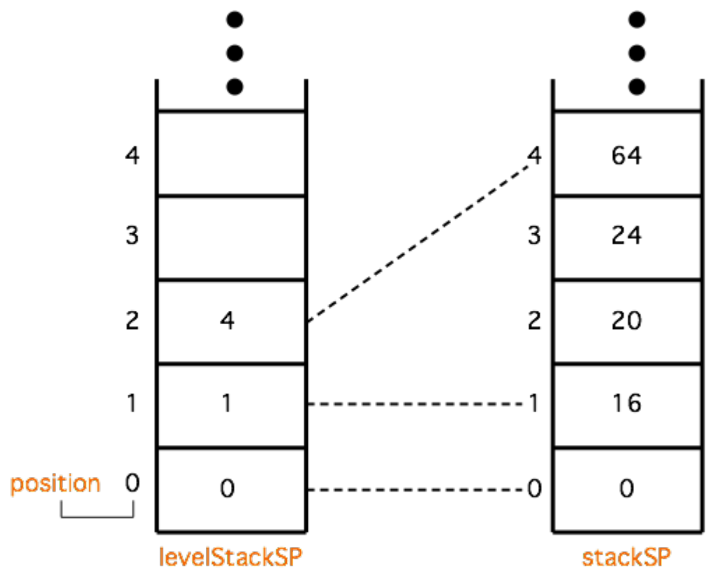


Figure 13.: Stack structure

In Figure 13, 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*.

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.

Calling a function

Calling a function requires some jump instruction. And those jump instructions may or may not loose informations available in the registers before processing the function.

That is why we need to create a mechanism for not loosing those informations before processing a function in MIPS assembly code.

And the mechanism is simple, every functions whose arguments have the possibility of calling a function or are used in an expression must use the stack available in MIPS architecture.

Let's first explain the use of the stack regarding to function call in expression. But before starting the explanation, notice that the arguments in function call doesn't allow the use of functions.

If a function call is used in an expression, it needs to check firstly the availability of the temporary registers.

If a temporary register contains a value and in this case, isn't empty; then it must save that information into the stack before calling the function call (example can be seen in Listing 4.49).

```
i = 2 + calculate();
```

Listing 4.48: Example of a function call in an expression statement

In Listing 4.49, the information that is stored to stack is the number 2. This is a valuable information that needs to be stored in order to continue and generate the correct code. At the end of the processement function, it will be restored the value.

Let's see the code generated in Listing 4.49.

```
li $t0,2
addi $sp, $sp, -4
sw $t0, 0($sp)
jal calculate
lw $t0, 0($sp)
addi $sp, $sp, 4
move $t1, $v0
add $t0, $t0, $t1
sw $t0, i
```

Listing 4.49: Code generated for the Listing 4.49

In Listing 4.49, it loads the value number 2 (line 1); then, the compiler knows that we function call will executed so it needs to save the number two into the stack (line 2 and 3); after it calls the instruction for executing the function (line 4); at the end of the execution of the function, it loads firstly the state that he got previously (which is loading the number two) to the register (line 5 and 6); then it loads the result of the function to the next register (line 7) and finally, it does the arithmetic operation (line 8) and stores it to the variable *i* (line 9).

However, if a function call sees that the temporary registers are empty. Then none of those mechanisms will be applied.

Let's see an example in Listing 4.50.

```
i = calculate();
```

Listing 4.50: Example of a function call in an assignment

As we can seen, in Listing 4.50, no information are processed before the call of the function *calculate()*. Which means that the mechanism of saving valuable information won't be applied (see in Listing 4.51).

```
jal calculate
move $t0, $v0
sw $t0, i
```

Listing 4.51: Code generated for Listing 4.51

Now, regarding to the sequence function, the mechanism that is done with function call is also applied with the same way. The difference is that on sequence function you can applied also the mechanism relatively to the arguments that have an expression statement.

For example, let's consider the next piece of code which uses a sequence function (see in Listing 4.52).

```
sequence2 = cons(3+head(sequence2),sequence2);
```

Listing 4.52: Example of using a sequence function in LISS

In Listing 4.52, the mechanism will be applied to both of the arguments available in the *cons* function. Firstly, due to the fact that in the second argument of the function *cons*, it must be a sequence type and use an expression statement (this means that we can nest more sequence functions in this argument). Secondly, the first argument must be an integer type and use an expression statement too (this mean that it can nest some sequence function in there (where those sequence function returns an integer)).

And for that purpose, the compiler needs to store all those valuable informations into the stack relatively for not loosing the consistency of the code.

Let's see and explain the code generated from the example in Listing 4.52 (see Listing 4.53).

```
1  lw $t0,sequence2
2  addi $sp, $sp, -4
3  sw $t0, 0($sp)
4  li $t0,3
5  addi $sp, $sp, -4
6  sw $t0, 0($sp)
7  lw $t0,sequence2
8  move $s0, $t0
9  jal head_sequence
10 lw $t0, 0($sp)
11 addi $sp, $sp, 4
12 move $t1, $v0
13 add $t0, $t0, $t1
14 move $s1, $t0
15 lw $t0, 0($sp)
16 addi $sp, $sp, 4
17 move $s0, $t0
18 jal cons_sequence
19 move $t0, $v0
20 sw $t0, sequence2
```

Listing 4.53: Example of code generated for Listing 4.52

In Listing 4.53, we load the variable *sequence2* relatively to the second argument of the function *cons* (line 1); then we store that information to the stack (line 2 to 3); after, we process for the information relatively to the first argument of the *cons* function and this means by loading the number 3 to a temporary register (line 4); now, the compiler knows that it must sum that number with a sequence function (which will use a jump instruction), so it will save the number 3 to the stack before calling the function *head* (line 5 to 6); then, he will process for the calling function (line 7 to 9); after finishing of executing the function, the compiler knows that it needs to load the previous state which was available before calling the function, so it gets back the number 3 by going to the stack (line 10 to 11); then, put the result of the *head* function to the next register available and do the sum (line 12 to 13); finally, the compiler will move the result of the sum to the appropriated register for calling the *cons* function (line 14) and reload the information that he stored previously to the stack (relatively the second argument of the *cons* function) (line 15 to 17); lastly, it will process the call of the *cons* function and store the result to the variable *sequence2* (line 18 to 20).

4.4.2 *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.

4.4.3 *Creating a variable in LISS*

- Level scope equals to zero

Let's see an example of LISS code in Listing 4.54.

```
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 4.54: Example of creating variables in LISS

In Listing 4.54, 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 4.54, 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 4.55).

```
.data
a : .word 0
b : .word 4
c : .word -1
d : .word +2
```

Listing 4.55: 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 4.55, 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 4.54, 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 4.56).

```
flag : .word 0
flag2 : .word 1
flag1 : .word 0
```

Listing 4.56: 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 4.54, 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 4.57).

```
array2 : .space 12
array1 : .space 12
array3 : .space 12
array4 : .space 36
```

Listing 4.57: Code generation of array variables in MIPS assembly code

In Listing 4.57, 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 4.54, 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 4.54, 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 :

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

Listing 4.58: 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 14).

So we created a system which has this structure (see in Listing 4.59).

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

Listing 4.59: 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).

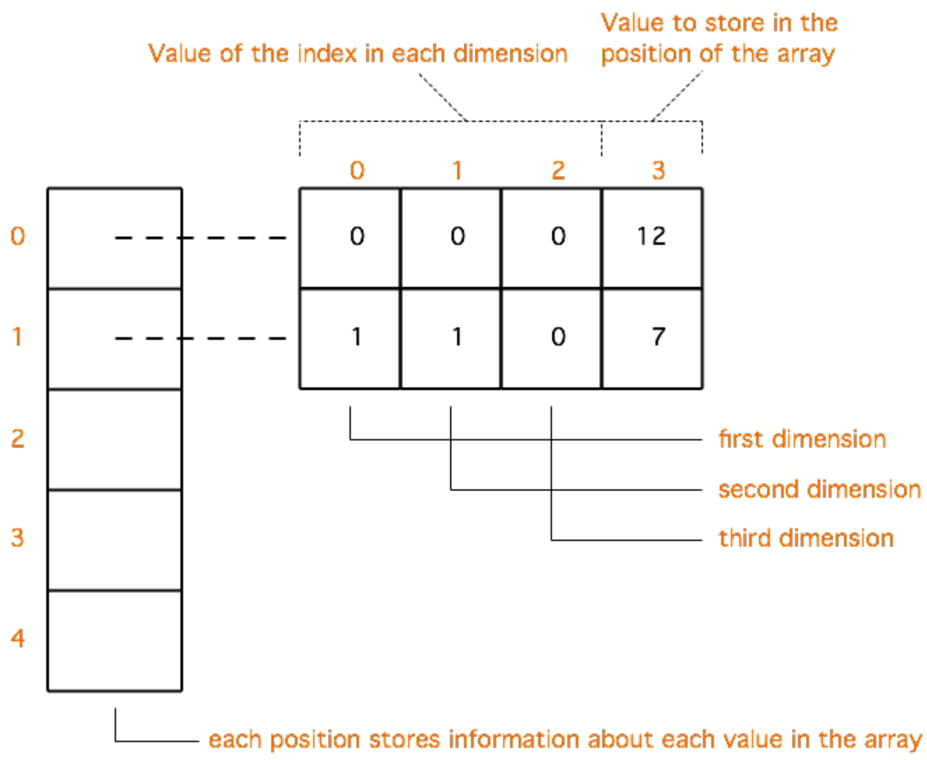


Figure 14.: Structure for saving information of each value declared in a array

So, in Figure 14, 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 14.

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 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:

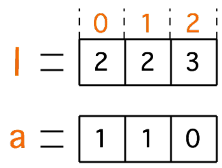
- **l** - array variable which has the informations about the limits of the array in question.
- **a** - array variable which has the information of the position of the array that need to be processed.

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 14 and Listing 4.58):



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 15 if the calculation was done correctly.

Using the positions of the variable array **l** and using them to find the right position in the array structure in Figure 15, 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.

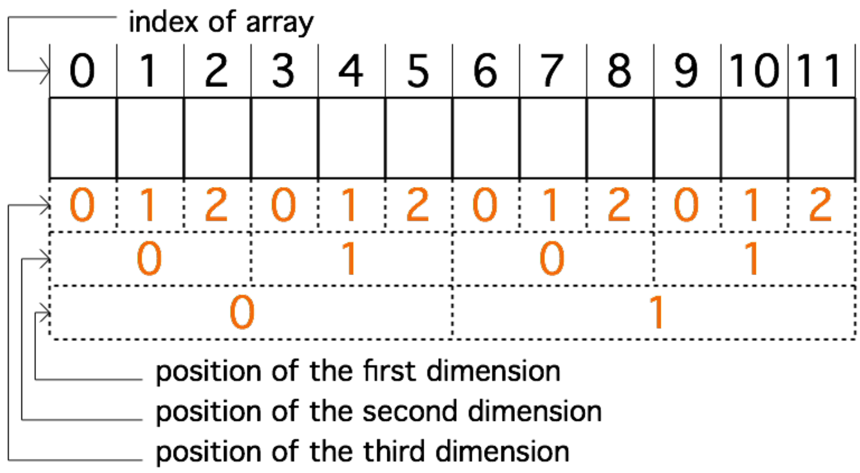


Figure 15.: Array structure with size 2,2,3.

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 4.54.

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 4.60.

```
.text
  main:
      ##### Initialize Value Array :array2#####
      li $t0,2    # 5:12
      li $t1,0    # 5:12
      sw $t0, array2($t1)
      li $t0,1    # 5:12
      li $t1,4    # 5:12
      sw $t0, array2($t1)
      li $t0,1    # 5:12
      li $t1,8    # 5:12
      sw $t0, array2($t1)
      ########################################
```

Listing 4.60: 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 4.60 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 4.60, 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 4.60, 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 4.54, 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 16.

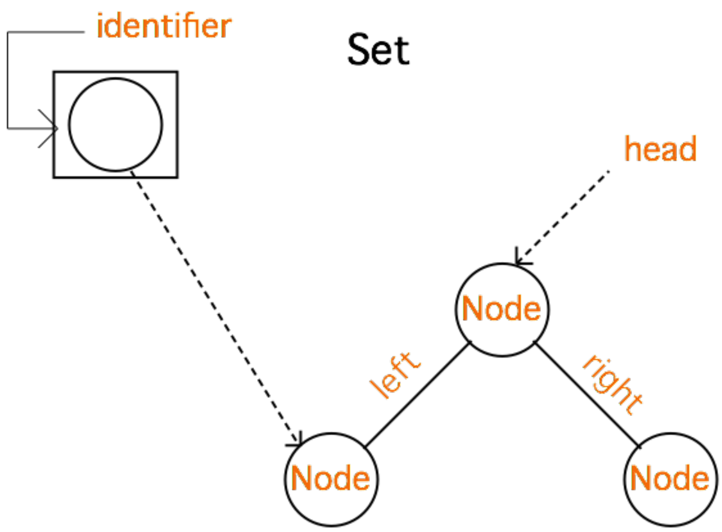


Figure 16.: Set structure in JAVA

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 17.

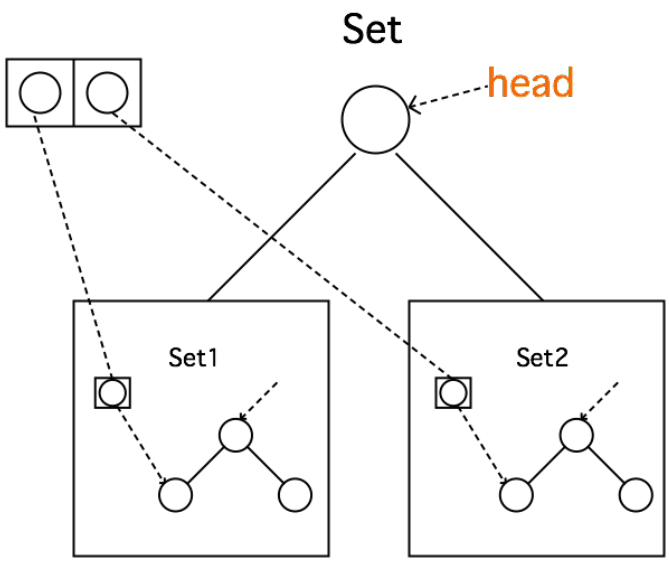


Figure 17.: Set structure in JAVA

In Figure 17, 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.

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:

```
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:

```
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:

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

Let's talk finally with the last line 8 in Listing 4.54, 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 4.54:

```
seq2 : .word -1
seq1 : .word -1
```

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 4.54, the code generated for the variable **seq2**.

```
lw $s0, seq2
li $s1, 1
jal cons_sequence
move $s0, $v0
li $s1, 2
jal cons_sequence
sw $v0, seq2
```

Listing 4.61: 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 4.54, the compiler needs to take the value 1 and 2 and generate code for them to the sequence.

In Listing 4.61, the compiler firstly creates an instruction which puts the address of the sequence variable to a saved registers ($ *s0*), 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 $v0 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 $v0 to the register $s0, 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 4.62.

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

Listing 4.62: Example of creating variables in a level scope greater than 0

As we can see in Listing 4.62, 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 18.

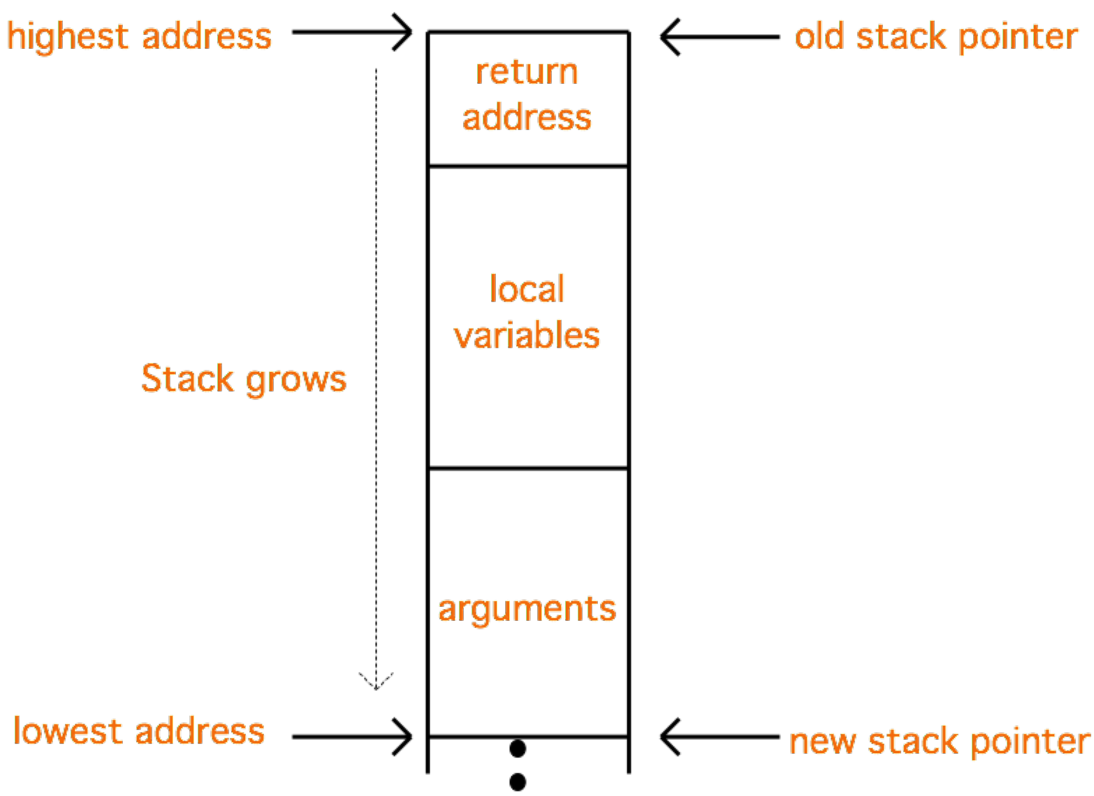


Figure 18.: Architecture of the stack relatively to a function in LISS

So, in Figure 18, the stack is organized by this way:

- 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 18, 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 4.62.

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 4.63).

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

Listing 4.63: Initialization of MIPS code generated for a function

In Listing 4.63, 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 4.62, we are declaring integer variables and the code generated for those variables are available in Listing 4.64.

```
li $t0,0
sw $t0, 0($sp)
li $t0,4
sw $t0, 4($sp)
```

```
5  li $t0,-1
6  sw $t0, 8($sp)
7  li $t0,+2
8  sw $t0, 12($sp)
```

Listing 4.64: Declaring integer variables in level scope greater than 0 on MIPS.

In Listing 4.64, 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 4.62, refers to variables declared with the type boolean and the code generated for that type is visible in Listing 4.65.

```
1  li $t0,0
2  sw $t0, 16($sp)
3  li $t0,1
4  sw $t0, 20($sp)
5  li $t0,0
6  sw $t0, 24($sp)
```

Listing 4.65: Declaring boolean variables in level scope greater than 0

In Listing 4.65, 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 4.65 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 4.62, refers to variables declared with the type array and the code generated for the variable **array2** is visible in Listing 4.66.

```
##### Initialize Array :array2#####
li $t0,0
sw $t0, 28($sp)
li $t0,0
sw $t0, 32($sp)
```

```
6  li $t0,0
7  sw $t0, 36($sp)
8  ######################################
9  ##### Initialize Value Array :array2#####
10 li $t0,2
11 li $t1,0
12 li $t2,28
13 add $t1, $t1, $t2
14 add $t1, $t1, $sp
15 sw $t0, ($t1)
16 li $t0,1
17 li $t1,4
18 li $t2,28
19 add $t1, $t1, $t2
20 add $t1, $t1, $sp
21 sw $t0, ($t1)
22 li $t0,1
23 li $t1,8
24 li $t2,28
25 add $t1, $t1, $t2
26 add $t1, $t1, $sp
27 sw $t0, ($t1)
28 ######################################
```

Listing 4.66: 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 4.66). 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 4.66:

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 4.62, 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 4.62, refers to the declaration of variables with the type sequence and the code generated for the variable **seq2** is visible in Listing 4.67.

```
1 li $t2,-1
2 sw $t2, 148($sp)
3 lw $s0, 148($sp)
4 li $s1, 1
5 jal cons_sequence
6 move $s0, $v0
7 li $s1, 2
8 jal cons_sequence
9 sw $v0, 148($sp)
```

Listing 4.67: 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 4.67) 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 4.67), then we load the value to a register (line 5 of Listing 4.67) and finally, we call the function that will process the concatenation of the value to the sequence.

Notice that after creating those generated code relatively to those variable declarations, comes the flow of the execution subprogram (everything that is created under the statements part of the subprogram). Finally, at the end of the execution of the subprogram, we need to remove the allocated stack memory of the function and return.

Let's see the code generated in Listing 4.68.

```
lw $ra, 156($sp)
```

```
2 addi $sp, $sp, 160
3 jr $ra
```

Listing 4.68: Exiting the function in MIPS assembly code

In Listing 4.68, line 1 will get the return address of the function; then at line 2 it will remove the stack allocated for the function and finally at line 3, it will process for exiting the function with the return address.

4.4.4 *Loading a variable or a value*

Loading a variable depends in only one factor, the level scope of the variable.

If the level scope of the variable is equals to zero then it will load the variable by using the name of the variable (see in Listing 4.69).

```
lw $t0,b
```

Listing 4.69: Loading a variable with level scope equals to zero

Otherwise if the level scope is greater than zero, then it must use the algorithm which will found the position in the stack and load the value throw the position in the stack (see in Listing 4.70).

```
lw $t0, 4($sp)
```

Listing 4.70: Loading a variable with level scope greater than zero

Regarding to load a value, the only possible values that can be loaded are integer and boolean types and they doesn't depend on the level scope. Notice that the instruction for loading is the same, just it depends the type that will be used.

If it is a boolean type, the value *true* will be loaded with one; otherwise the value *false* will be loaded with zero (see in Listing 4.71).

```
li $t0,1     # true value being loaded
li $t0,0     # false value being loaded
```

Listing 4.71: Loading a boolean value

Yet, if it is an integer type, then it will load the value that it will be declared (see in Listing 4.72).

```
li $t0, 5          # loading the number 5
```

Listing 4.72: Loading an integer value

4.4.5 *Assigning in LISS*

Assigning in LISS requires to understand the mechanics of that structure due to its complexity maneuverability.

For assigning in LISS, it requires three things: the sign equals, the left content and the right content relatively to the equal sign.

In order to make the things workable, the types in the left and right content of the equal sign must be equal. Let's show, in this case, which cases that can appear for an assignment statement in LISS (see in Listing 4.73).

```
1 i = 1 + 2 + 3;
2 flag1 = 1 < 3;
3 array1[1] = 10;
4 array1 = array2;
5 array1 = [2];
6 set1 = set2;
7 sequence1 = sequence2;
```

Listing 4.73: Examples of assignment for different types in LISS

Assignment for integer

In Listing 4.73, line 1 and 3 are statement that deals with integer type and their operations mode for generating the MIPS assembly code are distinct and needs to make a particular attention. The normal and general way is done by assigning the content to an integer variable, instead that assigning the content to a position of an array needs to do some particular mechanism.

Let's explain it with the code generated for both of those lines.

```
1 li $t0,1
2 li $t1,2
3 add $t0, $t0, $t1
4 li $t1,3
5 add $t0, $t0, $t1
6 sw $t0, i
```

Listing 4.74: Code generated for line 1 in Listing 4.73

For example, if you store an arithmetic operations to an integer variable then it must firstly create the code of the arithmetic operations (line 1 to 5 of Listing 4.74) and then store it to the variable (line 6 of Listing 4.74).

But if we wants to assign a value to an array, the code generated will be by this way (see in Listing 4.75).

```
1  li $t0,1
2  ####Verify limits of the array####
3  li $t1,0
4  slt $t2, $t0, $t1
5  sltu $t2, $zero, $t2
6  xori $t2, $t2, 1
7  li $s0, 7
8  beqz $t2, indexoutofboundError
9  li $t1,4
10 slt $t2, $t0, $t1
11 li $s0, 7
12 beqz $t2, indexoutofboundError
13 ####End of the verification####
14 li $t1,4
15 mul $t0, $t0, $t1
16 li $t1,10
17 sw $t1, array1($t0)
```

Listing 4.75: Code generated for line 3 in Listing 4.73

First, it loads the position where the value will be stored (in this case, it loads the number 1 (line 1 of Listing 4.75).

Then it is introduced some mechanism for checking the position of the value that needs to be stored and check if they are inside of the limits of the array (line 3 to 12 of Listing 4.75). Basically, it tests the position between the position 0 and 4, if they are not on those limits then it will give an index out of bound error and will stop the program. Though, if the position is inside of the limits then it will multiply the position by four (line 14 to 15 of Listing 4.75) and after, it will load the right content relatively to the code generated for this part (line 16 of Listing 4.75). Finally, it will store the content to the position of the array (line 17 of Listing 4.75).

Notice that the assignment statement, has the possibility of writing an expression in the right part of the equal sign. And this means that it can be used some operators for writing some complex arithmetic operations (only integer operators) and they are listed above:

- + (plus sign)
- - (minus sign)
- * (multiply sign)
- / (division sign)

Relatively to those operators, we must inform that each one uses only one instruction and not a lot of instructions (will be talked further for others operators) and it can be seen in Listing 4.76.

```
add $t0, $t0, $t1 # plus instruction
sub $t0, $t0, $t1 # minus instruction
mul $t0, $t0, $t1 # multiply instruction
div $t0, $t0, $t1 #division instruction
```

Listing 4.76: Code generated for arithmetic operators

So, the mechanism of generating the code is always the same if one of those operators are found, load the code generated relatively to the left part of the operator then load the code generated relatively to the right part and finally add the operator instruction.

Assignment for boolean

In Listing 4.73, line 2 is a statement that deals with boolean type.

The method of processing the information is equals to storing an arithmetic operations to an integer variable, first it loads the code generated to the content available at the right equal sign and then the code generated to the left equal sign for storing.

Let's see the code generated for a boolean assignment (see in Listing 4.77).

```
1 li $t0,1
2 li $t1,3
3 slt $t0, $t0, $t1
4 sw $t0, flag1
```

Listing 4.77: Code generated for line 2 of Listing 4.73

In Listing 4.77, it is processed the content relatively to the right part of the equal sign (line 1 to 3) and then it is stored to the variable associated to the assignment statement (line 4).

As explained before, some operators are available for boolean assignment which can be listed under:

- == (double equal sign)
- != (different sign)
- < (less sign)
- > (greater sign)
- <= (less or equal sign)

- >= (greater or equal sign)
- || (or sign)
- ! (negation sign)

Relatively to those operators, only tree operators uses one instruction($<$, $>$, $||$) and the others uses a logic of mechanism for having the same behaviour which requires more than one instruction. Notice that this is due to the fact that MIPS architecture doesn't have all the logic instruction implemented.

Let's see the code generated for each one.

```
slt $t2, $t0, $t1
sltu $t2, $zero, $t2
xori $t2, $t2, 1
slt $t3, $t1, $t0
sltu $t3, $zero, $t3
xori $t3, $t3, 1
and $t2, $t2, $t3
```

Listing 4.78: Code generated for double equal sign in MIPS

In Listing 4.78, we translate the behaviour of a double equal sign to:

$$x == y \iff (\neg(x < y)) \wedge (\neg(x > y)) \tag{3}$$

```
slt $t2, $t0, $t1
slt $t3, $t1, $t0
or $t2, $t2, $t3
```

Listing 4.79: Code generated for different sign in MIPS

In Listing 4.79, we translate the behaviour of the different sign to:

$$x! = y \iff (x < y \vee x > y) \tag{4}$$

```
slt $t0, $t1, $t0
sltu $t0, $zero, $t0
xori $t0, $t0, 1
```

Listing 4.80: Code generated for less or equal sign in MIPS

In Listing 4.80, we translate the behaviour of the less or equal sign to:

$$x <= y \iff \neg(x > y) \tag{5}$$

```
slt $t0, $t0, $t1
sltu $t0, $zero, $t0
xori $t0, $t0, 1
```

Listing 4.81: Code generated for greater or equal sign in MIPS

In Listing 4.81, we translate the behaviour of the greater or equal sign to:

$$x >= y \iff \neg(x < y) \tag{6}$$

Now, let's see the code generated for a negation (see Listing 4.82).

```
sltu $t0, $zero, $t0
xori $t0, $t0, 1
```

Listing 4.82: Code generated for negation sign in MIPS

And finally, the last three operators which are available under:

```
slt $t0, $t0, $t1 # less instruction
slt $t0, $t1, $t0 # greater instruction
or $t0, $t0, $t1  # or instruction
```

Notice that the difference between a less and a greater instruction is only in the use of the registers and also, the mechanism for generating those code (boolean assignment) is the same as talked previously. First, load the code generated relatively to the left part of the operator; second, load the code generated for the right part of the operator and third, add the operator instruction.

Assignment for array

Assigning arrays can be made by two ways (see line 4 and 5 in Listing 4.73).

One way is to assign the name of an array variable to another array variable and the idea of that purpose is to copy the entire array to the other array (see in Listing 4.83).

```
li $t0,0
lw $t1, array2($t0)
sw $t1, array1($t0)
li $t0,4
lw $t1, array2($t0)
sw $t1, array1($t0)
```

Listing 4.83: Code generated for line 4 in Listing 4.73

In Listing 4.83, it is copying an array with size two to another array. And before to begin the process of copying, it needs to check if the limits and the size are the same. Otherwise, the copy won't be able to do and it will throw an error.

Basically, the idea is to copy the content of each position in array2 to the same position in array1. First, it loads the position (line 1 of Listing 4.83), then it loads the value available at that position in array2 (line 2 of Listing 4.83) and finally it stores the value to the same position in array1 (line 3 of Listing 4.83). Yet, it must continue to do the same process again for the next remaining position of those arrays in order to complete the copy process.

Now, the other way for assign an array is by declaring the content of each position of the array (line 5 of Listing 4.73). And the processus is different because it needs to do some calculations for knowing the position of each value that needs to be stored. Notice that the calculation is the same as it was declaring an array in the *declarations* part and that the limits and the size must be less or equal than the variable. Otherwise, if the limits and size aren't in agreement then it will throw an error message.

Let's explain it with the code generated for that situation (see in Listing 4.84).

```
1  li $t0,2
2  li $t1,0
3  sw $t0, array1($t1)
4  li $t0,0
5  li $t1,4
6  sw $t0, array1($t1)
```

Listing 4.84: Code generated for line 5 of Listing 4.73

In Listing 4.84, we first load the value that needs to be stored (line 1), then comes the position of the array (line 2) and finally, the storage of that value to the required position in array1 (line 3). If some positions weren't declared or missed then it will put them with the value zero. For instance, the case of the last position (line 4 to 6) of array1 because his size is equal to two and it was declared only one value in the assignment statement.

Assignment for set

Assigning sets is a little different from the others assignment (see line 6 of Listing 4.73).

Basically, it doesn't compute any instructions but instead, it reorganize the JAVA structure available and linked to the set variable, in the symbol table, by changing it to the correct structure according to the assignment.

Also, notice that it is possible to create a statement with more expressivity by using some operators relatively to the set type and they are listed above:

- ++ (union sign)

- ** (intersection sign)

Again, using those operators with sets doesn't compute any instructions but instead reorganize the structure of the set. And the meaning of each one is the same as the mathematical way (see equations above).

$$A \cup B = \{x : x \in A \text{ or } x \in B\} \tag{7}$$

$$A \cap B = \{x : x \in A \text{ and } x \in B\} \tag{8}$$

Assignment for sequence

Assigning sequences is really simple and uses a certain mechanism took by another imperative language (see line 7 of Listing 4.73).

The idea of assigning two sequences is to change the pointer of one the sequence to the other sequence and in this case, the sequence that is available in the left part of the equal sign will change the address (where it is pointing in the heap memory) to the address of the sequence available in the right part of the equal sign.

Let's see a code generated relatively to assigning a sequence in Listing 4.85.

```
1 lw $t0,sequence2
2 sw $t0, sequence1
```

Listing 4.85: Code generated for line 7 of Listing 4.73

In Listing 4.85, we load the address that the variable *sequence2* is pointing in the heap memory (line 1) and then, it stores that address to the variable *sequence1* (line 2).

Notice that assigning sequences, doesn't copy every elements of one sequence to another one. Instead, there is a function which does that and named by *copy* function.

4.4.6 *Set operations*

The type sets can be operated only by one way: using it as an assignment statement (see Listing 4.86).

```
program test{
    declarations
       flag -> boolean;
       set1 = {x | x>1} -> set;
    statements
       flag = 4 in set1;
}
```

Listing 4.86: Example of using a set in LISS

And for that, we need to use a special operator (named *in*). The idea of that operator is to test a certain number relatively to the set used in the assignment.

In this case, the example available in line 6 of Listing 4.86 will test if the number 4 is in the range of the set named *set1*.

But it needs to be generated something in order to check the truth of that expression and at the end get the result. That is why we need to generate the code with the use of that number accordingly to the expression of the set.

Let's look the code in Listing 4.87.

```
1 li $t0,4
2 li $t1,1
3 slt $t0, $t1, $t0
4 sw $t0, flag
```

Listing 4.87: Code generated for line 6 of Listing 4.86

Basically, from line 1 to 3 of Listing 4.87, it generates the code of that set with the number that it is associated to the left of the *in* operator. After processing and checking the expression, it stores the result to the boolean variable *flag* (line 4 of Listing 4.87). And the result will say if the number 4 is in the range of the set.

The mechanism of generating the code with sets is done by generating the code of the expression relatively to the set accordingly the number that we are trying to test and then, storing the result to a boolean variable.

4.4.7 *Sequence operations*

The type sequences can be operated by two ways:

1. statement
2. expression

And each sequence functions are defined by those mode of operations (see Table 29).

Each function available in Table 29, are, basically, functions in MIPS assembly code. This means that it needs to use a jump instruction for processing any sequence function. And notice that it uses the name of the sequence function in MIPS (available in Table 29) for generating the appropriated sequence jump instruction (see an example in Listing 4.88).

Table 29.: Sequence function name and their mode of operations

Mode operations	Name of the sequence function in LISS	Name of the sequence function in MIPS
expression	tail	tail_sequence
	head	head_sequence
	cons	cons_sequence
	isMember	member_sequence
	isEmpty	is_empty_sequence
	length	length_sequence
	del	delete_sequence
statement	copy	copy_sequence
	cat	cat_sequence

```
jal head_sequence
```

Listing 4.88: Example of processing a head function in MIPS

Now, take care that before invocating the jump instructions; it is needed to load the arguments relatively to the type of the sequence function. And they process with a certain logic for loading those arguments:

1. load the register $s0 with the name of the variable sequence.

2. load, in a sequential way, the rest of the arguments (relatively to the type of the sequence function) into the next saved temporary registers ($s1, $s2,...).

For example, *cons* function have two arguments in order to make it workable. Let's see how is the code generated for that case in Listing 4.89.

```
1 lw $t0,sequence2
2 addi $sp, $sp, -4
3 sw $t0, 0($sp)
4 li $t0,3
5 move $s1, $t0
6 lw $t0, 0($sp)
7 addi $sp, $sp, 4
8 move $s0, $t0
9 jal cons_sequence
```

Listing 4.89: Example of a code generated for the function *cons*

In Listing 4.89, we load the *sequence2* variable (line 1) in a first step; then we process to storing that information into the stack (line 2 to 3); after, we process the information relatively to the code generated for the other argument (second one) and we move to the correct register (register $s1) (line 4 and 5); finally, we go get back of the value that we stored earlier in the stack and we put it to the register $s0 (line 6 and 8). Lastly, we call the function *cons_sequence* for processing to the execution of the function (line 9).

Notice, that the way we are doing those mechanics aren't done in a random way but yes in a certain way and this is due for one reason.

Only the functions who are available to use in the *expression* rule and can use sequence type in some of their arguments, must use that way of invocating the code generated relatively to Listing 4.89.

Let's see an example for understanding it better (see Listing 4.90).

```
head(cons(3,cons(4,cons(5,sequence2))));
```

Listing 4.90: Example of using a function call as argument

We must process the information for the most inner function in Listing 4.90, in this case *cons(5,sequence2)*, and then getting back and process the other functions.

Let's see how it is done above:

1. cons(5,sequence2)
2. cons(4,cons(5,sequence2))
3. cons(3,cons(4,cons(5,sequence2)))
4. head(cons(3,cons(4,cons(5,sequence2))))

And this is why we always generate the code relatively to the sequence type in the first position and then, the others arguments (due to that reason).

Relatively to the *copy* and *cat* function, as they are not used on an expression but yes as a statement. They use only one simple mechanism and this is done by loading the sequence variable to the respective saved temporary register.

Let's see a code relatively to those two functions in Listing 4.91.

```
lw $s0, sequence1
li $s1, -1
jal copy_sequence
sw $v0, sequence2
li $v0, 0
lw $s0, sequence1
lw $s1, sequence2
jal cat_sequence
sw $v0, sequence1
li $v0, 0
```

Listing 4.91: Example of code generated for copy and cat statement

Basically, regarding to the use of a *copy* statement (from line 1 to 4 of Listing 4.91), it load the first argument of the function (line 1) and then load an empty sequence (line 2) (remind that an empty sequence holds the value NULL which is minus one). Then it call the cat function (line 3) and finally store the new address to the variable *sequence2* (line 4). Lastly, we reset the content of the register \$v0 and this is done for internal reason relatively to the execution of assembly code (line 5).

Take care that the copy statement, copies every element of a sequence to another sequence.

Relatively to the use of a *cat* statement (from line 6 to 10 of Listing 4.91, it is the same way as the *copy* mechanism. But instead of loading an empty sequence as second argument, it loads the appropriated sequence (line 7).

4.4.8 *Calling a function in LISS*

Calling functions which are already created in LISS, requires some mechanism.

As we know, functions deals with the stack available in MIPS architecture and the stack will hold informations about the arguments, local variables and return adress relatively for the use of a function.

This means that if a function have one or more arguments associated to the function, it is in the duty to use a certain mechanism for passing that information to the stack.

And by this way, it is needed to generate some code for that situation(see in Listing 4.92).

```
1 lw $t0 ,a
2 sw $t0 , −12($sp)
3 jal factorial
```

Listing 4.92: Code generated for calling a function in MIPS

So the idea of calling a function is done by this way: if there are some arguments, then it is needed to generate the code relatively to those arguments and store them in the right position of the stack and finally, call the function. Otherwise if there is no arguments, then it must only call the function.

In Listing 4.92, line 1 and 2 loads the variable *a* and then stores in the right position of the stack. Lastly, it calls the function by doing a jump instruction (line 3).

4.4.9 *Creating I/O in LISS*

Let's see an example of LISS code in Listing 4.93.

```
1 write();
2 write(a);
3 write("hello");
4 writeln();
5 writeln(a);
6 writeln("hello");
7 input(a);
```

Listing 4.93: Example of I/O statements in LISS

Firstly, let's talk about the output statements. They are known by *write* and *writeln*.
Write statement is an output which only prints the content and doesn't add the return carriage. However *writeln* statement is an output which prints the content and add the return carriage at the end.
Those output statements have three possibility of being declared:

1. empty (line 1 and 4 of Listing 4.93)
2. expression (line 2 and 5 of Listing 4.93)
3. string (line 3 and 6 of Listing 4.93)

Basically, for generating each output statements, we need to generate firstly the code regarding to the content available in the parentheses and then the code related to the type of output statements is available.
If the output of the statement is a *write*, then the code will be :

```
jal write
```

Otherwise if the output of the statement is a *writeln*, then the code will be :

```
jal writeln
```

Notice that the behavior of the output statement *writeln* reuses the *write* statement plus adds the instruction code for adding the carriage return (see Listing 4.94).

```
la $a0, writestring0
li $v0, 4
jal write
jal writeln
```

Listing 4.94: Code generated for line 6 in Listing 4.93

And the code already predifined regarding to both of the statement is available in Listing 4.95.

```
write:
syscall
jr $ra
writeln:
li $v0, 4
la $a0, newline
syscall
jr $ra
```

Listing 4.95: MIPS assembly code of write and writeln

Now, let's talk about the input statement which is known by *input*.

The idea behind of that statment for generating the code, it is to call the function which process for the output in MIPS and then store the information to the variable that is associated to the *input* statment inside of the parentheses.

Let's see the code generated of line 7 in Listing 4.93 below:

```
jal read
move $t0, $v0
sw $t0, a
```

As we can see, after processing the read statement. We need to move the value that the user have put to another register and finally, store the value to the variable that was associated.

This is the code that is predefined for the read statement (see in Listing 4.96).

```
read:
  li $v0,4
  la $a0,messagereadvalue
  syscall
  li $v0,5
  syscall
```

```
    jr $ra
```

Listing 4.96: Read statement code in MIPS

4.4.10 *Creating conditional statement in LISS*

Creating a conditional statement in LISS is basically following that schema available in Figure 19.

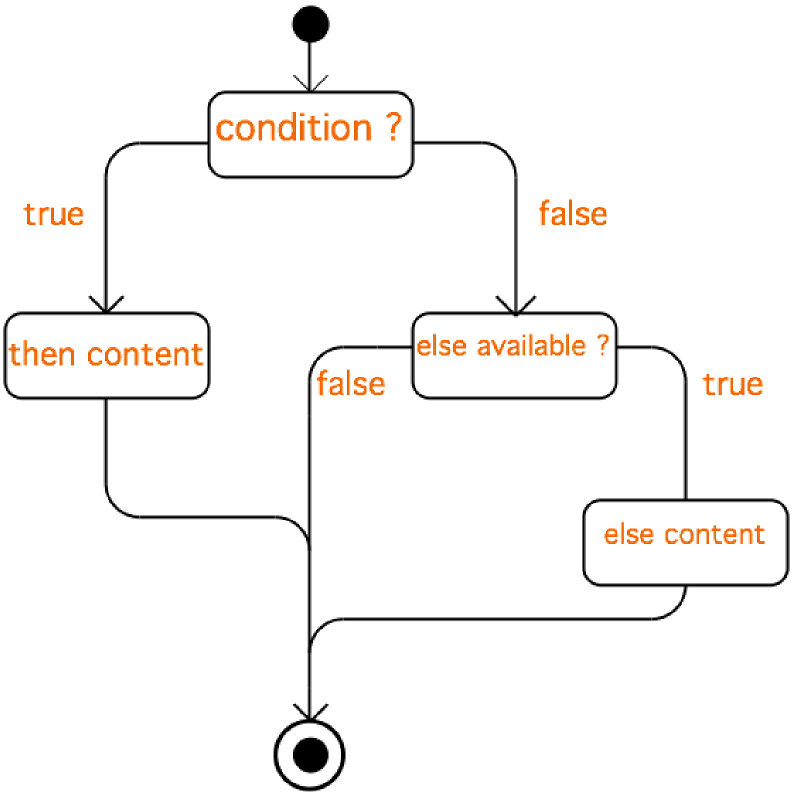


Figure 19.: Schema of the conditional statements in LISS

And for a better view of how it is generated let's see an example of a conditional statement in Listing 4.97.

```
if(flag)
then{
  writeln("Then content.");
}else{
  writeln("Else content.");
}
```

Listing 4.97: Example of conditional statement in LISS

Basically we have, in Listing 4.97, a conditional statement with an if and an else statement. The condition of the if-statement in this case is false, so by this way the else content will be processed when it will be executed.

Now let's see how the code is generated in Listing 4.98.

```
1    lw $t0,flag
2    bne $t0, 1, else1
3    la $a0, writestring0
4    li $v0, 4
5    jal write
6    jal writeln
7    j l1
8  else1:
9    la $a0, writestring1
10   li $v0, 4
11   jal write
12   jal writeln
13 l1:
```

Listing 4.98: Code generated for conditional statement in MIPS

In Listing 4.98, line 1 is the piece of code relatively to the condition of the conditional statement, then comes at line 2 the instruction which compares the value of the condition to the value true. If the value are different then it must jump to the else-condition. Otherwise if the value are the same, then it process the next instruction available in line 3 which is the code instruction available in the content of the then-statment. At the end of the then-statement, it exit the conditional statement (line 7).

Now, if the condition of the statement is true, then it will jump to the branch of the else-statement (line 8) and execute every code instruction available there. Notice that it won't jump at the end as if it will be with the then-statement content, it will process the next instruction sequentially without jumping.

Let's consider now that in the LISS code, no else-statement is used. Then the only thing that disappears is the whole code below of the else branch in Listing 4.98.

Let's see an example of the same code relatively to Listing 4.98 without the else-statement in Listing 4.99.

```
  lw $t0,flag
  bne $t0, 1, else1
  la $a0, writestring0
```

```
    li $v0, 4
    jal write
    jal writeln
    j l1
  else1:
```

Listing 4.99: Code generated for conditional statements without an else-statement in MIPS

4.4.11 *Creating iterative statement*

In LISS, we have three different ways of creating iterative statement.

They are known by:

1. For-loop with 'in' condition.
2. For-loop with 'inArray' condition.
3. While-loop.

In Figure 20, we can see the routine of a for-loop with an 'in' condition.

Notice that the black circle is where the flow of the for-loop statement begins and that the double circle is where it finish of executing the flow of the for-loop statement.

Let's see a piece of LISS code relatively to a for-loop statement with an 'in' condition in Listing 4.100.

```
for(a in 1..5) stepUp 1 satisfying flag==true{
  writeln(a);
}
```

Listing 4.100: Example of a for-loop statement with 'in' condition in LISS

We can see in Listing 4.100, that it is created a for-loop statement with an 'in' condition and beside that there is also more informations relatively to that loop. Informations that refers about the method for incrementing (*stepUp*) and a condition which must be tested every time before executing the block content inside of the for-loop statement (*satisfying*).

Let's see the code generated for that piece of code in Listing 4.101.

```
    li $t0,1
    sw $t0, a
  for_loop1:
    lw $t0,a
    li $t1,5
    slt $t0, $t1, $t0
```

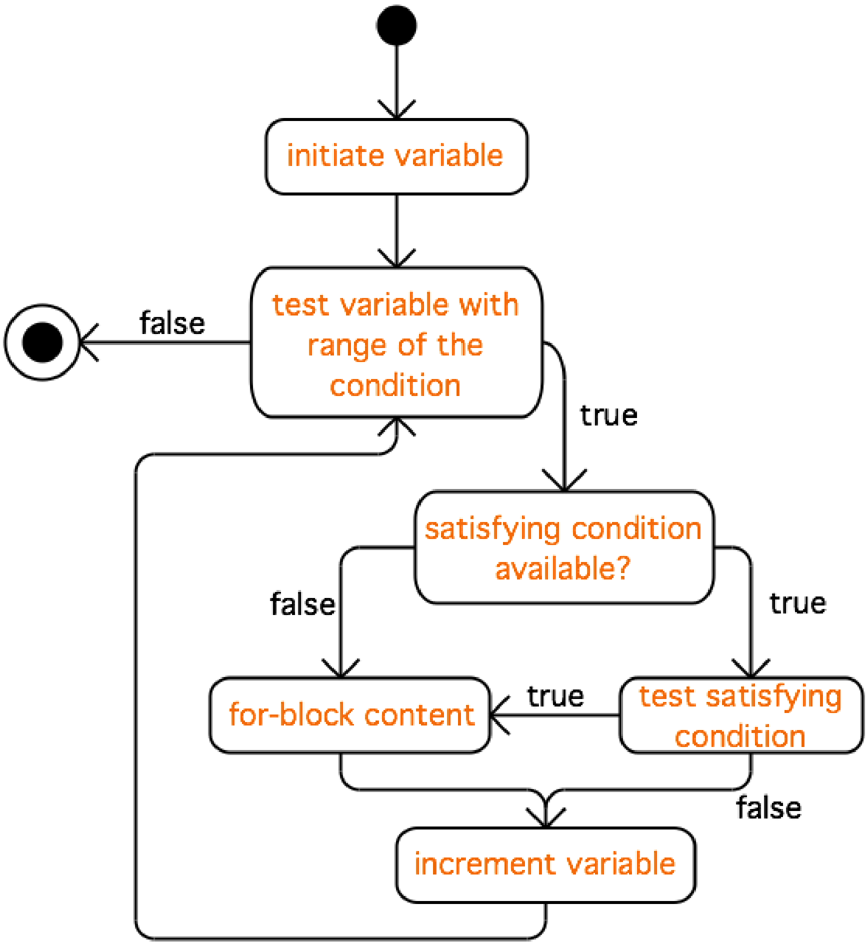


Figure 20.: Schema of the for-loop statement relatively with the use of the condition 'in'

```
    sltu $t0, $zero, $t0
    xori $t0, $t0, 1
    bne $t0, 1, for_exit1
    lw $t0,flag
    li $t1,1
    slt $t2, $t0, $t1
    sltu $t2, $zero, $t2
    xori $t2, $t2, 1
    slt $t3, $t1, $t0
    sltu $t3, $zero, $t3
    xori $t3, $t3, 1
    and $t2, $t2, $t3
    move $t0, $t2
    bne $t0, 1, satisfying_exit1
    lw $t0,a
```

```
22      move $a0, $t0
23      li $v0, 1
24      jal write
25      jal writeln
26   satisfying_exit1:
27      lw $t1,a
28      li $t2,1
29      add $t1, $t1, $t2
30      sw $t1, a
31      j for_loop1
32    for_exit1:
```

Listing 4.101: Code generated for the LISS code in Listing 4.100

In Listing 4.101, line 1 to 2 we create the variable *a*; line 3 we create the for-loop branch; line 4 to 9 we test if the value of the variable *a* is in the range of the for-loop condition. If no, we exit the for-loop flow by going to the branch named *for_exit1*(line 32); otherwise we continue the flow of the for-loop execution by going to line 10. Line 10 to 20 refers to the *satisfying* condition. If the condition isn't satisfied then it goes to the branch named *satisfying_exit1*, otherwise it continues the flow of the execution in line 21. Line 21 to 25 is the code contained in the block of the for-loop statement. Notice that line 27 to 30 is the piece of code who will increment the variable *a* relatively to *stepUp*. At the end (line 31), it jumps to the branch of the for-loop statement *for_loop1*(line 3).

Notice also, that *stepUp* and *satisfying* informations are optional statement. If the satisfying statement is not available then the line from 10 to 20 and line 26 of Listing 4.101 will be removed. Regarding to the *step* statement, even if the information is not available, by definition a for-loop statement needs to have some kind of iteraction by step by step. So, by omission, it will increment the variable by one; otherwise if the information is available, it will increment or decrease relatively the information shown.

In Figure 22, we can see the routine of a for-loop with an 'inArray' condition.

The idea of that for-loop statement is to have some kind of a foreach statement in LISS. Basically, the for-loop statement will pass to every positions of the array. Notice that when it is used that for-loop statement, it is unavailable for the user of declaring any *step* or *satisfying* statement.

Let's see a piece of LISS code relatively to a for-loop statement with an 'inArray' condition in Listing 4.102.

```
for(a inArray array1){
    writeln(a);
}
```

Listing 4.102: Example of a for-loop statement with 'inArray' condition in LISS

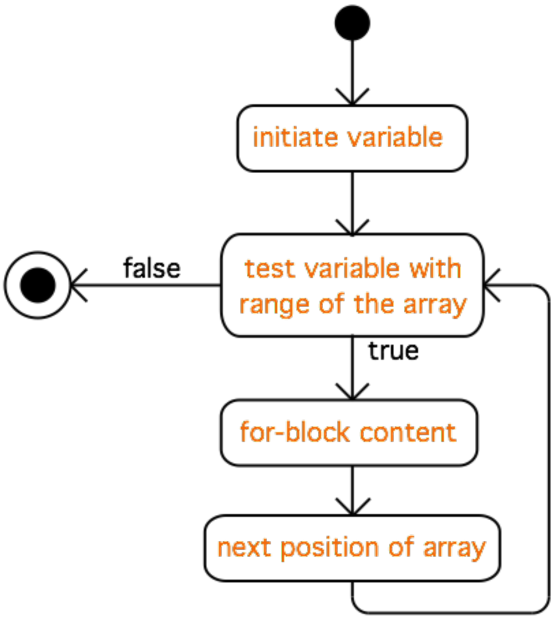


Figure 21.: Schema of the for-loop statement relatively with the use of the condition 'inArray'

We can see in Listing 4.102, that it is created a for-loop statement where the *array1* is the variable of type array and the variable *a* will have the value of each position of the variable *array1*.

By doing a *writeln(a)* in line 2 of Listing 4.102, it will output the value of each position of the array named *array1*.

Let's see the code generated for that piece of code in Listing 4.103.

```
  li $t0,0
  sw $t0, for_var4
for_loop4:
  lw $t0,for_var4
  li $t1,16
  slt $t0, $t1, $t0
  sltu $t0, $zero, $t0
  xori $t0, $t0, 1
  bne $t0, 1, for_exit4
  lw $t0,for_var4
  lw $t0, array1($t0)
  sw $t0, a
  lw $t0,a
  move $a0, $t0
  li $v0, 1
  jal write
  jal writeln
```

```
18    lw $t1,for_var4
19    li $t2,4
20    add $t1, $t1, $t2
21    sw $t1, for_var4
22    j for_loop4
23  for_exit4:
```

Listing 4.103: Code generated for the LISS code in Listing 4.102

In Listing 4.103, line 1 to 2 we create a variable named *for_var4* which will be used for accesing each index of the array; line 3 we create the branch name relatively to the for-loop statement; line 4 to line 9 we test if the variable *for_var4* is in the limits of the array. If it isn't then it must exit the for-loop statement by going to the branch named *for_exit4* (line 23), otherwise it continues the flow of the execution of the code to line 10. Line 10 to line 12, we refresh the value of the variable *a* by getting the value throw the index (variable *for_var4*) of the array. Line 13 to 17, this is where the content of the block relatively to the for-loop statement that will be put. In this case, it is the code instruction for writing to the output the variable *a*. Line 18 to 21, we refresh the variable *for_var4* to the next position of the array and this is done by summing up with 4 bytes relatively to the old value of the variable *for_var4*. Lastly, line 22, we jumps to the branch *for_loop4* and continue the flow of the execution of the for-loop statement.

In Figure 22, we can see the routine of a while-loop.

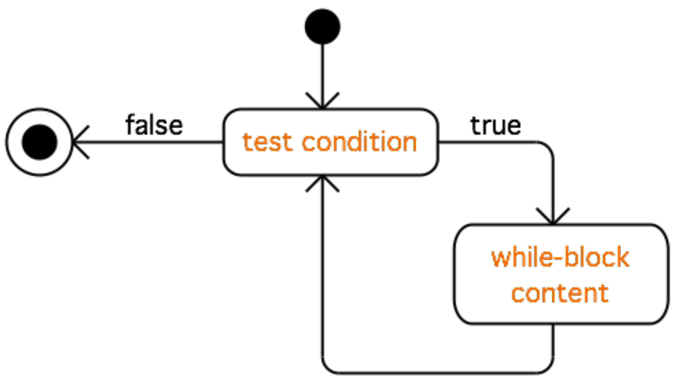


Figure 22.: Schema of the while-loop statement

Basically, behind the idea of being the most simple iterative statement, the behavior of the while-loop statement is to check the truth of the condition every iteraction. If it is true, then the content who is inside will be executed otherwise it will exit the while-loop statement.

Let's see a piece of LISS code relatively to a while-loop statement in Listing 4.104.

```
while(flag){
    writeln("Hello");
```

```
}
```

Listing 4.104: Example of a while-loop statement in LISS

In Listing 4.104, we have a variable *flag* which hold the value true. In this case, the condition is true and then it should proceed to the content available inside of the parentheses. Otherwise if the condition is false, then it will quit the while-loop statement.

Let's see the code generated for that piece of code in Listing 4.104.

```
1 while5:
2   lw $t0,flag
3   bne $t0, 1, while_exit5
4   la $a0, writestring0
5   li $v0, 4
6   jal write
7   jal writeln
8   j while5
9 while_exit5:
```

Listing 4.105: Code generated for the LISS code in Listing 4.104

In Listing 4.105, line 1 is created for the branch name of the while-loop statement. Firstly, we incorporate the code of the condition associated to the while-loop statement (line 2), then we test the truth of the condition in line 3. If the condition is false, it will jump to the branch *while_exit5* (line 9) for exiting the while-loop statement; otherwise, it will continue the flow of the execution by going to line 4. At this part, it is included all the code relatively to the content available in the while-loop statement (line 4 to 7). Finally, at line 8, it will jump back to the branch *while5* and redo the algorithm.

4.4.12 *Creating increment or decrement variables*

If we wanna increment an integer variable, we can use and do by this following way in Listing 4.106.

```
succ i;
```

Listing 4.106: Increment variable in LISS

Basically, we use the word *succ* to give notion of incrementing and then we write the variable which will be incremented (*i*).

Let's see the piece of LISS code relatively to Listing 4.106 in Listing 4.107.

```
lw $t0,i
```

```
2 li $t1,1
3 add $t0, $t0, $t1
4 sw $t0, i
```

Listing 4.107: Code generated for the LISS code in Listing 4.106

In Listing 4.107, line 1 load the variable *i* to register; line 2 load the value 1 to register; line 3 add both last registers (value of the variable and the value 1); line 4 refresh and store the result to the variable *i*.

If we wanna decrement an integer variable, we can use and do by this following way in Listing 4.108.

```
pred i;
```

Listing 4.108: Decrement variable in LISS

We use the word *pred* for decrementing the variable and then we write the variable which will be drecremented (*i*).

Let's see the piece of LISS code relatively to Listing 4.108 in Listing 4.109.

```
1 lw $t0,i
2 li $t1,1
3 sub $t0, $t0, $t1
4 sw $t0, i
```

Listing 4.109: Code generated for the LISS code in Listing 4.109

In Listing 4.109, line 1 load the variable *i* to register; line 2 load the value 1 to register; line 3 subtract both last registers (value of the variable and the value 1); line 4 refresh and store the result to the variable *i*.

5

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 or interpreter, a debugger, and other tools. IDEs are designed for maximizing the productivity of programmers with visual interface (see Figure 23).

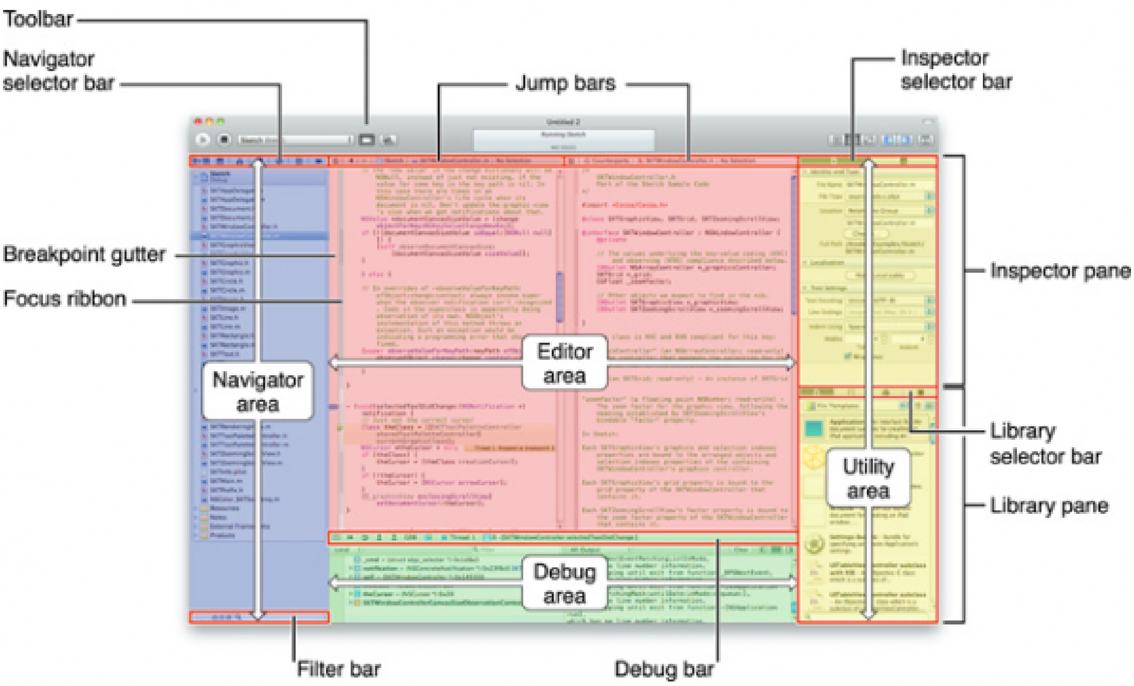


Figure 23.: Example of an IDE visual interface (XCode) [1]

Programs are created top down in the editor window 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 debug 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 guided by the language syntax.

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. 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.

5.1 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 value (word, number, or string) or another template and they are generated by editor commands, according to the grammar production.

```
IF( condition )
 THEN statement
 ELSE statement
```

Listing 5.1: Example of a IF Conditional template

In Listing 5.1 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 other templates or by concrete values (numeric or string constants or identifiers).

To clarify the definition of SDE, we will explain it with the help of an example.

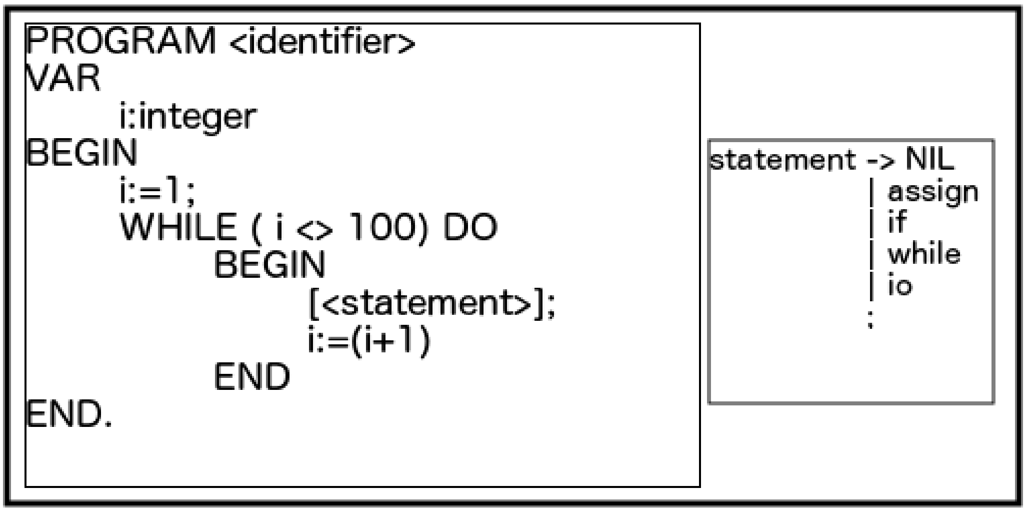


Figure 24.: SDE example

Figure 24 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 template 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 24 is placed at the placeholder corresponding to a *statement*; at the same time, the right box was updated to show all the possible next templates according to the *statement* derivation rules (RHS).

To sum up, this is how a SDE works.

5.2 Conception of the SDE

By taking the ideas explained in the previous section, we managed to create a simple and easy to use Syntax Directed Editor based on the principles:

- having a window for visualization of the rules of the language grammar and the templates associated to the "LISS" program under development.

- showing the source code produced until the moment by choosing templates and fulfilling placeholder according to the rules generated.
- displaying semantic or runtime errors, and outputing the results of the execution.

These guidelines led the creation of the program called *liss|SDE* (see Figure 25).

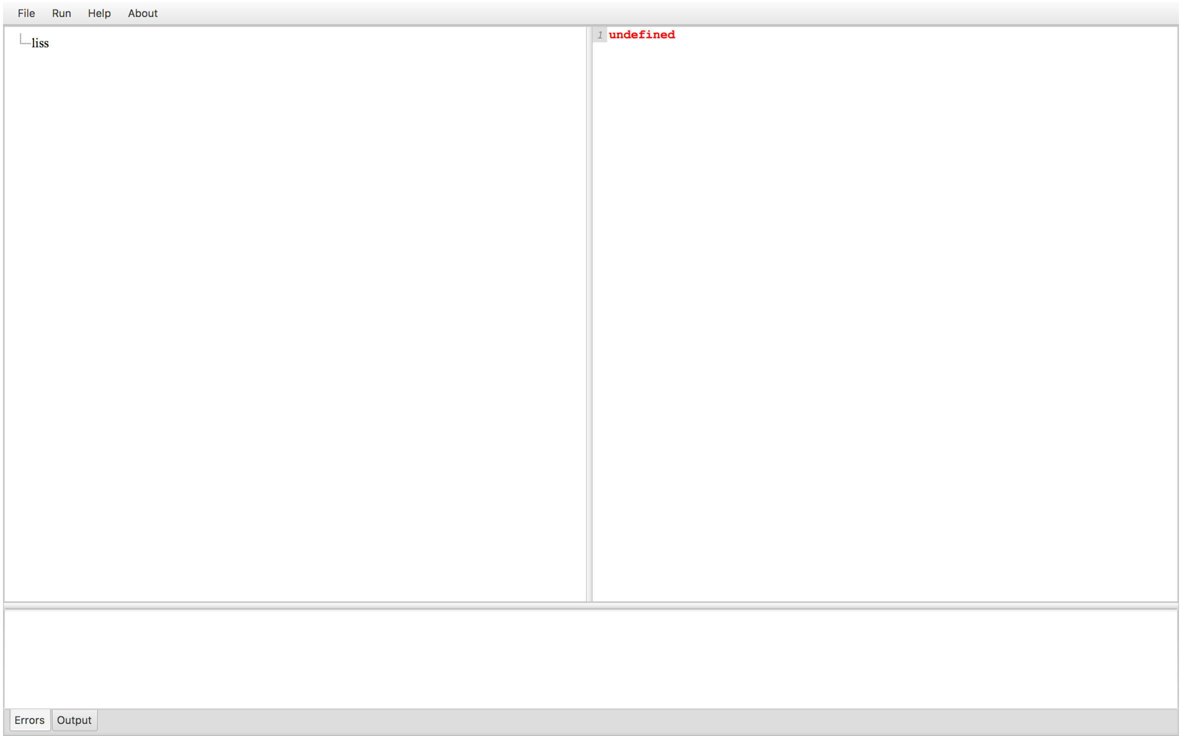


Figure 25.: liss|SDE

liss|SDE system interface is divided in three main areas (see Figure 26).

The number 1, in Figure 26, is where the templates, or the rules of the language LISS, are displayed. This part was implemented with the technology called HTML and Javascript, and the main reason of implementing that part with that technology was the fact that we needed to implement a tree visualization structure.

Creating some visualization content within the JAVA context is really hard. For that reason we decided to use another technology where JAVA could handle it; HTML and Javascript were the perfect key for creating those contents, due to their powerful and easy use to create some visualization content.

Also, notice that, we implemented a tree visualization structure for one simple reason: a programming language is represented by a tree structure. So we decided to adapt it and create a tree visualization structure.

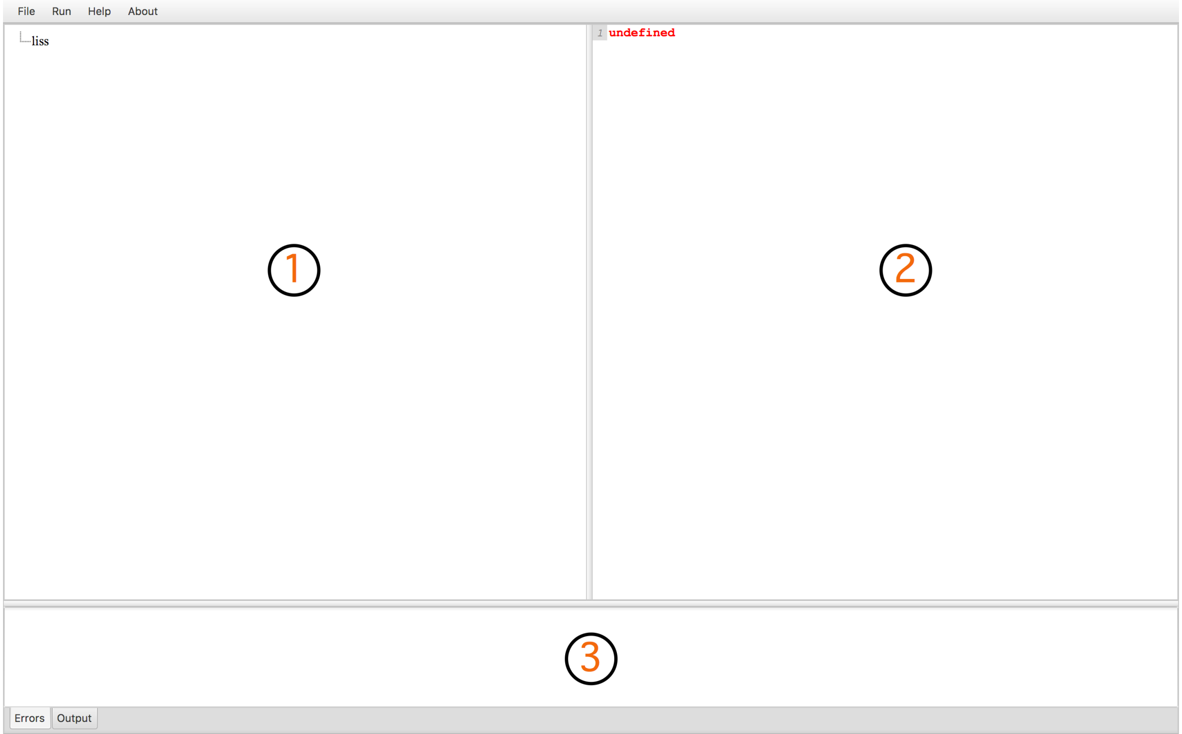


Figure 26.: liss|SDE structure

The number 2, in Figure 26, is where the code of the language is processed regarding to the rules generated to the view number 1. Each time, the user selects a rule, the view in the window number 2 will be represented to reflect the new program code synthesized.

Lastly, the window number 3 is related to every syntax and semantic errors, as well as, the output of the execution.

5.2.1 *Toolbar meaning*

The toolbar in *liss|SDE* is available at the top of the program and it control various functions of the program.

It is divided in four boxes (see Figure 27).

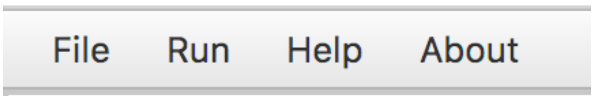


Figure 27.: Toolbar of *liss|SDE*

The *File* button shows the commands for creating new projects, saving/loading projects and exiting the application (see Figure 28).

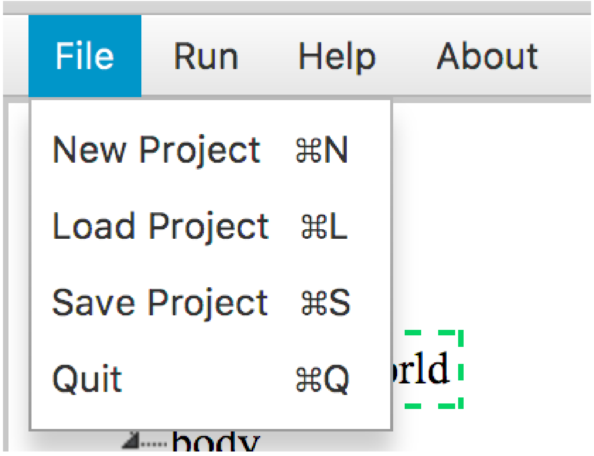


Figure 28.: File option of toolbar in *liss|SDE*

The *Run* button activates the compiler (this means that the semantic system will be run and if everything is correct, the code is generated, getting the MIPS assembly code) and executes the code created (see Figure 29).

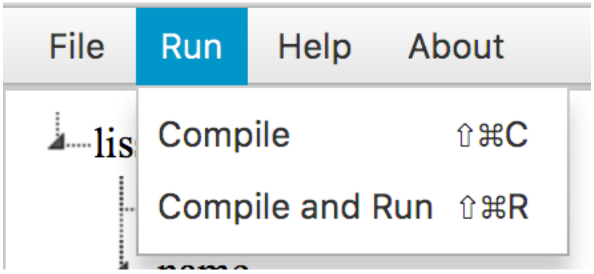


Figure 29.: Run option of toolbar in *liss|SDE*

The *Help* button shows information about the use of the editor and about a LISS program structure (see Figure 30).

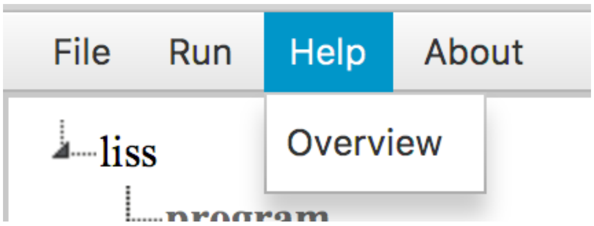


Figure 30.: Help option of toolbar in *liss|SDE*

About button displays information about the program: which technology and plugins were used; the name of the program creator; etc.. (see Figure 31).

Notice that some shortcuts for the most used functions were implemented in the toolbar, as can be seen in Figures 28 and 29.

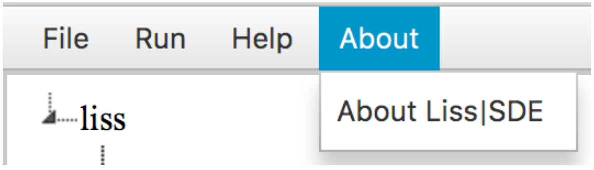


Figure 31.: About option of toolbar in *liss|SDE*

5.2.2 *Creating a program*

For a better understanding of this section, let's see a simple piece of LISS code in Listing 5.2.

```
program helloWorld{
  declarations
  statements
    writeln("Hello World!");
}
```

Listing 5.2: LISS code

In Listing 5.2, we created a 'hello world' program in LISS which basically outputs the string "Hello World!".

Now, let's try and create the same LISS program using *liss|SDE*.

In Figure 32, we left click on the non-terminal *liss*; this selects the *liss* rule and expands the non-terminal with three branches:

- program
- name
- body

program is a terminal of type keyword: this can be seen by its bold and blurry visual; the other branches are non-terminal (*name* and *body*) and they are not bolded nor blurried. Those visual effects are really important for the user because it means that terminals aren't clickable and non-terminal are clickable.

Notice, also, that by selecting the *liss* rule, the right window is refreshed with the code that is possible to synthesize until that moment the keyword is printed correctly and the label 'undefined' is displayed in red for each non-terminal not yet expanded. This means that the non-terminal must be selected and derived and that the code isn't valid in this state.

Figure 32.: Creating a LISS program (1/17)

So, in this case, we need to expand those two non-terminal rules and we will do it by clicking with the left mouse button on the *name* branch. This generates a rectanglular box with a name inside (*IDENTIFIER*), see Figure 33.

Basically, a rectangle as the one shown in Figure 33 means that this is a placeholder of that kind "input interaction" and the label *IDENTIFIER* informs the kind of value that must be typed.

In this case, the *IDENTIFIER* label specifies that it must be typed a text matching the following pattern:

```
('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
```

Figure 33.: Creating a LISS program (2/17)

If the text typed in by the user does not match the pattern, then the box will change the color to red (see Figure 34).

Red color means that there is a lexical error in the input; instead, green color means that the input is correct.

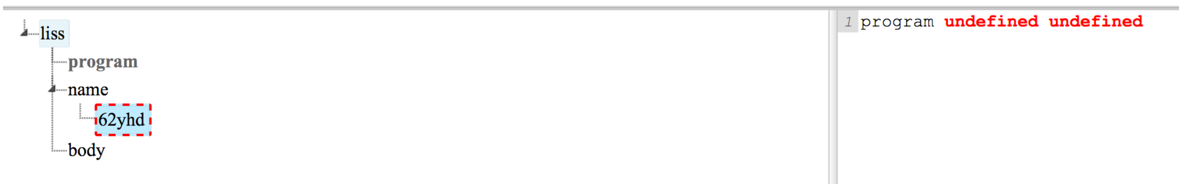


Figure 34.: Creating a LISS program (3/17)

If the color of the box is green (see Figure 35) then the input is correct and the *undefined* word seen in the right window previously (see Figure 34), will change to the value of the input.

Now, let's proceed with the other branch *body* by clicking it with the left mouse button.

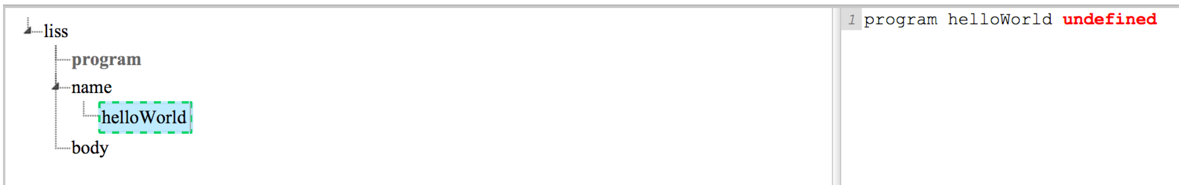


Figure 35.: Creating a LISS program (4/17)

As can be seen, in Figure 36, this non-terminal symbol expands to more rules and notice, that in the right windows, the code is changed. The next step is always the same, generating the rules in order to synthesize a correct program (no *undefined* must be displayed in the right window).

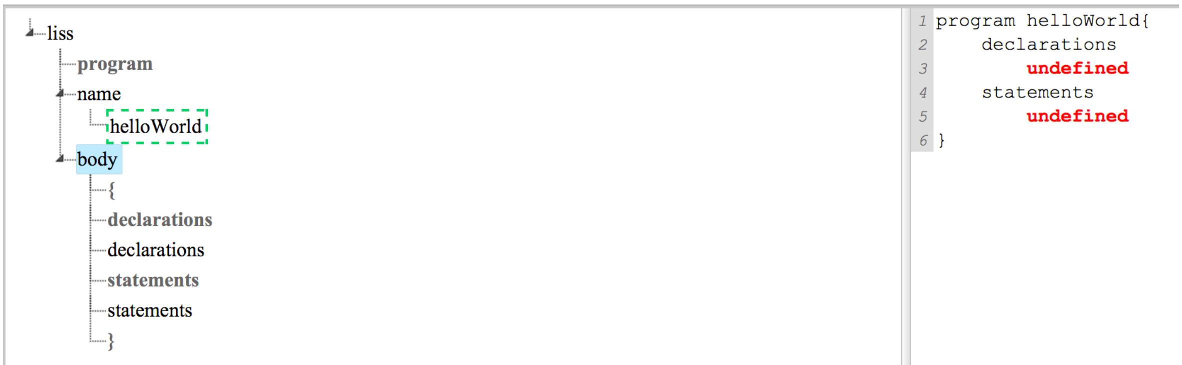


Figure 36.: Creating a LISS program (5/17)

By clicking with the left mouse button over the rule called *declarations* in Figure 36, it will generate more branches (see Figure 37). Notice that, in the right window, the first *undefined* word disappeared and that in the left window, two branches were created (the two possible kind of *declarations*) with a star at the end of their names. That star means that the non-terminal symbol can be expanded to zero or more elements.

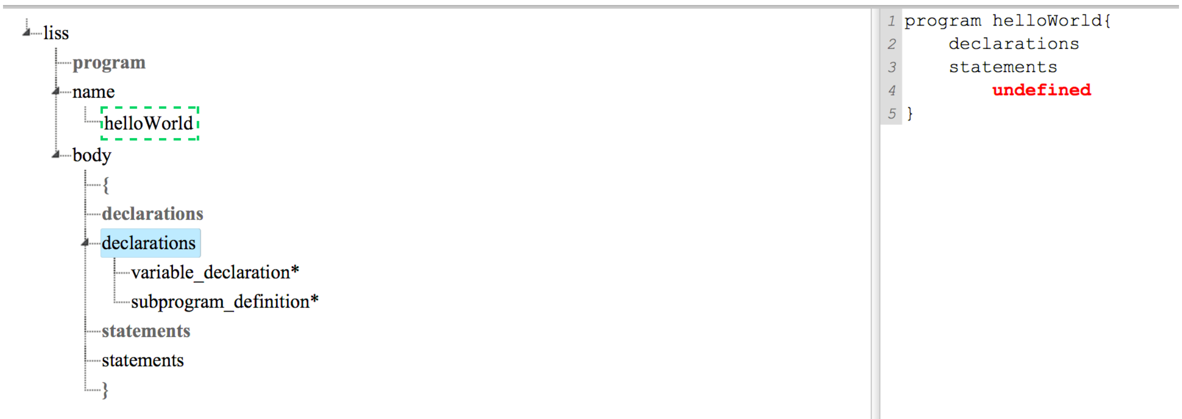


Figure 37.: Creating a LISS program (6/17)

The program created until this stage is similar to the first part of the one in Listing 5.2; let's now work out the rules for the *statements* part by left clicking on that symbol (see Figure 38).

At this moment, no *undefined* label is displayed in the right window which means that the code can be compiled. But the problem is that it is not yet finished relatively to the program in Listing 5.2.

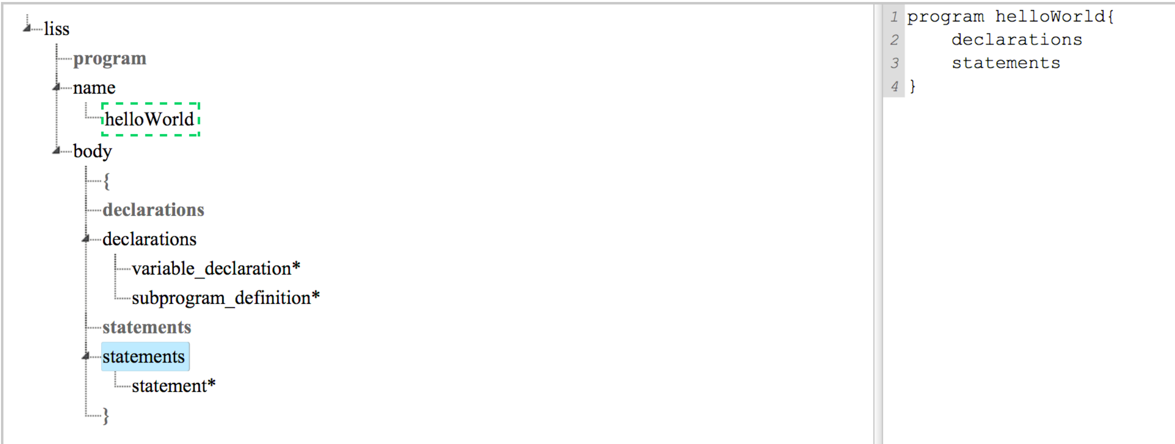


Figure 38.: Creating a LISS program (7/17)

So, we need to expand the *statement** rule by left clicking it. Then a pop-up menu will appear (see Figure 39).

This pop-up menu (that exhibits just one option "Add" at the beginning) is available for one reason: if the user creates a lot of branches under the *statement**, after the second branch created that pop-up box will show another option for deleting all the branches created. And this is why we needed to create a menu for adding those two options (thinking on the easy use of the program).

Each time a statement is added, it will be appended to the previous ones, at the bottom place of the program.

Regarding the code in Listing 5.2, we just need to create one statement (writeln statement); so we only need to add one statement to the program (see Figure 40).

By adding that statement, we see that in the right window, the program becomes incorrect (an *undefined* label appears aggain).

By left clicking on the *statement* rule, a pop-up menu appears and we can see the rules that a *statement* can expand to (see Figure 41).

So, left click on the option of adding the write statement to expand to the appropriate rule (see Figure 42).

After left clicking over the *write* statement rule, the content of that rule is generated and the right window is refreshed accordingly (see Figure 43).

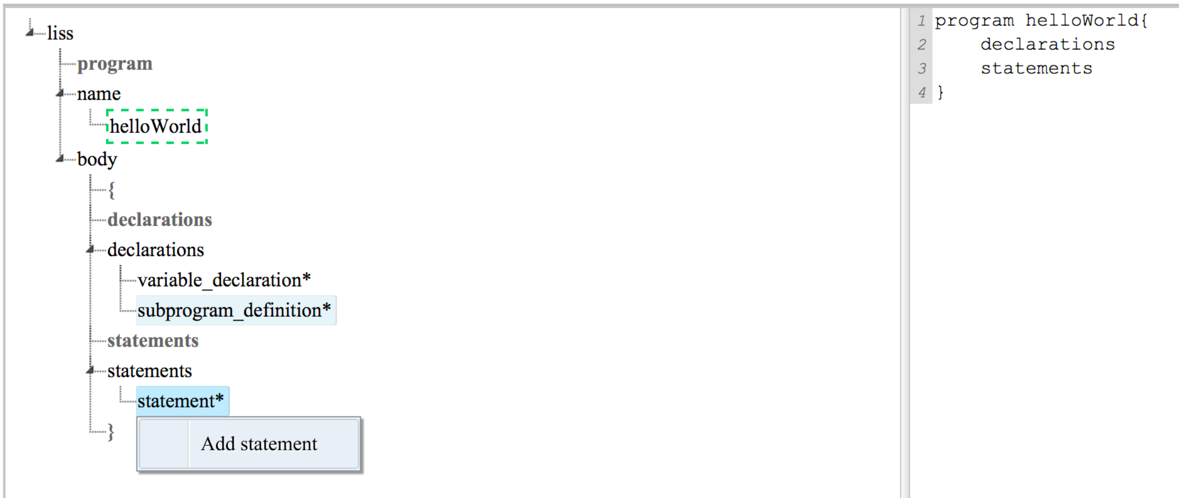


Figure 39.: Creating a LISS program (8/17)

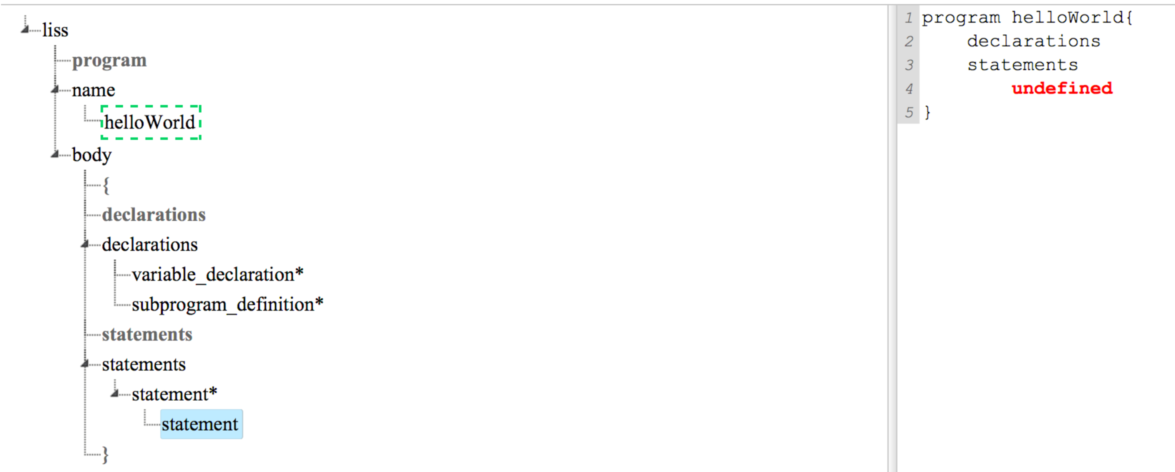


Figure 40.: Creating a LISS program (9/17)

First it is necessary to expand the *write_expr* rule by left clicking on it; it will open another pop-up menu to choose the desired output statment (Figure 44).

To clone the program in Listing 5.2, we must choose the *Writeln* option.

So, left clicking on that option, the right part will be refreshed accordingly to the terminal keyword. Then it is needed to process the rule *print_what* for finishing the code (see Figure 45).

The *print_what* non-terminal shows a pop-up menu with three options (see Figure 46); for our case, we need to choose the option of adding a string.

By clicking on that option, a rectangle with the label **STRING** appears below (see Figure 47), following the same idea talked previously for the *IDENTIFIER* box. The only thing that differs is the pattern that now is:

```
'"' ( ESC_SEQ | ~('"') )* '"'
```

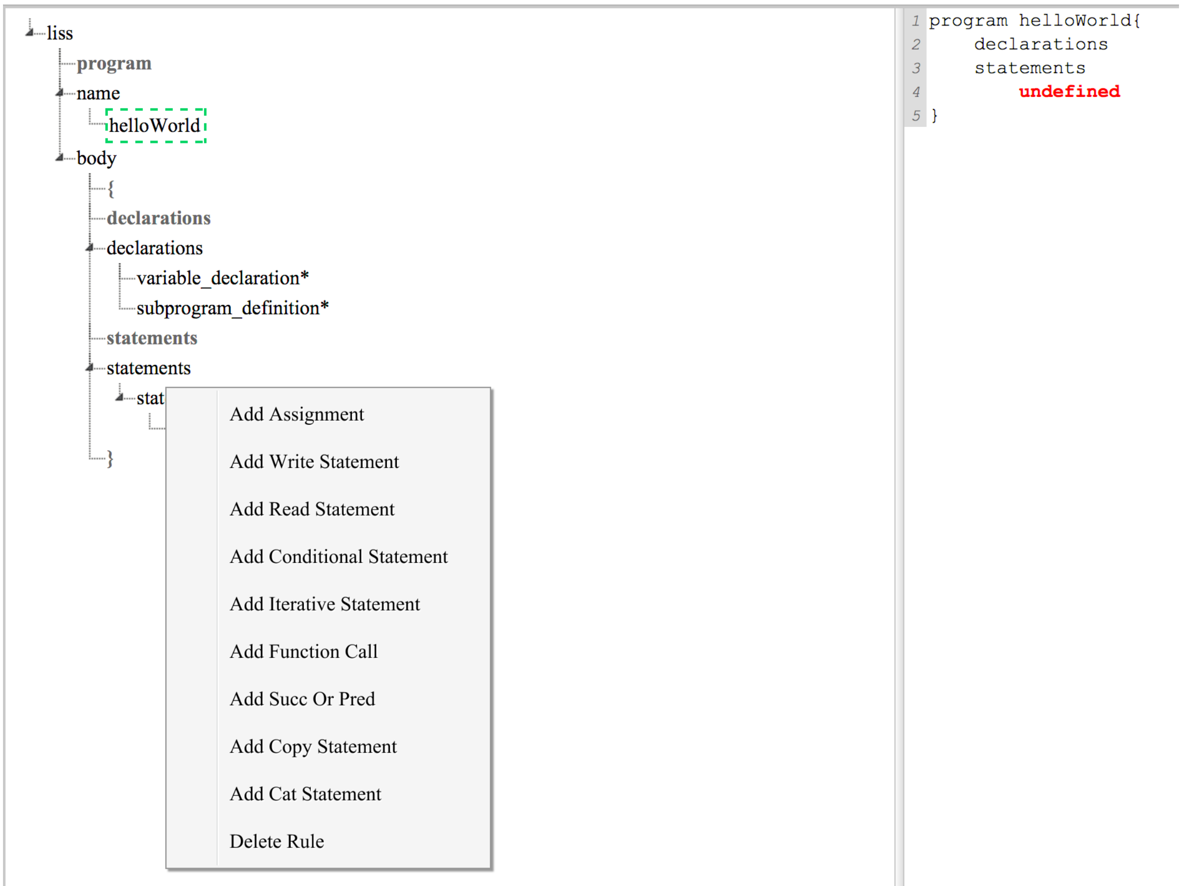


Figure 41.: Creating a LISS program (10/17)

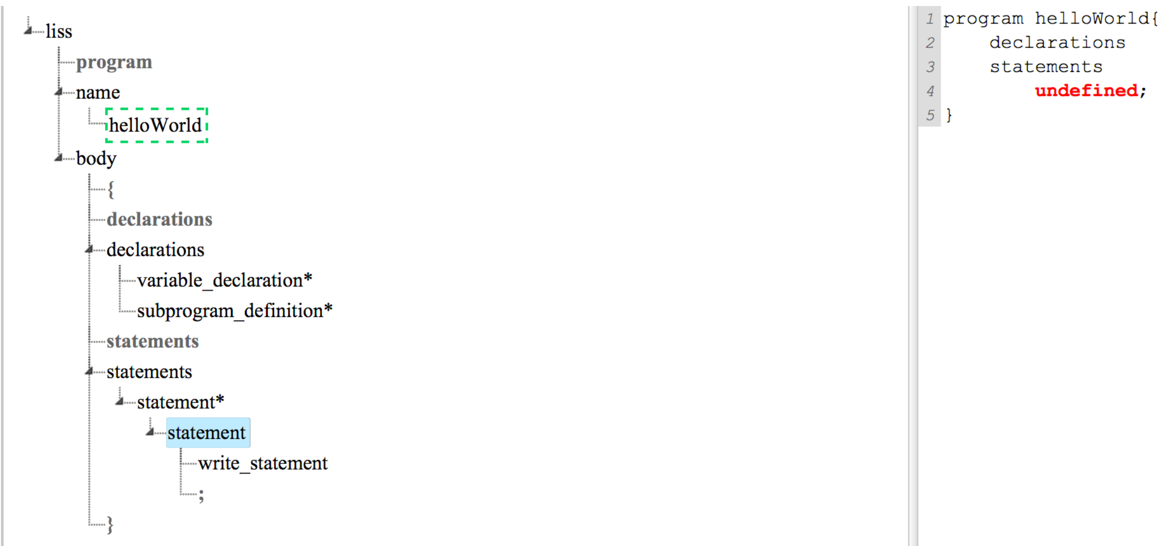


Figure 42.: Creating a LISS program (11/17)

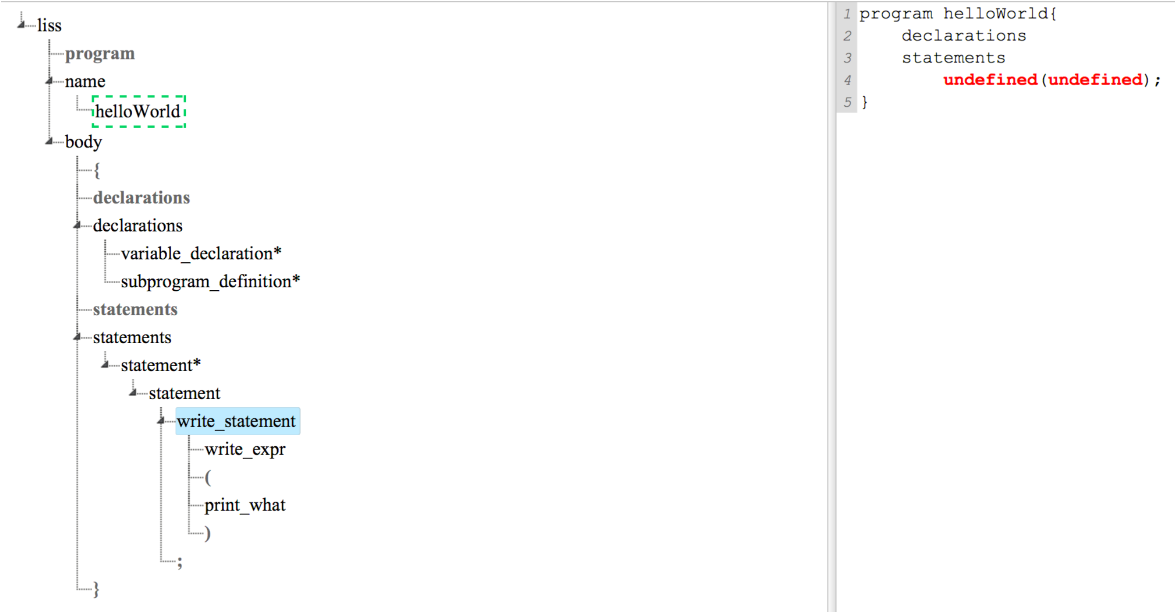


Figure 43.: Creating a LISS program (12/17)

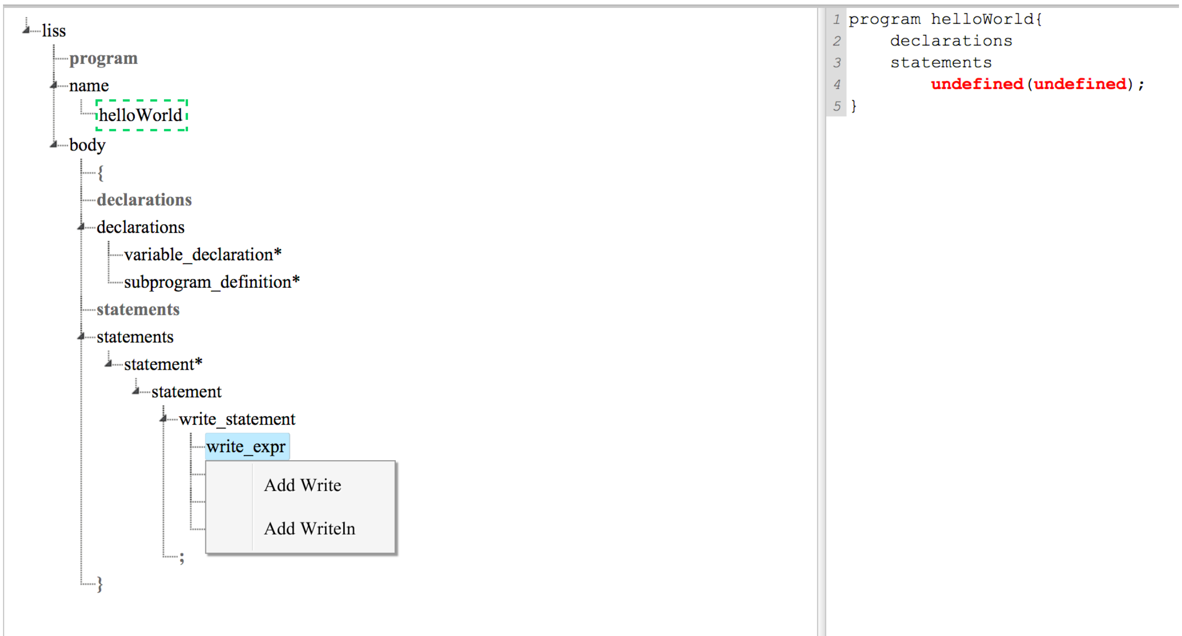


Figure 44.: Creating a LISS program (13/17)

Basically the idea of that pattern is that the string must be a sequence of any characters inside quotation marks.

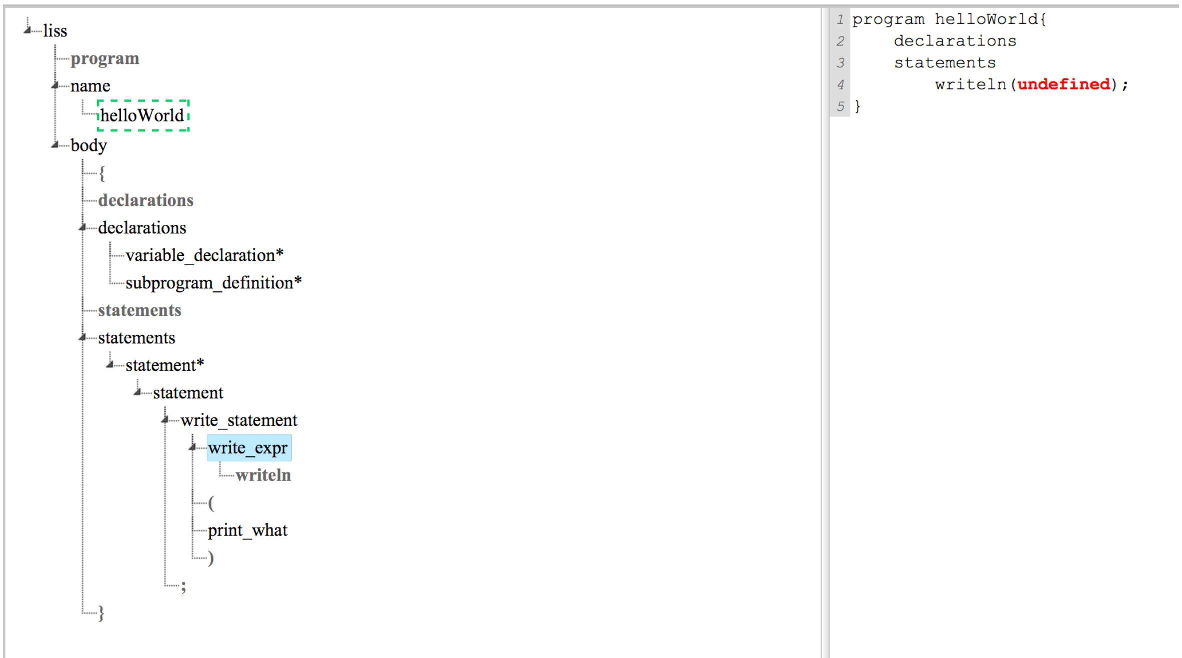


Figure 45.: Creating a LISS program (14/17)

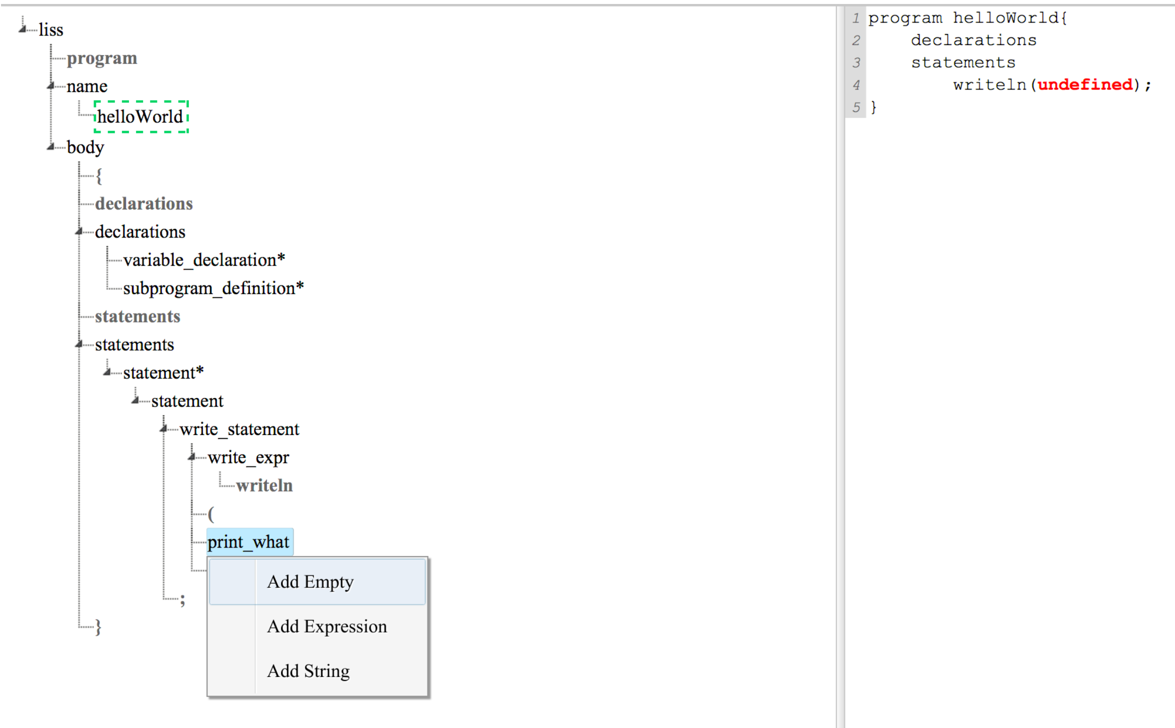


Figure 46.: Creating a LISS program (15/17)

And so, for the last step, we manage to write the string *"Hello World!"* in that rectangular box for finishing the creation of the program in Listing 5.2.

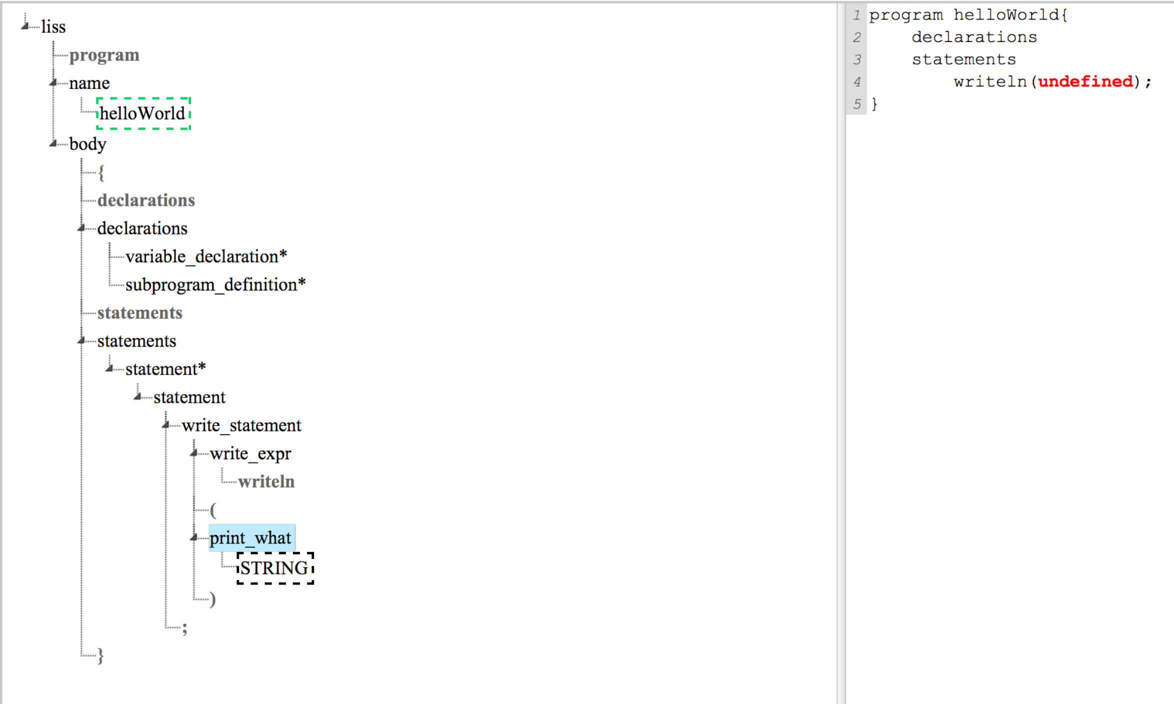


Figure 47.: Creating a LISS program (16/17)

Notice that in the right window of Figure 48, no *undefined* label is shown, what means that the code can be compiled and executed.

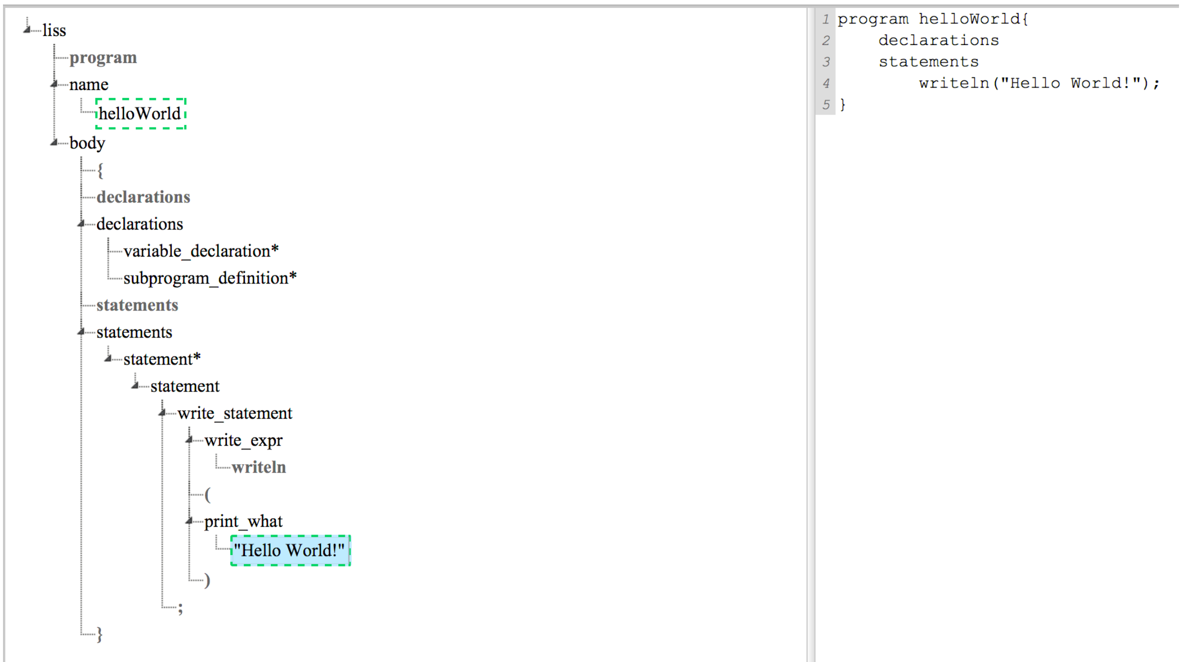


Figure 48.: Creating a LISS program (17/17)

Finally, notice that the user can at any time delete rules by clicking on every non-terminal in the tree structure.

And this is how the user can interact and create a program in *liss|SDE*.

5.2.3 *Executing a program*

For executing a program in *liss|SDE*, we just need to go to the toolbar, press the *Run* button and then choose the *Compile and Run* option (remember Figure 29).

By doing that, it will pass throw a lot of steps (see Figure 49).

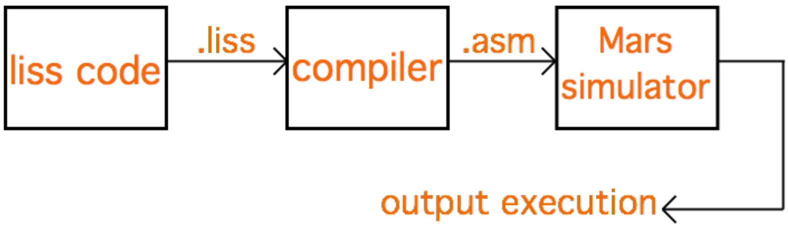


Figure 49.: Flow of the execution of a liss code in the *liss|SDE*

The first step is to take the liss program and pass it to the compiler. In this moment, the compiler will check the consistency of the code (semantic system); then, if everything is fine, it will pass the MIPS assembly code generated (at the end of the process of the compiler) to the simulator (Mars Simulator) and execute the code. Lastly and finally, it will print the output of the execution of the liss code to the window number 3 (see Figure 26).

For a visual example, let's execute the program created above with the *liss|SDE* tool.

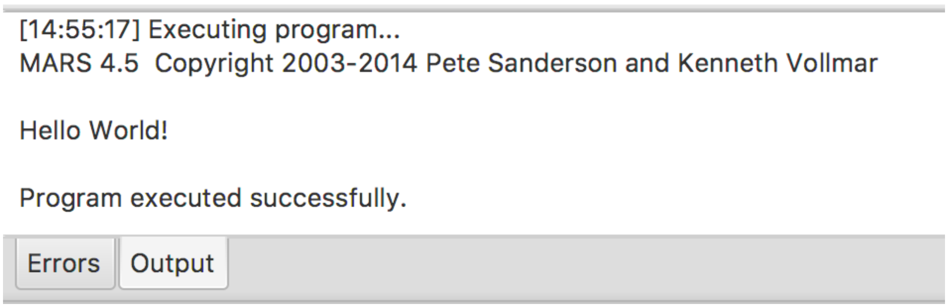


Figure 50.: Output of the execution of the *HelloWorld* program

In Figure 50, we can see that the string *"Hello World!"* is printed and that the program was executed and terminates successfully.

5.2.4 *Error System in liss|SDE*

Each time a LISS program is compiled, an error table is built. All the semantic error that are found in the LISS program, will be added to that error table.

If the error table is empty, then the code can be executed. Otherwise, if the error table contains some errors, then those errors must be outputed in the tab *Errors* at the window number 3 (see Figure 26).

Let's see an example of the error system in *liss|SDE*.

Consider the program below created with *liss|SDE* environment:

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

Listing 5.3: Example of a liss code that isn't semantically correct

Now, if we try to run the compiler in *liss|SDE*, the compiler will throw error messages due to the inconsistencies that were found (see in Listing 5.4).

```
[16:10:40] Semantics errors found:
[16:10:40] line: 3:8  Variable 'int'  has type 'integer',when It should
    be 'boolean'.
[16:10:40] line: 5:16  Expression 'int + 3'  has type 'boolean +
   integer',required type 'integer + integer'.
[16:10:40] line: 5:16  Expression 'int+3'  has type 'null',when It
   should be 'integer | boolean | sequence | array'.
```

Listing 5.4: Error messages in *liss|SDE*

The notation conventions used for those messages, in Listing 5.4, is as follows: firstly, is shown the time that the error was found (embraced by square brackets); secondly, the line number of the error regarding to the liss code in window number 2 (see Figure 26) and lastly, the error message text.

Notice that the line number is very important to locate and understand the error and correct it.

6

CONCLUSION

In this final chapter, it will be summarized the information that was exposed throughout this document in order to remember the aims of the project and how they were achieved.

First, it begins by contextualizing the reasons of helping the developers for being more productive regarding the creation of some programs for computers. This was attained by creating a software system (a compiler) which makes the life of a developer easier by allowing him to write programs at a high level of productivity. But as always there is the necessity of achieving more (creating a program with a high level language and in a safety way) and that is why, we introduce the notion of Syntax Directed Editor, an editor that helps the programmer writing his code guiding him through the language syntax.

The main idea underlying of a SDE is to create programs whithout the use of the keyboard avoiding syntax errors; instead, the programmer will use the mouse and create the code by selecting some rules available in the grammar of the language. For this project, it was aimed to create a SDE for the language called LISS.

So, it was needed to first understand that language, writting some test programs. After understanding the syntax of the language, some improvements were introduced in the language (due to its old age) by changing or deleting some rules.

After designing those improvements in the grammar, it was necessary to develop the compiler to analyse and check the LISS programs and generate the associated assembly code. For that purpose, the original grammar (a CFG - Context-Free Grammar) was evolved to another type (an AG - Attribute Grammar).

Before creating the compiler, it was needed to understand the MIPS architecture which is the chosen target machine (the compiler will generate MIPS assembly code). The architecture is a RISC architecture which makes the learning phase easier. After understanding the architecture, it was necessary to test it, writting some programs in MIPS assembly code for a better view of that programming language.

With those knowledge acquired relatively to the MIPS architecture, it was essential to create some adapted data structures (stack, registers) to support the code generation and the execution of LISS program.

As a result of all the research and reasoning made, we began by creating the compiler. It is important to remember that the compiler only pass once throw the LISS code which has some pros and cons. In that single pass, two tasks are performed by the compiler's back-end; the semantic analysis and the code generation.

The semantic analysis requires some structures that are a symbol table and an error table. The symbol table save information about all the LISS identifiers; the error table stores the error messages to inform the user relatively to the semantic analysis (both of those structures are in a sense connected). The symbol table is also used to support the most complex stage in the project, the code generation.

The code generation rose up a lot of difficulties and this is due to some hard issues listed below:

- processing the LISS code only once and caring about the order that it is processing relatively for generating the code.
- saving some informations in the memory relatively for generating the code in the correct order (due to the specifications of the MIPS architecture).
- creating a linked-list in MIPS assembly code to implement the sequence type.
- creating MIPS code with alignment address.
- defining a certain architecture for generating the code relatively to the context that the compiler is dealing with (using which register, stack or heap).
- creating some complex algorithm for the use of some structure available in the project.

After developing the compiler there is the need of testing it and checking the correctness of the code generated. 18 LISS programs prepared specifically to cover the different types and statements that were tested and approved.

Once the compiler was created, it was built the visual (Syntax Directed Editor - SDE) program. By studying and reading some articles about the concept of SDE, we managed to draw the interface and its visual appearance, focusing on the most important notions of such an editor:

- having a visualization of the abstract syntax tree for the LISS language.
- having a window where the LISS code being created can be seen.
- having an output window where the error messages found at runtime by the compiler are displayed as well as the ouput produced by the execution.

After developing the visual editor and integrating it with the LISS compiler, it was needed to connect the compiler and the MARS simulator. This task for connecting the MARS simulator with the SDE environment, was one of the hardest parts of the project due to the complexity of using processes and threads in JAVA for sending some input and getting the output.

The last step was the final, testing of the SDE environment to check if everything works properly.

It was made an inquiry for testing the application and learn more about the usability of the syntax direct editor. It was concluded that this concept is a must for learning a new programming language due to the fact of getting a better perception of the syntax of the programming language and the easy way for creating some code. However when the user has better and deeper knowledge of the programming language, that approach is not the most appropriated.

6.1 FUTURE WORK

To conclude that dissertation, it is intended to discuss some future work that might be done for improving the application concerning the productivity of the programmer and the compiler efficiency.

- Adding some semantic to the abstract syntax tree available in the visual for generating the rules. This means that if we declare a variable where it is initialized with an integer value, it should know that the variable will be an integer type. In this case, we could create a SSDE (Semantic Syntax Directed Editor).
- Optimizing the code generated by the compiler.
- Adding the possibility of moving statements in the abstract syntax tree. For instance, if you want to swap two statements, you cannot do it.
- Adding a feature for incremental compilation.
- Adding a feature to the editor (SDE) to be adaptable to any programming language who deal with other programs not written in LISS.

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 ';'
                     ;

```

```
vars : var (',' var )*
     ;

var : identifier value_var
    ;

value_var :
          | '=' inic_var
          ;

type : 'integer'
     | 'boolean'
     | 'set'
     | 'sequence'
     | 'array' 'size' dimension
     ;

typeReturnSubProgram : 'integer'
                     | 'boolean'
                     ;

dimension : number (',' number )*
          ;

inic_var : constant
         | array_definition
         | set_definition
         | sequence_definition
         ;

constant : sign number
         | 'true'
         | 'false'
         ;

sign :
     | '+'
     | '-'
     ;

/* ****** Array definition ****** */

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

```
                    ;

array_initialization : elem (',' elem)*
                        ;

elem : number
      | array_definition
      ;

/* ****** Sequence definition ****** */

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

sequence_initialization :
                            | values
                            ;

values : number (',' number )*
        ;

/* ****** Set definition ****** */

set_definition : '{' set_initialization '}'
                 ;

set_initialization :
                      | identifier '|' expression
                      ;

/* ****** SubProgram definition ****** */

subprogram_definition: 'subprogram' identifier '(' formal_args ')'
    return_type f_body
                        ;

f_body : '{'
          'declarations' declarations
          'statements' statements
          returnSubPrg
          '}'
        ;

```

```
/* ****** Formal args ****** */

formal_args :
            | f_args
            ;

f_args   : formal_arg (',' formal_arg )*
         ;

formal_arg : identifier '->' type
           ;

/* ****** Return type ****** */

return_type :
            | '->' typeReturnSubProgram
            ;

/* ****** Return ****** */

returnSubPrg :
             | 'return' expression ';'
             ;

/* ****** Statements ****** */

statements : statement*
           ;

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

/* ****** Assignment ****** */

assignment : designator '=' expression
```

```
            ;

/* ****** Designator ****** */

designator : identifier array_access
            ;

array_access :
              | '[' elem_array ']'
              ;

elem_array : single_expression (',' single_expression )*
            ;

/* ****** Function call ****** */

function_call : identifier '(' sub_prg_args ')'
               ;

sub_prg_args :
              | args
              ;

args : expression (',' expression )*
     ;

/* ****** Expression ****** */

expression : single_expression ( rel_op single_expression )?
            ;

/* ****** Single expression ****** */

single_expression : term ( add_op term )*
                   ;

/* ****** Term ****** */
term : factor ( mul_op factor )*
     ;

/* ****** Factor ****** */

factor : inic_var
```

```
        | designator
        | '(' expression ')'
        | '!' factor
        | function_call
        | specialFunctions
        ;

specialFunctions : tail
                 | head
                 | cons
                 | member
                 | is_empty
                 | length
                 | delete
                 ;

/* ****** add_op, mul_op, rel_op ****** */

add_op : '+'
       | '-'
       | '||'
       | '++'
       ;

mul_op : '*'
       | '/'
       | '&&'
       | '**'
       ;

rel_op : '=='
       | '!='
       | '<'
       | '>'
       | '<='
       | '>='
       | 'in'
       ;

/* ****** Write statement ****** */

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

```
write_expr : 'write'
           | 'writeln'
           ;

print_what :
           | expression
           ;

/* ****** Read statement ****** */

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

/* ****** Conditional & Iterative ****** */

conditional_statement : if_then_else_stat
                      ;

iterative_statement : for_stat
                    | while_stat
                    ;

/* ****** if_then_else_stat ****** */

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

else_expression :
                | 'else' '{' statements '}'
                ;

/* ****** for_stat ****** */

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
        ;

/* ****** While_Stat ****** */
while_stat : 'while' '(' expression ')'
             '{' statements '}'
           ;

/* ****** Succ_Or_Predd ****** */

succ_or_pred : succ_pred identifier
             ;

succ_pred : 'succ'
          | 'pred'
          ;

/* ****** SequenceOper ****** */

```

```
tail // tail : sequence -> sequence
      : 'tail' '(' expression ')'
      ;

head // head : sequence -> integer
      : 'head' '(' expression ')'
      ;

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

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

copy_statement // copy_statement : seq x seq -> void
                : 'copy' '(' identifier ',' identifier ')'
                ;

cat_statement //cat_statement : seq x seq -> void
               : 'cat' '(' identifier ',' identifier ')'
               ;

is_empty // is_empty : sequence -> boolean
          : 'isEmpty' '(' expression ')'
          ;

length // length : sequence -> integer
        : 'length' '(' expression ')'
        ;

/* ****** set_oper ****** */

member // isMember : integer x sequence -> boolean
        : 'isMember' '(' expression ',' identifier ')'
        ;



/*
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     */
```

```

string  :  STR
        ;

number  :  NBR
        ;

identifier  :  ID
            ;
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 */


/* ****** Lexer ****** */

NBR  :  ('0'..'9')+
     ;

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

WS   :    ( [ \t\r\n] | COMMENT) -> skip
     ;

STR  :   '"' ( ESC_SEQ | ~('"') )* '"'
     ;


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

fragment
ESC_SEQ
     :    '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
     ;
```

lissGIC.g4