Damien da Silva Vaz

Implementing a Syntax Directed Editor for LISS.

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Damien da Silva Vaz

Implementing a Syntax Directed Editor for LISS.

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor Pedro Rangel Henriques
Professor Daniela da Cruz

September 2016

ACKNOWLEDGEMENTS

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

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

INTRODUCTION

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

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

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

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

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

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

1.1 OBJECTIVES

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

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

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

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

1.2 RESEARCH HYPOTHESIS

1.3 THESIS OUTCOMES

1.4 DOCUMENT STRUCTURE

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

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

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

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

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

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

2

LANGUAGES AND GRAMMAR: CONCEPT & TOOLS

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

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

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

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

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

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

where each side is composed by a sequence of symbols.

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

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

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

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

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

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

Figure 1.: CFG example [1]

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

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

```
1 facturas : fatura (facturas)*
2           ;
3
4 fatura : 'FATURA' cabec 'VENDAS' corpo {System.out.println("Total Factura
      : "+$corpo.totOut);}
5         ;
6
7 cabec : idFat idForn 'CLIENTE' idClie {System.out.println("Factura n: "+
      $idFat.text);}
8        ;
9
10 idFat : numFat ;
11
12 numFat : ID ;
13
14 idForn : nome morada 'NIF:' nif 'NIB:' nib
15        ;
16
17 idClie : nome morada 'NIF:' nif
18        ;
19
20 nome :   STR ;
```

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

Listing 2.1: Example of an AG

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

2.1 FORMAL GRAMMAR

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

- A finite set N of nonterminals symbols.

- A finite set $\sum$ of terminals symbols.

- A finite set P of production rules.

- A start symbol S ∈ P

A grammar is formally constructed by that tuple (N,∑,P,S).

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

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

Listing 2.2: A rule production

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

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

LISS LANGUAGE

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

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

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

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

3.1 LISS DATA TYPES

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

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

Let's obverse a LISS code example:

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

Listing 3.1: Declaring a variable in LISS

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

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

```
1   variable_declaration : vars '->' type ';'
2                   ;
3   vars : var (',' var )*
4       ;
5   var : identifier value_var
6     ;
7   value_var :
8           | '=' inic_var
9           ;
10  type : 'integer'
11      | 'boolean'
12      | 'set'
13      | 'sequence'
14      | 'array' 'size' dimension
15      ;
16  dimension : number (',' number )*
17          ;
18  inic_var : constant
19         | array_definition
20         | set_definition
21         | sequence_definition
22         ;
23  constant : sign number
24         | 'true'
25         | 'false'
26         ;
27  sign :
28      | '+'
29      | '-'
30      ;
```

Listing 3.2: CFG for declaring a variable in LISS

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

Table 1.: LISS data types

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

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

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

Listing 3.3: Initialize a variable

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

Table 2.: Operations and signatures in LISS

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

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

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

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

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

The operations defined over arrays are:

1. *indexing*

2. *assignment*

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

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

Thi is equivalent to the initialization below:

```
1    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.

```
1    array_definition : '[' array_initialization ']'
2                      ;
3
4    array_initialization : elem (',' elem)*
5                         ;
6
7    elem : number
```

```
8        |  array_definition
9        ;
```

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

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

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

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

The operations defined for sets are :

1. *union*

2. *intersection*

3. *in* (membership)

Let's see an example of its syntax below:

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

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

The syntax for set declaration and initialization is :

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

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

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

Listing 3.4: Example of valid operations using sequence on LISS

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

1. *tail* (all the elements but the first)

2. *head* (the first element of the sequence)

3. *cons* (adds an element in the head of the sequence)

4. *delete* (remove a given element from the sequence)

5. *copy* (copies all the elements to another sequence)

6. *cat* (concatenates the second sequence at the end of the first sequence)

7. *isEmpty* (true if the sequence is empty)

8. *length* (number of elements of the sequence)

9. *isMember* (true if the number is an element of the sequence)

Those operations will be explained further and deeper.

The grammar below defines how to declare a sequence:

```
1   sequence_definition : '<<' sequence_initialization '>>'
2                       ;
3
4   sequence_initialization :
5                       | values
6                       ;
7
8   values : number ( ',' number )*
9       ;
```

3.1.1 *LISS lexical conventions*

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

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

```
1   program single_variable_name{
2     declarations
3       int=1 -> integer;
4       int=true -> boolean; //cannot declare this variable with this name
             (already exists)
5     statements
6   }
```

Listing 3.5: Conflicts with variable names

Keywords cannot be used as variable names.

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

See the example in Listing 3.6.

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

Listing 3.6: Conflicts with keyword names

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

```
1   My_variable_1
2   MyVariable1
```

Numbers are composed of digits (one or more). Nothing more is allowed.
See example below:

```
1   1562
2   1
```

A string is a sequence of n-characters enclosed by double quotes.
See example below:

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

3.2 LISS BLOCKS AND STATEMENTS

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

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

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

```
1  program sum{
2      declarations
3          int=2 -> integer;
4      statements
5          writeln(int+3);
6  }
```

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

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

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

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

```
1  liss : 'program' identifier body
2      ;
3
4  body : '{'
5      'declarations' declarations
6      'statements'   statements
7      '}'
8      ;
```

Listing 3.8: CFG for program in LISS

3.2.1 *LISS declarations*

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

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

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

```
1   declarations : variable_declaration* subprogram_definition*
2          ;
```

Listing 3.9: CFG for declarations in LISS

3.2.2 *LISS statements*

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

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

```
1   statements : statement*
2          ;
3   statement : assignment ';'
4          | write_statement ';'
5          | read_statement ';'
6          | function_call ';'
7          | conditional_statement
8          | iterative_statement
9          | succ_or_pred ';'
10         | copy_statement ';'
11         | cat_statement ';'
12         ;
```

Listing 3.10: CFG for statements in LISS

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

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

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

Listing 3.11: Example of using statements in LISS

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

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

Let's see in Listing 3.12 an example.

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

Listing 3.12: Example of assignment in LISS

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

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

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

Listing 3.13: CFG for assignment in LISS

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

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

```
1    program input1{
```

```
2      declarations
3        myInteger −> integer;
4      statements
5        input(myInteger);
6    }
```

Listing 3.14: Example of input operation in LISS

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

```
1  read_statement : 'input' '(' identifier ')'
2                 ;
```

Listing 3.15: CFG for input operation in LISS

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

Listing 3.16 shows some more examples.

```
1    writeln(4∗3);
2    writeln(2);
3    writeln();
```

Listing 3.16: Example of output operations in LISS

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

```
1    write_statement : write_expr '(' print_what ')'
2                   ;
3
4    write_expr : 'write'
5             | 'writeln'
6             ;
7
8    print_what :
9             | expression
10            ;
```

Listing 3.17: CFG for output operation in LISS

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

See Listing 3.18 for a complete example.

```
1  program SubPrg {
2
3    declarations
4
5      a = 4, b= 5, c= 5 -> integer;
6      d = [10,20,30,40], ev -> array size 4;
7
8
9      subprogram calculate() -> integer
10     {
11       declarations
12             fac = 6 -> integer;
13             res = -16 -> integer;
14
15        subprogram factorial(n -> integer; m -> array size 4) -> integer
16            {
17               declarations
18                     res = 1 -> integer;
19               statements
20                     while (n > 0)
21                     {
22                         res = res * n;
23                         n = n -1;
24                     }
25
26                     for (a in 0..3) stepUp 1
27                     {
28                       d[a] = a*res;
29                     }
30                     return res;
31            }
32       statements
33             res = factorial(fac,d);
34             return res/2;
35     }
36
37
38    statements
```

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

Listing 3.18: Example of call function in LISS

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

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

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

Listing 3.19: CFG for call function in LISS

3.2.3 *LISS control statements*

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

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

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

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

Listing 3.20: CFG for control statement in LISS

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

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

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

Let's see an example in Listing 3.21.

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

Listing 3.21: LISS syntax of a if statement

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

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

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

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

Listing 3.22: CFG for iterative statement in LISS

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

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

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

```
1  for(a in 1..10) stepUp 2 satisfying elems[a]==1{
2      ...
3  }
```

Listing 3.23: LISS syntax of a for-loop statement

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

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

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

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

```
1  for(b inArray elems){
2          ...
3  }
```

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

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

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

```
1  for_stat : 'for' '(' interval ')' step satisfy
2              '{' statements '}'
3         ;
4  interval : identifier type_interval
5          ;
6  type_interval : 'in' range
7                | 'inArray' identifier
8                ;
9  range : minimum '..' maximum
10       ;
```

```
11    minimum : number
12          | identifier
13          ;
14    maximum : number
15          | identifier
16          ;
17    step :
18        | up_down number
19        ;
20    up_down : 'stepUp'
21          | 'stepDown'
22          ;
23    satisfy :
24          | 'satisfying' expression
25          ;
```

Listing 3.25: CFG for for-statement in LISS

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

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

Let's see an example in Listing 3.26.

```
1    while (n > 0)
2    {
3      res = res * n;
4      pred n;
5    }
```

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

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

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

```
1    while_stat : 'while' '(' expression ')'
2               '{' statements '}'
3               ;
```

Listing 3.27: CFG for while-statement in LISS

3.2.4 *Others statements*

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

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

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

Listing 3.28 illustrates both statements.

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

Listing 3.28: Example of using succ/pred in LISS

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

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

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

Listing 3.29: CFG for succ and pred in LISS

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

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

Listing 3.30: Example of copy statement in LISS

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

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

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

```
1    copy_statement : 'copy' '(' identifier ',' identifier ')'
2                        ;
```

Listing 3.31: CFG for copy statement in LISS

Cat statement.

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

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

Listing 3.32: Example of cat statement in LISS

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

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

```
1    cat_statement : 'cat' '(' identifier ',' identifier ')'
2                        ;
```

Listing 3.33: CFG for cat statement in LISS

3.3 LISS SUBPROGRAMS

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

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

```
1    subprogram_definition: 'subprogram' identifier '(' formal_args ')'
         return_type f_body
2                            ;
3    f_body : '{'
4            'declarations' declarations
5            'statements' statements
6            returnSubPrg
7            '}'
8        ;
9    formal_args :
10            | f_args
```

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

Listing 3.34: CFG for block structure in LISS

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

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

3.4   EVOLUTION OF LISS SYNTAX

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

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

```
1 declaration : variable_declaration * subprogram_definition *
2             ;
```

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

```
1 statement : assignment   ';'
2            | write_statement   ';'
3            | read_statement   ';'
4            | conditional_statement
5            | iterative_statement
6            | function_call ';'
7            | succ_or_pred   ';'
8            | copy_statement   ';'
9            | cat_statement   ';'
10            ;
```

Listing 3.35: Function statement

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

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

```
1 elem_array : single_expression (',' s2=single_expression )*
2             ;
```

Listing 3.36: Rule element of array

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

```
1 expression : single_expression (rel_op single_expression )?
2             ;
```

Listing 3.37: Rule for Boolean expression

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

```
1 factor: '(' expression ')'
2       ;
```

Listing 3.38: Rule factor

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

```
1 cons // integer x sequence -> sequence
2      : 'cons' '(' expression ',' expression ')'
3      ;
4
5 delete // del : integer x sequence -> sequence
6        : 'del' '(' expression ',' expression ')'
7        ;
```

Listing 3.39: Rule cons and delete

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

```
1 type_interval : 'in' range
2                 | 'inArray' identifier
3                 //| 'inFunction' identifier
4                 ;
```

Listing 3.40: Rule type interval

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

```
1 fragment
2 COMMENT
3     : '/*'.*?'*/' /* multiple lines comment*/
4     | '//'~('\r' | '\n')* /* single line comment*/
5     ;
```

Listing 3.41: Lexical rule for Comment

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.

4.1 MIPS COPROCESSORS

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

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

1. Coprocessor 0, denoted by *CP0*, is incorporated in the CPU chip; it supports the virtual memory system and exception handling (also known as the *System Control Coprocessor*).

2. Coprocessor 1, denoted by *CP1*, is reserved for floating point coprocessor.

3. Coprocessor 2, denoted by *CP2*, is reserved for specific implementations.

4. Coprocessor 3, denoted by *CP3*, is reserved for the implementations of the architecture.

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

4.2 MIPS CPU DATA FORMATS

The CPU of MIPS defines four differents formats:

- *Bit* (1 bit, b)

- *Byte* (8 bits, B)

- *Halfword* (16 bits, H)

- *Word* (32 bits, W)

4.3 MIPS REGISTERS USAGE

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

Table 3.: MIPS registers

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

Note: N/A (Not applicable)

Table 3 is composed of 4 columns:

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

2. *Number* column defines the number of each register. This number can also be used to refer to the register in an instruction.

3. *Use* column refers to the meaning/definition of each register.

4. *Callee must preserve?* column provides information about the volatility of the register (used when a function is called).

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

- *PC* - Program Counter register

- *HI* - Multiply and Divide register higher result

- *LO* - Multiply and Divide register lower result

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

- when there is a multiply (*mul* instruction) operation, the *HI* and *LO* registers store the result of integer multiply.

- when there is a multiply-add (*madd* instruction) or multiply-subtract (*msub* instruction) operation, the *HI* and *LO* register store the result of integer multiply-add or multiply-subtract.

- when there is a division (*div* instruction) operation, the *HI* register store the remainder of the division and the *LO* register store the quotient of the division operation.

- when there is a multiply-accumulate (instruction) operation, the *HI* and *LO* registers store the accumulated result of the operation.

See an overview of the MIPS registers in Figure 3.

General Purpose Registers

Special Purpose Registers

31	0
r0 (hardwired to zero)	
r1	
r2	
r3	
r4	
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
r13	
r14	
r15	
r16	
r17	
r18	
r19	
r20	
r21	
r22	
r23	
r24	
r25	
r26	
r27	
r28	
r29	
r30	
r31	

31	0
HI	
LO	

31	0
PC	

Figure 3.: MIPS register

4.4 MIPS INSTRUCTION FORMATS

Instructions, in MIPS, are divided into three types:

- R-Type

- I-Type

- J-Type

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

Next sections provide the necessary details.

4.4.1 *MIPS R-Type*

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

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

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

Listing 4.1: R-Type instruction format

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

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

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

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

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

Listing 4.2: Example of a R-Type instruction

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

The following equivalence explains that meaning.

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

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.

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

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

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

In Table 5, the instruction 'add $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).

4.4.2 *MIPS I-Type*

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

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

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

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

Table 6.: Distinct I-Type instruction formats

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

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

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

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

Let's see examples of Imm16 instruction:

```
addi $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 primarily used for instruction encodings that have been moved, such as *LL* menmonic and *SC* mnemonic instruction. See Table 11.

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

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

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

4.4.3 *MIPS J-Type*

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

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

Table 12.: J-Type instruction format

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

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

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

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

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

Listing 4.3: Examples of J-Type instruction

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

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

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

4.5   MIPS ASSEMBLY LANGUAGE

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

4.5.1   *MIPS data declarations*

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

Then comes the part when the variable names are declared.

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

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

Listing 4.4: Syntax format of data declarations in MIPS

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

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

- *.ascii* store a string in memory without a null terminator.

- *.asciiz* store a string in memory with the null terminator.

- *.byte* store 'n' bytes contiguously in memory.

- *.halfword* store 'n' 16-bit halfwords contiguously in memory.

- *.word* store 'n' 32-bit words contiguously in memory.

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

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

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

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

Listing 4.5: Examples for declaring variables in MIPS

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

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

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

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

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

4.5.2 *MIPS text declarations*

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

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

Let's see an example in Listing 4.6.

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

Listing 4.6: Example of Text declarations in MIPS

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

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

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

Notice that the code will execute sequentially.

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

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

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

```
1    . data
2    . text
3      main :
4        li  $t0 ,  5
5        li  $t1 ,  5
6        mul  $t0 ,  $t0 ,  $t1
7        jal  jump_condition  #needs  to  jump  to  the  field  jump_condition
8        li  $t0 ,  4
9        li  $v0 ,  10
10       syscall
11     jump_condition :   #syntax  for  jump  and  conditional  instruction  in  mips
12       li  $t1 ,  5
13       jr  $ra
```

Listing 4.7: Example of a loop declaration in MIPS

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

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

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

Listing 4.8: Example of a comment in MIPS

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

```
1   # Comment giving name of program and description of function
2   # Template.s
3   # Bare−bones outline of MIPS assembly language program
4
5      .data       # variable declarations follow this line
6                   # ...
7
8      .text       # instructions follow this line
9
10       main:     # indicates start of code (first instruction to execute)
11                   # ...
```

Listing 4.9: Template of a MIPS assembly language

4.6 MIPS INSTRUCTIONS

MIPS has 6 type of instructions :

- instructions for data transfer

- instructions for arithmetic operations

- instructions for logical operations

- instructions for bitwise shift

- instructions for conditional branch

- instructions for unconditional branch

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

Table 13.: Example of Data transfer instruction in MIPS

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

Table 14.: Example of Arithmetic instruction in MIPS

Name	Instruction Syntax	Meaning	Type	Opcode	Funct
Add	add $d, $s, $t	$d = $s + $t	R	0x0	0x20
Add immediate	addi $t, $s, C	$t = $s + C (signed)	I	0x8	N/A
Subtract	sub $d, $s, $t	$d = $s - $t	R	0x0	0x22
Move	move $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 4.4.

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

Table 19.: Example of Pseudo Instructions in MIPS

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

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

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

Table 20.: Example of SYSCALL instruction in MIPS

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

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

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

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

Listing 4.10: Example of printing integer in MIPS

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

4.7 MIPS MEMORY MANAGEMENT

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

1. stack

2. heap

4.7.1 *MIPS stack*

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

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

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

Let's see an example in Listing 4.11.

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

Listing 4.11: Example of push and pop instructions in MIPS

4.7.2 *MIPS heap*

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

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

Let's see an example in Listing 4.12.

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

Listing 4.12: Example of code for allocating in the heap

4.8 MIPS SIMULATOR

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

For this project, we considered two nice free simulators:

- MARS simulator [3]

- SPIM simulator [4]

---

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

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

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

4.8.1 *MARS at a glance*

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

Let's see MARS IDE in Figure 4.

Figure 4.: MARS GUI

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

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

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

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

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

Figure 5.: MARS GUI (Execution mode)

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

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

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

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

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

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

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

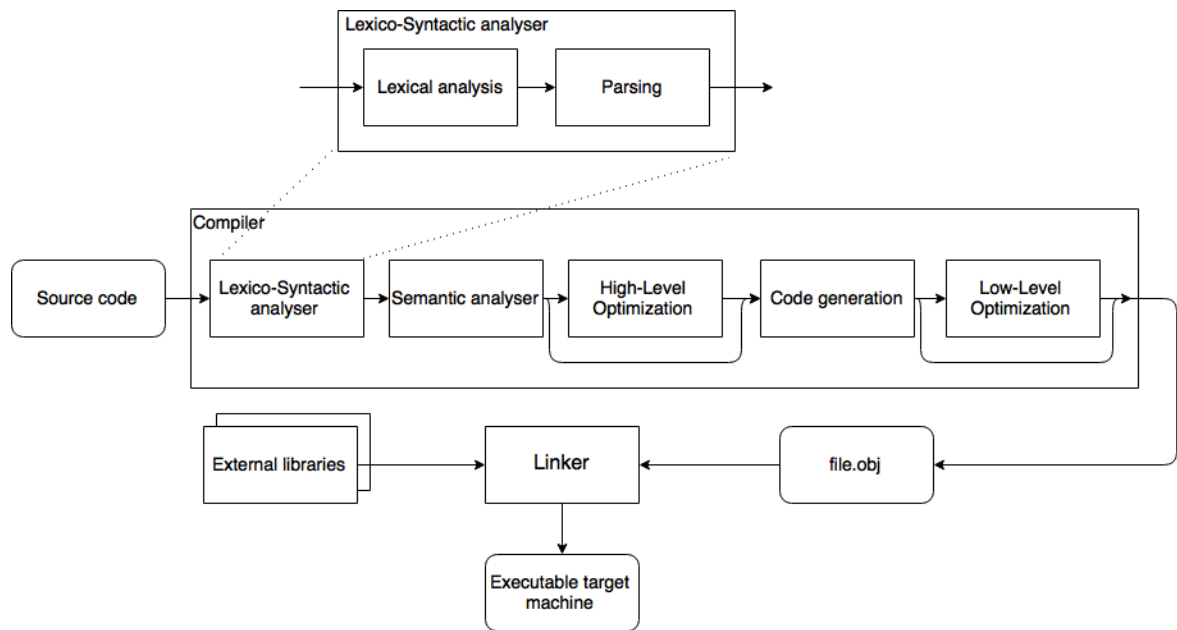3. Semantic analysis

4. Optimization

5. Code generation

Figure 6.: Traditional compiler

The task of constructing a compiler for a particular source language is complex. 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 (see Figure 7). 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 semantic analysis to catch some inconsistencies. Finally, after that, it may or may not have some optimization regarding the source code. Then the code generator translates the intermediate representation of the high-level programming into assembly code (lower level programming).
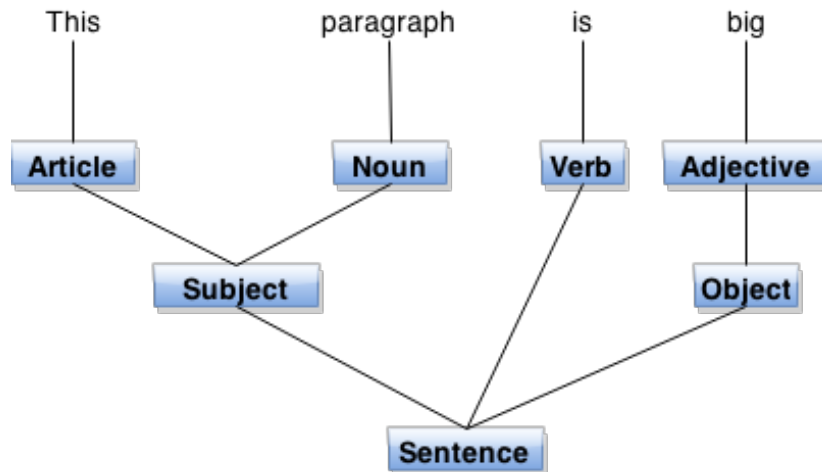
Figure 7.: Parsing

5.1 COMPILER GENERATION WITH ANTLR

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

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

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

```
1 facturas : fatura +
2          ;
3 fatura   : 'FATURA' cabec 'VENDAS' corpo
4          ;
5 cabec    : numFat idForn 'CLIENTE' idClie
6          { System.out.println("FATURA num: " + $numFat.text);}
7          ;
8 numFat   : ID
9          ;
10 idForn  : nome morada 'NIF:' nif 'NIB:' nib
```

Listing 5.1: AG representation on ANTLR

On the other hand, the lexer grammar defines the lexical rules which are regular expressions as can be seen in Listing 5.2. They define the set of possible character sequences that

are used to form individual tokens. A lexer recognizes strings and for each string found, it produces the respective tokens.

```
1  /*———————————— Lexer ——————————————————*/
2
3  ID   :    ('a'..'z'|'A'..'Z'|'_')  ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'-')*
4       ;
5
6  NUM  :    '0'..'9'+
```

Listing 5.2: Lexer representation

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 8).
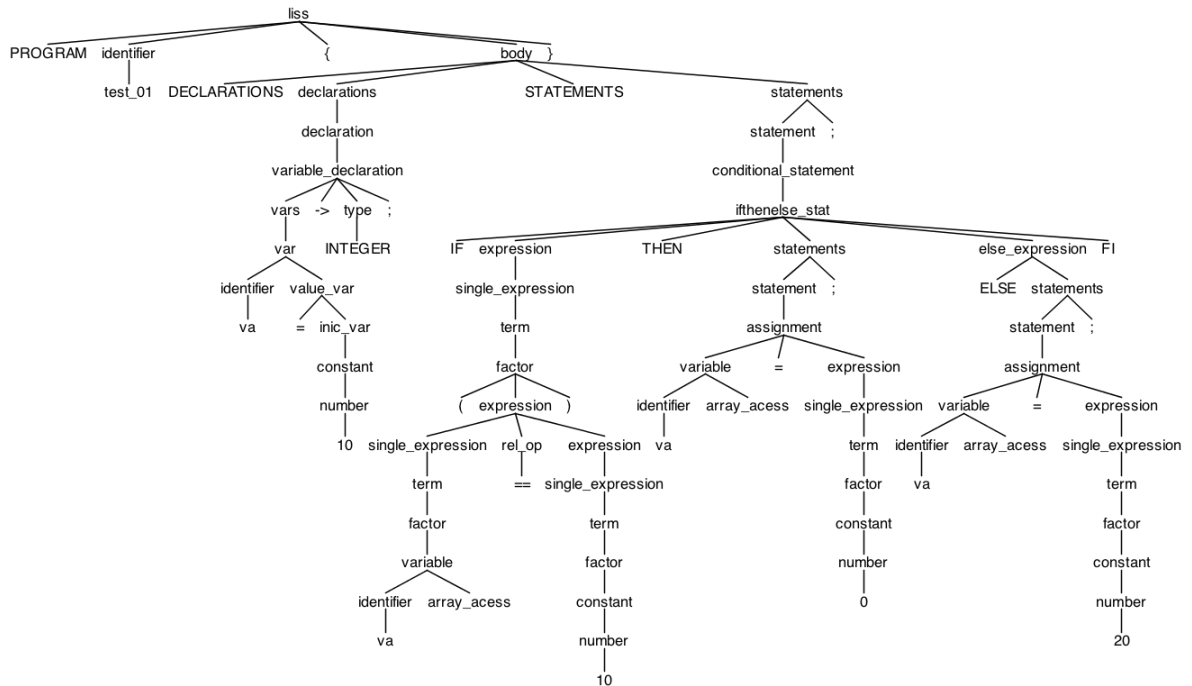

Figure 8.: AST representation

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

5.1.1 *Semantic Analyzer*

Symbol Table

Semantic system for LISS

SDE: DEVELOPMENT

Before we try to explain the concept of a Syntax-Directed Editor (SDE) (Reps and Teitelbaum, 1989b; Ko et al., 2005; MI-students et al., 2010; Teitelbaum and Reps, 1981; Reps et al., 1986; Reps and Teitelbaum, 1989a; Arefi et al., 1989), let's start defining what is an Integrated Development Environment (IDE).

An IDE is described as a software application that provides facilities to computer programmers for software development. It consists , normally, of a source code editor, a compiler, a debugger, and others tools. IDEs are designed for maximizing the productivity of programmers with visual interface and contains, normally, an interpreter, a compiler or both (see Figure 9).
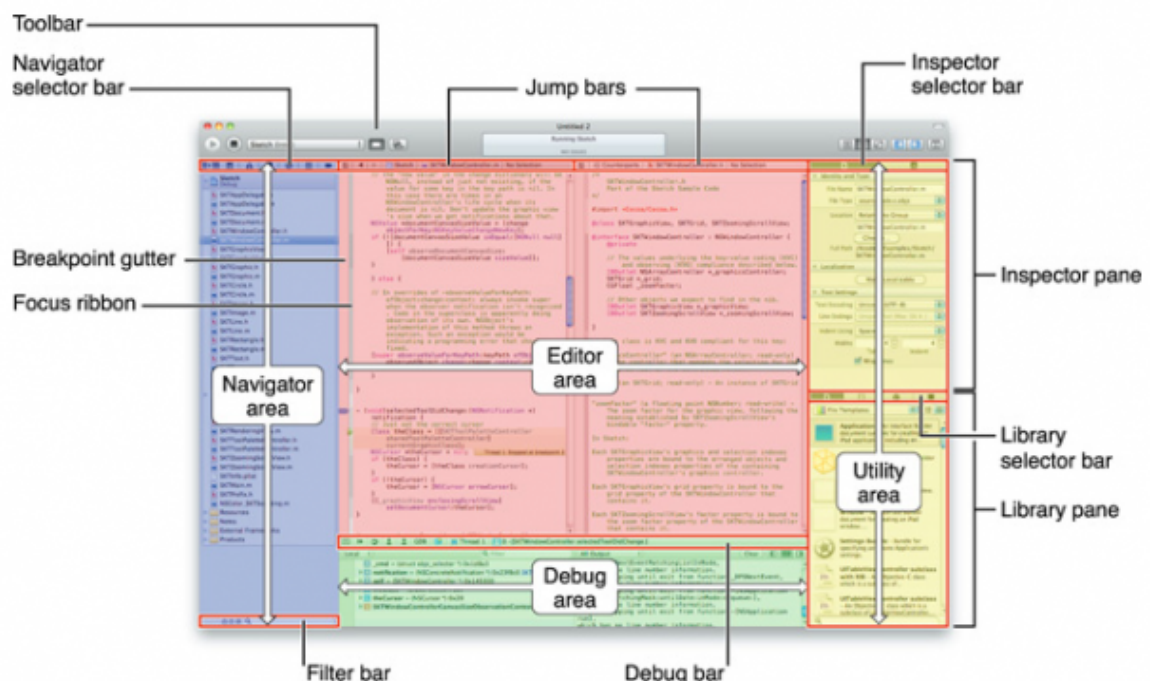

Figure 9.: Example of an IDE visual interface (XCode) [1]

Programs are created top down in the editor sections by inserting statements and expressions at the right cursor position of the current syntactic template and we can, by the cursor, change simply from one line of text to another one.

A SDE has the same approach of an IDE which is (as said above) an interactive programming environment with integrated facilities to create, edit, execute and debugging programs. The difference between them is that SDE encourages the program writing at a high level of abstraction, and promotes the programming based on a step by step refinement process.

It liberates the user from knowing the language syntactic details while editing programs.

SDE is basically guided by the syntactic structure of a programming language in both editing and execution. It is a hybrid system between a tree editor and a text editor.

The notion of cursor is really important in the context of SDE because, when the editing mode is on, the cursor is always located in a placeholder of a correct template (see next section) and the programmer may only change to another correct template at that placeholder or to its constituents.

It reinforces the idea that the program is a hierarchical composition of syntactic objects, rather than a sequence of characters.

6.1 WHAT IS A TEMPLATE?

The grammar of a programming language is a collection of production (or derivation rules) that state how a non-terminal symbol (LHS) is decomposed in a sequence of other symbols (RHS). A template is just the RHS of a grammar rule. Templates cannot be altered, they have placeholders for inserting a phrase or another template and they are generated by editor commands, according to the grammar production.

```
1 IF( condition )
2   THEN statement
3   ELSE statement
```

Listing 6.1: Example of a IF Conditional template

In Listing 6.1 we can see the editor template for the if-statement, where *condition* and *statement* are placeholders.

The notion of template is very important because templates are always syntactically correct for two reasons:

1. First, the command is validated to guarantee that it inserts a template permitted.

2. Second, the template is not typed, so it contains no lexical errors.

So a correct program (i.e., a valid sentence of the programming language) is created by choosing templates and replacing placeholders by others templates or by concrets values (numeric or string constants or identifiers).

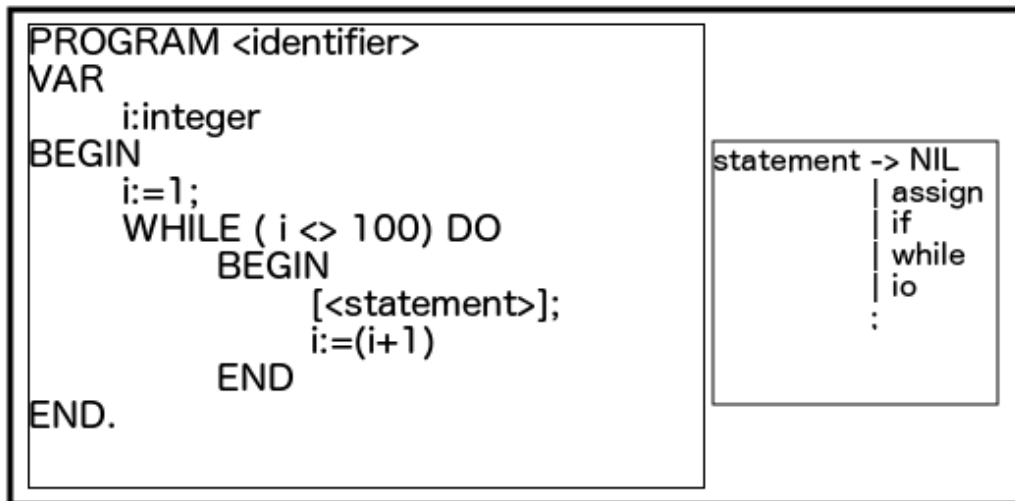To clarify the definition of SDE, we will explain it with the help of an example.

Figure 10.: SDE example

Figure 10 shows the main window of a standard Syntax-Directed Editor. In this figure, two boxes are displayed. The left one is the editor window where we code the program, and the right one exhibits templates choices.

Every <...> tag represents a placeholder, and [...] represents the actual cursor position.

As the cursor changes its position, moving from one placeholder to another placeholder, the right box will be updated according to the grammar rules in the context of the new cursor position. In this example, the cursor in Figure 10 is placed at the placeholder corresponding to a *statement*; at the same time, the right box will be updated with all the possibile templates according to the *statement* derivation rules (RHS).

To sum up, this is how a SDE works.

6.2 CONCEPTION OF THE SDE

7

CONCLUSION

7.1 FUTURE WORK

BIBLIOGRAPHY

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, 1986.

Henk Alblas. Introduction to attribute grammars. In H. Alblas and B. Melichar, editors, *Int. Summer School on Attribute Grammars, Applications and Systems*, pages 1–15. Springer-Verlag, Jun. 1991. LNCS 545.

F. Arefi, C.E. Hughes, and D.A. Workman. The object-oriented design of a visual syntax-directed editor generator. In *Computer Software and Applications Conference, 1989. COMP-SAC 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.

Bibliography

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.

Bibliography

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 (da Cruz and Henriques, 2007a) is an imperative programming language, defined by
the Language Processing members (Pedro Henriques and Leonor Barroca) at UM for teach-
ing 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 lan-
guages.

```
1  grammar  LissGIC ;
2
3  /* ****** Program ****** */
4
5  liss  :  'program'  identifier  body
6       ;
7
8
9  body  :  '{'
10          'declarations'  declarations
11          'statements'    statements
12          '}'
13       ;
14
15  /* ****** Declarations ****** */
16
17  declarations  :  variable_declaration∗ subprogram_definition∗
18               ;
19
20  /* ****** Variables ****** */
21
22  variable_declaration  :  vars '−>' type ';'
23                       ;
24
```

```
25 vars : var ( ',' var )*
26      ;
27
28 var : identifier value_var
29     ;
30
31 value_var :
32           | '=' inic_var
33           ;
34
35 type : 'integer'
36      | 'boolean'
37      | 'set'
38      | 'sequence'
39      | 'array' 'size' dimension
40      ;
41
42 typeReturnSubProgram : 'integer'
43                      | 'boolean'
44                      ;
45
46 dimension : number ( ',' number )*
47           ;
48
49 inic_var : constant
50          | array_definition
51          | set_definition
52          | sequence_definition
53          ;
54
55 constant : sign number
56          | 'true'
57          | 'false'
58          ;
59
60 sign :
61      | '+'
62      | '-'
63      ;
64
65 /* ****** Array definition ****** */
66
67 array_definition : '[' array_initialization ']'
```

```
68                     ;
69
70  array_initialization : elem (',' elem)*
71                        ;
72
73  elem : number
74       | array_definition
75       ;
76
77  /* ****** Sequence definition ****** */
78
79  sequence_definition : '<<' sequence_initialization '>>'
80                        ;
81
82  sequence_initialization :
83                          | values
84                          ;
85
86  values : number (',' number )*
87         ;
88
89  /* ****** Set definition ****** */
90
91  set_definition : '{' set_initialization '}'
92                  ;
93
94  set_initialization :
95                     | identifier '|' expression
96                     ;
97
98  /* ****** SubProgram definition ****** */
99
100 subprogram_definition: 'subprogram' identifier '(' formal_args ')'
     return_type f_body
101                        ;
102
103 f_body : '{'
104         'declarations' declarations
105         'statements' statements
106         returnSubPrg
107         '}'
108       ;
109
```

```
110 /* ****** Formal args ****** */
111
112 formal_args :
113             | f_args
114             ;
115
116 f_args   : formal_arg (',' formal_arg )*
117         ;
118
119 formal_arg : identifier '->' type
120             ;
121
122 /* ****** Return type ****** */
123
124 return_type :
125             | '->' typeReturnSubProgram
126             ;
127
128 /* ****** Return ****** */
129
130 returnSubPrg :
131             | 'return' expression ';'
132             ;
133
134 /* ****** Statements ****** */
135
136 statements : statement*
137           ;
138
139 statement : assignment ';'
140           | write_statement ';'
141           | read_statement ';'
142           | conditional_statement
143           | iterative_statement
144           | function_call ';'
145           | succ_or_pred ';'
146           | copy_statement ';'
147           | cat_statement ';'
148           ;
149
150 /* ****** Assignment ****** */
151
152 assignment : designator '=' expression
```

```
153                ;
154
155 /* ****** Designator ****** */
156
157 designator : identifier array_access
158                ;
159
160 array_access :
161                | '[' elem_array ']'
162                ;
163
164 elem_array : single_expression ( ',' single_expression )*
165           ;
166
167 /* ****** Function call ****** */
168
169 function_call : identifier '(' sub_prg_args ')'
170                ;
171
172 sub_prg_args :
173                | args
174                ;
175
176 args : expression ( ',' expression )*
177      ;
178
179 /* ****** Expression ****** */
180
181 expression : single_expression ( rel_op single_expression )?
182           ;
183
184 /* ****** Single expression ****** */
185
186 single_expression : term ( add_op term )*
187                     ;
188
189 /* ****** Term ****** */
190 term : factor ( mul_op factor )*
191      ;
192
193 /* ****** Factor ****** */
194
195 factor : inic_var
```

```
196          | designator
197          | '(' expression ')'
198          | '!' factor
199          | function_call
200          | specialFunctions
201        ;
202
203  specialFunctions : tail
204                   | head
205                   | cons
206                   | member
207                   | is_empty
208                   | length
209                   | delete
210                   ;
211
212  /* ****** add_op, mul_op, rel_op ****** */
213
214  add_op : '+'
215         | '-'
216         | '||'
217         | '++'
218         ;
219
220  mul_op : '*'
221         | '/'
222         | '&&'
223         | '**'
224         ;
225
226  rel_op : '=='
227         | '!='
228         | '<'
229         | '>'
230         | '<='
231         | '>='
232         | 'in'
233         ;
234
235  /* ****** Write statement ****** */
236
237  write_statement : write_expr '(' print_what ')'
238                  ;
```

```
239
240 write_expr : 'write'
241          | 'writeln'
242          ;
243
244 print_what :
245          | expression
246          ;
247
248 /* ****** Read statement ****** */
249
250 read_statement : 'input' '(' identifier ')'
251          ;
252
253 /* ****** Conditional & Iterative ****** */
254
255 conditional_statement : if_then_else_stat
256                       ;
257
258 iterative_statement : for_stat
259                     | while_stat
260                     ;
261
262 /* ****** if_then_else_stat ****** */
263
264 if_then_else_stat : 'if' '(' expression ')'
265                   'then' '{' statements '}'
266                   else_expression
267                 ;
268
269 else_expression :
270              | 'else' '{' statements '}'
271              ;
272
273 /* ****** for_stat ****** */
274
275 for_stat : 'for' '(' interval ')' step satisfy
276          '{' statements '}'
277      ;
278
279 interval : identifier type_interval
280      ;
281
```

```
282 type_interval : 'in' range
283              | 'inArray' identifier
284              ;
285
286 range : minimum '..' maximum
287       ;
288
289 minimum : number
290         | identifier
291         ;
292
293 maximum : number
294         | identifier
295         ;
296
297 step :
298      | up_down number
299      ;
300
301 up_down : 'stepUp'
302         | 'stepDown'
303         ;
304
305 satisfy :
306         | 'satisfying' expression
307         ;
308
309 /* ****** While_Stat ****** */
310 while_stat : 'while' '(' expression ')'
311              '{' statements '}'
312            ;
313
314 /* ****** Succ_Or_Predd ****** */
315
316 succ_or_pred : succ_pred identifier
317              ;
318
319 succ_pred : 'succ'
320           | 'pred'
321           ;
322
323 /* ****** SequenceOper ****** */
324
```

```
325  tail // tail : sequence -> sequence
326      : 'tail' '(' expression ')'
327      ;
328
329  head // head : sequence -> integer
330      : 'head' '(' expression ')'
331      ;
332
333  cons // integer x sequence -> sequence
334      : 'cons' '(' expression ',' expression ')'
335      ;
336
337  delete // del : integer x sequence -> sequence
338       : 'del' '(' expression ',' expression ')'
339       ;
340
341  copy_statement // copy_statement : seq x seq -> void
342               : 'copy' '(' identifier ',' identifier ')'
343               ;
344
345  cat_statement //cat_statement : seq x seq -> void
346               : 'cat' '(' identifier ',' identifier ')'
347               ;
348
349  is_empty // is_empty : sequence -> boolean
350          : 'isEmpty' '(' expression ')'
351          ;
352
353  length // length : sequence -> integer
354        : 'length' '(' expression ')'
355        ;
356
357  /* ****** set_oper ****** */
358
359  member // isMember : integer x sequence -> boolean
360        : 'isMember' '(' expression ',' identifier ')'
361        ;
362
363
364
365  /*
      +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
       */
```

```
366
367 string : STR
368        ;
369
370 number : NBR
371        ;
372
373 identifier : ID
374            ;
375 /*
      ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      */
376
377
378 /* ****** Lexer ****** */
379
380 NBR : ('0'..'9')+
381     ;
382
383 ID : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')* //removi o uso
     do signal '-' conflitos com os valores do signal
384     ;
385
386 WS :    ( [ \t\r\n] | COMMENT) -> skip
387     ;
388
389 STR :   '"' ( ESC_SEQ | ~('"') )* '"'
390     ;
391
392
393 fragment
394 COMMENT
395     : '/*'.*?'*/' /* multiple comments */
396     | '//'~('\r' | '\n')* /* single comment */
397     ;
398
399 fragment
400 ESC_SEQ
401     :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
402     ;
```

lissGIC.g4

Auxiliary results which are not main-stream; or

71

Details of results whose length would compromise readability of main text; or
Specifications and Code Listings: should this be the case; or
Tooling: Should this be the case.