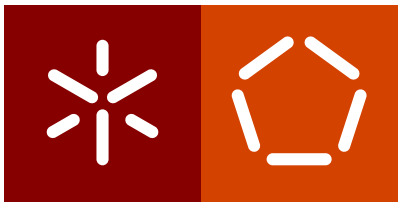
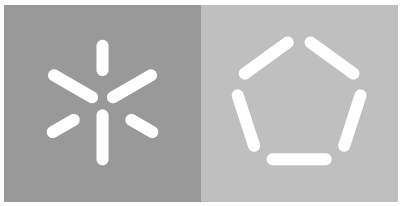


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Damien da Silva Vaz

Implementing a Syntax Directed Editor for LISS.

June 2016



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

June 2016

ACKNOWLEDGEMENTS

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

ABSTRACT

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

RESUMO

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

CONTENTS

LIST OF FIGURES

1 http://www.biiet.org/blog/wp-content/uploads/2013/07/img028.jpg
2 http://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/MIPS_Architecture(Pipelined).svg/300px-MIPS_Architecture_(Pipelined).svg.png
3 **?**
4 http://www.alauda.ro/wp-content/uploads/2011/04/XCode-interface-e1302035068112.png

LIST OF TABLES

List of Tables

1

INTRODUCTION

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

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

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

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

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

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

1.1 OBJECTIVES

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

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

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

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

1.2 RESEARCH HYPOTHESIS

1.3 THESIS OUTCOMES

1.4 DOCUMENT STRUCTURE

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

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

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

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

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

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

2

LANGUAGES AND GRAMMAR: CONCEPT & TOOLS

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

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

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

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

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

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

where each side is composed by a sequence of symbols.

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

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

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

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

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

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

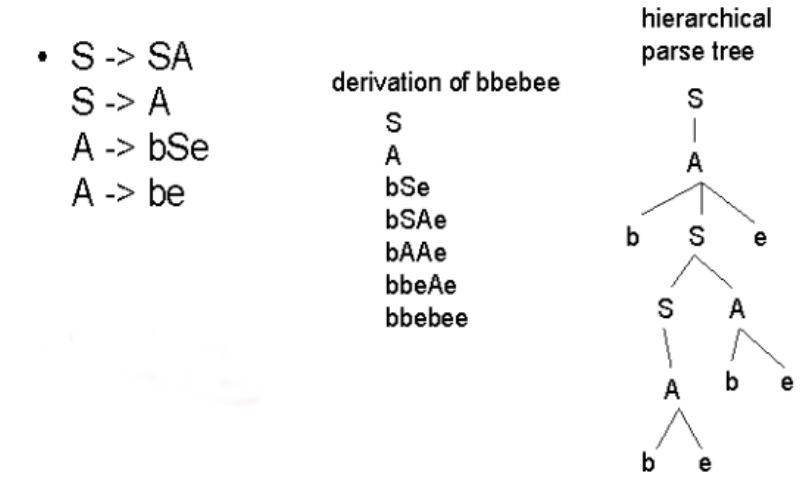


Figure 1.: CFG example [1]

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

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

```
facturas : fatura (facturas)*
         ;

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

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

idFat : numFat ;

numFat : ID ;

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

idClie : nome morada 'NIF:' nif
       ;

nome :  STR ;
```

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

Listing 2.1: Example of an AG

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

2.1 FORMAL GRAMMAR

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

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

- A start symbol S $\in$ P

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

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

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

Listing 2.2: A rule production

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

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

3

LISS LANGUAGE

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

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

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

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

3.1 LISS DATA TYPES

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

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

Let's obverse a LISS code example:

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

Listing 3.1: Declaring a variable in LISS

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

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

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

Listing 3.2: CFG for declaring a variable in LISS

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

Table 1.: LISS data types

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

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

```
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:

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

Thi is equivalent to the initialization below:

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

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

The grammar for array declaration and initialization is shown below.

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

array_initialization : elem (',' elem)*
                     ;

elem : number
```

```
8       | array_definition
9       ;
```

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

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

To define a set by comprehension, the free variable and the expression shall be return between curly brackets. The 'identifier' 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 (operation interval) and others numbers are not included in the set.

The syntax for set declaration and initialization is :

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

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

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

Listing 3.4: Example of valid operations using sequence on LISS

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

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

Those operations will be explained further and deeper.

The grammar below defines how to declare a sequence:

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

sequence_initialization :
                        | values
                        ;

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

3.2 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 3.5).

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

Listing 3.5: 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 implementing the variable always in a certain part of the code, it become confusing. This makes the life harder to the programmer in understanding the code when the code is quite long/big.

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 3.6).

```
liss : 'program' identifier body
   ;

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

Listing 3.6: BNF of 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 3.7).

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

Listing 3.7: 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 3.8.

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

Listing 3.8: BNF of statements in LISS

Let's see one example of a LISS program which shows how the language shall be used (see Listing 3.9).

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

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

Listing 3.9: 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 3.10 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 3.10: Example of assignment in LISS

In Listing 3.10, 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 3.11.

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

Listing 3.11: BNF of assignment in LISS

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

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

```
1  program input1{
```

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

Listing 3.12: Example of input operation in LISS

Notice that, in Listing 3.12, 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 3.13.

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

Listing 3.13: BNF of 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 3.14 shows some more examples.

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

Listing 3.14: 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 3.15).

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

write_expr : 'write'
           | 'writeln'
           ;

print_what :
           | expression
           ;
```

Listing 3.15: BNF of output operations 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 3.16 for a complete example.

```
program SubPrg {

 declarations

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


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

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

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


 statements
```

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

Listing 3.16: Example of call function in LISS

In Listing 3.16, we can see that the function *calculate()* in the main program, that is called and that is created under the declarations section.

The grammar who rules the function call is shown in Listing 3.17.

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

Listing 3.17: BNF of 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 behavior.

The first one is called conditional statement and it has only one variant in LISS language (see Listing 3.18).

The second one is called cyclic statement or iterative statement, and it has two variants (see Listing 3.18).

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

Listing 3.18: BNF of control statements in LISS

These control statements, mimics the syntax and the behavior 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 3.19.

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

Listing 3.19: LISS syntax of a if statement

The code shown in Listing 3.19, 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 3.20.

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

Listing 3.20: BNF of iterative statement in LISS

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

The 'for' statement has two different ways for defining his comportment. 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 a explicit value.

In LISS, the control variable is set in a given integer interval defined by the lower and upper bounds. Therefore, by default, the step is incremented by one but it is possible to increment or decrease it by setting it explicitly. Additionally, we may write a condition for filtering on each step the for-loop statement, like setting an if-condition. This can be done with the following syntax:

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

Listing 3.21: LISS syntax of a for-loop statement

In Listing 3.21, the control variable 'a' is set to a range by 1 to 10 and would be increased (due to 'stepUp' syntax) by 2. Also there is a filter condition which is applied after the 'satisfying' syntax and it will be tested on each step of the for-loop statement. Notice that the filter condition must boolean and that if it pass the condition, then the inner content of the for-loop statement would be executed.

Otherwise the control variable would be incremented with the explicit value 2 and the filter condition tested again. This is the first example, about expressing in a way the for-loop statement. Let's see the second way on the next lines.

There is also the possibility for ease use, in expressing a for-each statement on a array. And this is done by calling a simple syntax in LISS.

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

Listing 3.22: LISS syntax of a for-each statement on array

In Listing 3.22, 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 behavior and also a filter condition.

The next grammar fragment describes the cycle for in LISS:

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

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

Listing 3.23: BNF of for statement in LISS

Finally, the while-statement consists in a block of code which execute some actions repeatly until the condition associated would be false. It is considered to be the best of those two worlds: if-statement and for-loop statement.

Each time that the content of a while-statement is performed, the condition associated with it will be evaluated again and see if it can performs the action associated or quit the while-statement. Notice that the condition must be a boolean and it can be a niche of an indefinitely loop. Let's see an example in Listing 3.24.

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

Listing 3.24: LISS syntax of a while-statement in LISS

In Listing 3.24, the while-statement has the following condition 'n>0' and evaluates it firstly. 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 ruled behind the while-statement is shown below:

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

Listing 3.25: BNF of while-statement in LISS

3.2.4 *Others statements*

Succ/Pred. Those operations are available for incrementing or decreasing a variable. This is a common situation in modern programming language which LISS had to have it and, in this case, making life easier for the developers.

The syntax 'succ' means increment and the syntax 'pred' means decrease. Beside that only integers variable can be used with those operations and that the increment/decrease is operated by incrementing/decreasing by one the variable.

Let's see an example of this operation.

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

Listing 3.26: Example of using succ/pred in LISS

As we can see in Listing 3.26, the example increments the variable 'int1' firstly following by a 'decrease' on the next line of the same variable.

Grammar of succ and pred in LISS shown at 3.27.

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

Listing 3.27: BNF of succ and pred in LISS

Copy statement. This operation works with only and only the type: sequence. Basically, it copies the sequence to another sequence. Let's see an example at 3.28.

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

Listing 3.28: Example of copy statement in LISS

In Listing 3.28, the function copy is copying the content of the variable *seq1* to *seq2*. The output of this function is void (as shown to table 2) due to the fact that it works only internally with those variables and do not need to return anything.

The grammar of copy statement is available at 3.29.

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

Listing 3.29: BNF of copy statement in LISS

Cat statement.

This operation works with only and only the type: sequence. The behaviour of this function is to concatenate the sequence to another sequence. Let's see an example of this operation (see Listing **??**).

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

So, in Listing **??**, the function cat is concatenating the content of *seq1* to *seq2*. The output of this function is void (as shown to table 2) due to the fact that it works only internally with those variables and do not need to return anything.

The grammar of cat statement is available at 3.30.

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

Listing 3.30: BNF of cat statement in LISS

3.3 LISS SUBPROGRAMS

In LISS, it is possible to organize your code by splitting a portion of your code into sub-programs. This allows to the programmer to reuse or give more clarity to his code by creating functions or doing some procedure. Also, it is possible to create sub-programs into sub-programs by using a nesting strategy.

The syntax who defines a sub-program in LISS is shown in Listing 3.31.

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

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

Listing 3.31: BNF of block structure in LISS

Note that every variable, who are declared inside of a sub-program, are local, and they can be access to others nested sub-program of a sub-program. However, variables declared in the program (not in a sub-program) are considered global and not local.

3.4 EVOLUTION OF LISS SYNTAX

Due to the maturity of the language already done by the professors whom invented, we have added some few but extra changes for a better experience of the programming language.

One of the first objectives is that we have added some better clarification syntactically of the programming language which was by not mixing functions and variable declarations between them (See Figure 3.4). Like that we, indirectly, teach the programmer by doing it in the right way. So we declare, firstly, the variables and then the functions.

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

Another objectives was to add punctuation after each line of any statements (See Figure 3.32).

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

Listing 3.32: Function statement

As adding also a 'cat_statement' rule which works with only sets. It concatenate sets with another sets.

Regarding to the arrays, it was previously able to add some expressions over the access elements of the array. But these access were able to use some boolean expression which was non sense regarding to the semantic. So we changed and allowed any other expressions less than boolean expression (See figure 3.33).

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

Listing 3.33: Rule element of array

In the previous version of the LISS, it was allowed to create a big boolean expression, but we decided to change that and only able to create one boolean expression (See figure 3.34). It was a non sense of having an expression like that : '3 == 4 == 5 != 6'.

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

Listing 3.34: Rule expression

We added the possibility of using some parenthesis over expressions (See figure 3.35).

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

Listing 3.35: Rule factor

We changed the rules of two functions: 'cons' and 'del'. These functions were working with both in the same way. Waiting for an expression and a variable as arguments. Now, we decided to change that and giving more powers of expressivity to those functions 3.36.

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

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

Listing 3.36: Rule cons and delete

Beside of adding some improvements to the grammar, we additionally deleted a rule which we thought it wasn't necessary to have (see figure 3.37).

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

Listing 3.37: Rule type interval

Last but not least, we also added the way of adding some comments to the programming language. Giving more power for the user in giving some informations about his code.

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

Listing 3.38: Rule comment

4

TARGET MACHINE

4.1 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 instructions pipelines. And this was due to a main problem, back to the 80's, that some instruction as division would take a longer time to complete due to the fact that CPU needed to wait that the division ended before passing to the next instruction into the pipeline.

MIPS architecture began with a 32-bit version until now, which has his own 64-bit versions. It was primarily used in embedded systems and video games consoles until late 2006, when it came to computers.

The architecture of MIPS is composed by 5 stages (see Figure 2).

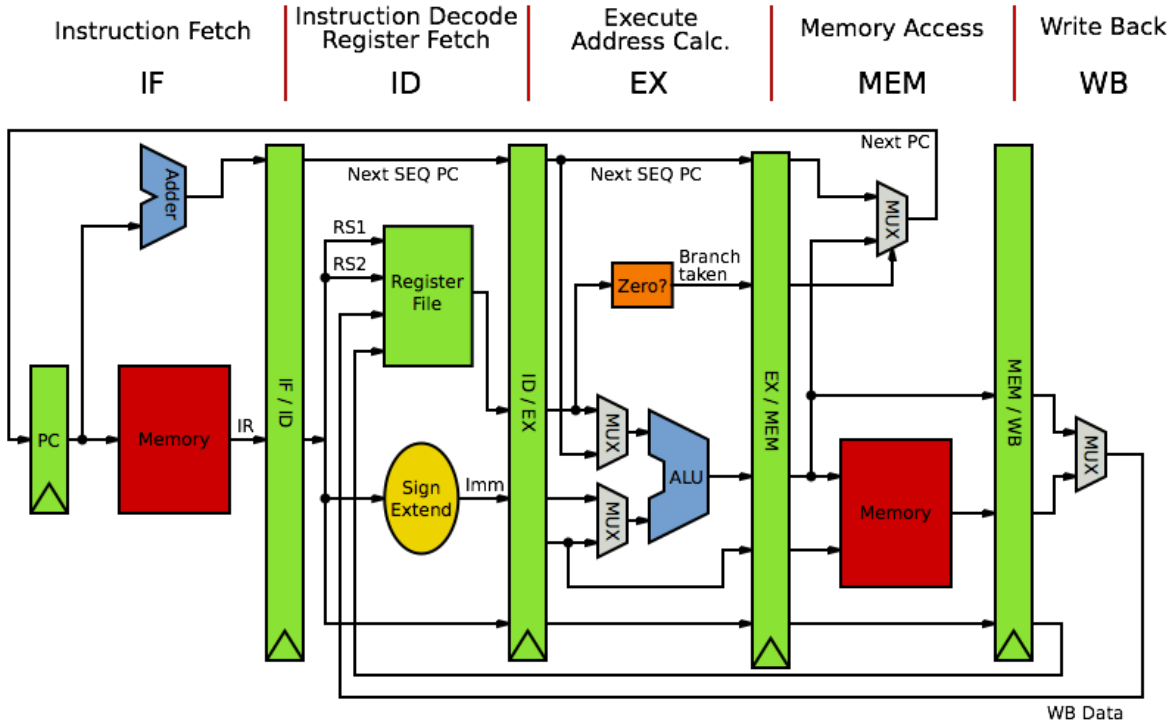


Figure 2.: MIPS architecture [1]

It has 32 registers and 5 type of instructions :

- instructions for data transfer
- instructions for arithmetic
- instructions for logical
- instructions for conditional branch
- instructions for unconditional jump

It uses a mnemonic to represent each low-level machine instruction or operation. Operations require three operands in order to form a complete instruction in MIPS assembly code, the address for the result and the address of the two operands.

A simple example of a complete instruction can be seen in Listing 4.1.

```
1  add   $s1 ,$s2 ,$s3
```

Listing 4.1: Sum of two registers in MIPS assembly code

The instruction shown in Listing 4.1, means that register $s2 shall be added to register $s3 and their sum (the result) stored in $s1 register (note that each operand is represented by $ sign).

As mentioned in the previous section, the AG processed by the compiler generator ANTLR will compose MIPS assembly. The MIPS assembly instructions generated so far will be converted into executable machine code by an assembler included in MARS environment (a MIPS simulator and debugger) according to the process depicted in Figure 3.

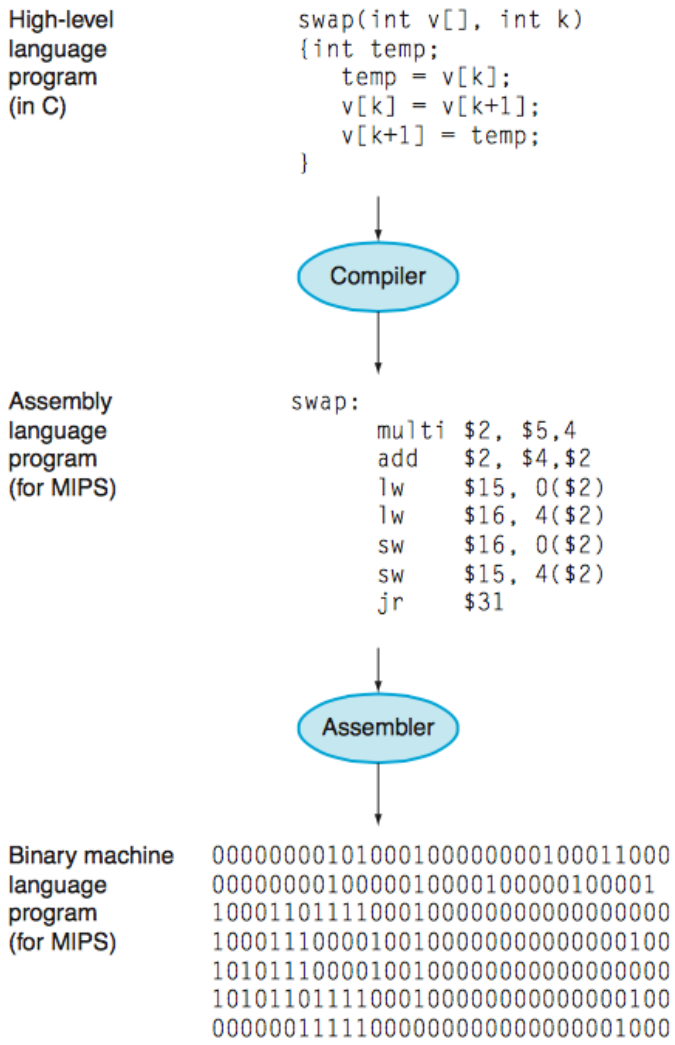


Figure 3.: Stages converting high-level language into binary machine language [2]

After presenting in the previous chapter the essential definitions for the intended work, in this chapter 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.

5

COMPILER DEVELOPMENT

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

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

The compiler task is divided into several steps (see Figure 4):

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

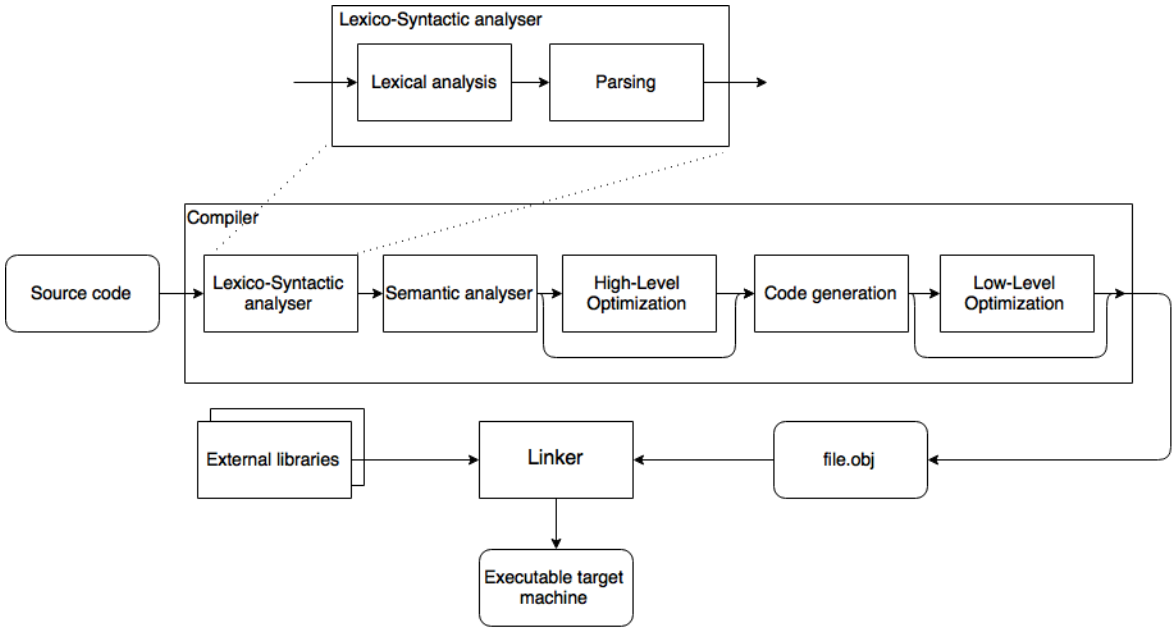


Figure 4.: Traditional compiler

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 5). 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).

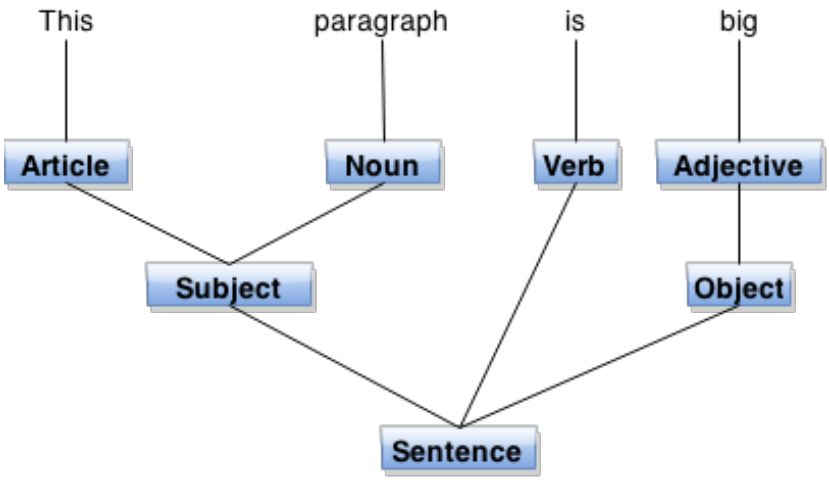


Figure 5.: Parsing

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.

```
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.

```
/*------------------ Lexer ------------------------------------------------*/

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

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 6).

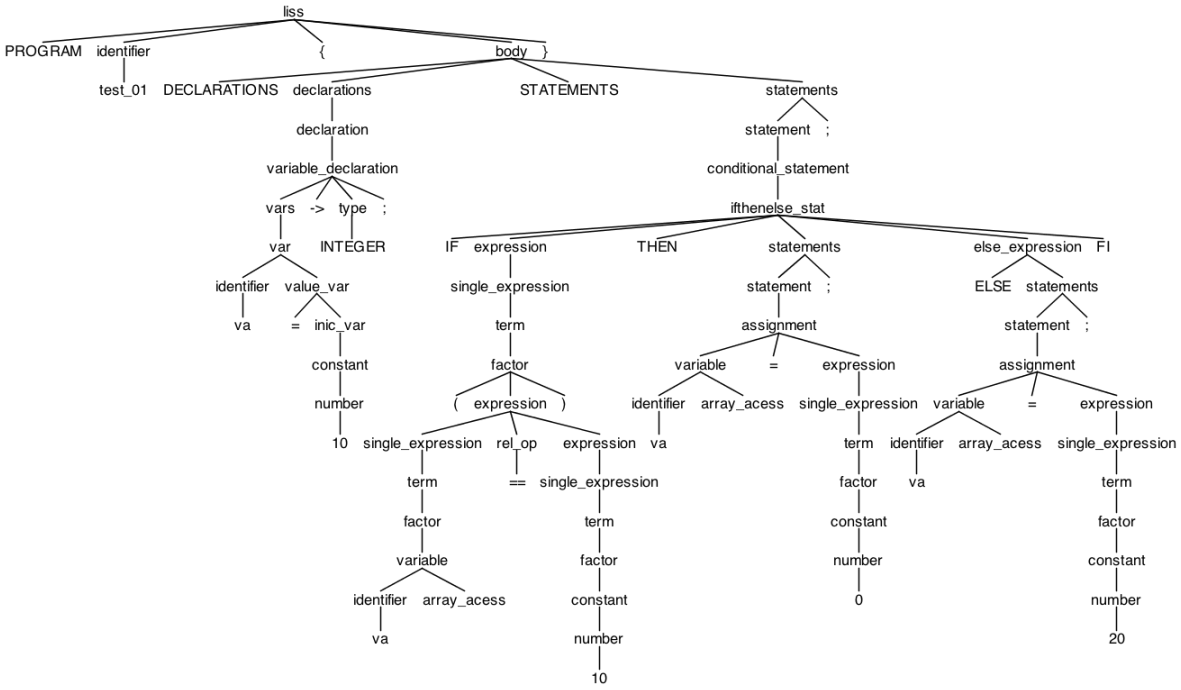


Figure 6.: AST representation

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

6

SDE: DEVELOPMENT

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

An IDE is described as a software application that provides facilities to computer programmers for software development. It consists , normally, of a source code editor, a compiler, a debugger, and others tools. IDEs are designed for maximizing the productivity of programmers with visual interface and contains, normally, an interpreter, a compiler or both (see Figure 7).

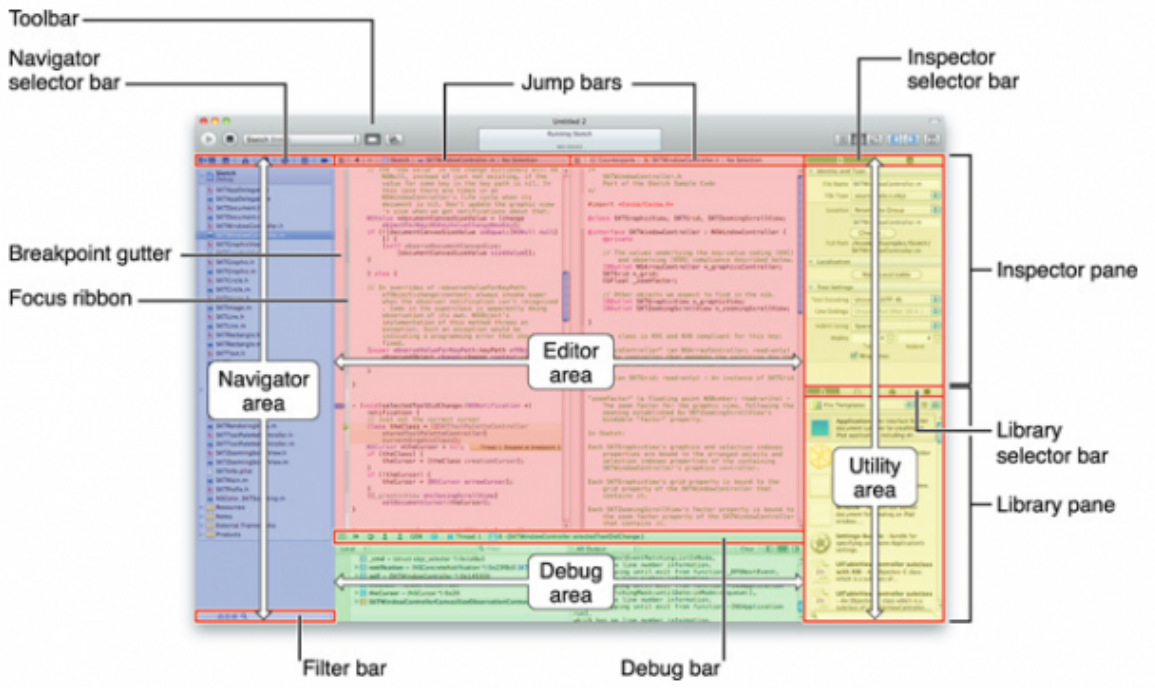


Figure 7.: Example of an IDE visual interface (XCode) [1]

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.

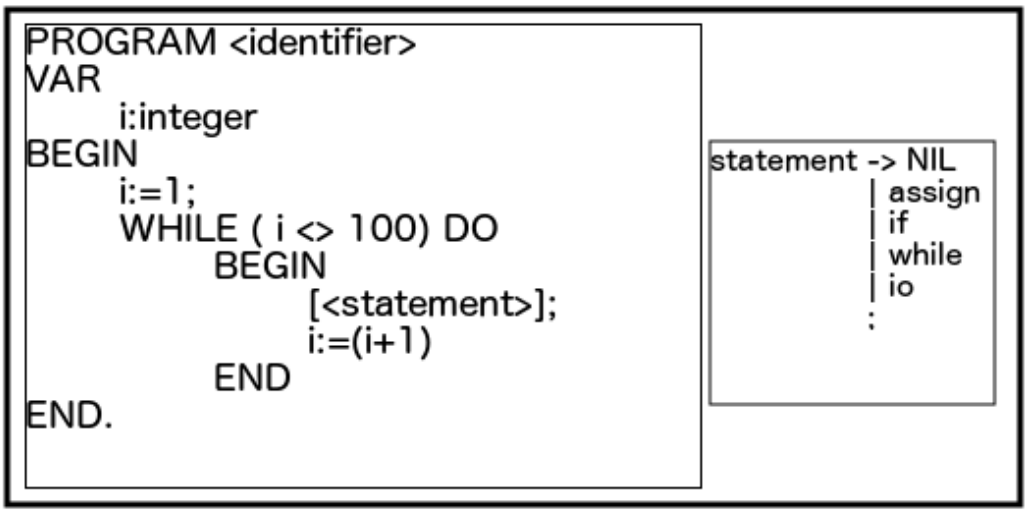


Figure 8.: SDE example

Figure 8 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 8 is placed at the placeholder corresponding to a *statement*; at the same time, the right box will be updated with all the possibile templates according to the *statement* derivation rules (RHS).

To sum up, this is how a SDE works.

6.2 CONCEPTION OF THE SDE

7

CONCLUSION

7.1 FUTURE WORK

BIBLIOGRAPHY

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

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

F. Arefi, C.E. Hughes, and D.A. Workman. The object-oriented design of a visual syntax-directed editor generator. In *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pages 389 –396, sep 1989. doi: 10.1109/CMPSAC.1989.65112.

Noami Chomsky. Context-free grammars and pushdown storage. RLE Quarterly Progress Report 65, MIT, Apr. 1962.

Daniela da Cruz and Pedro Rangel Henriques. Liss - language of integers, sequences and sets. Talk to the gEPL, Dep. Informática / Univ. Minho, Oct. 2005.

Daniela da Cruz and Pedro Rangel Henriques. Liss – language, compiler & companion. In *Proceedings of the Conference on Compiler Technologies for .Net (CTNET'06 - Universidade da Beira Interior, Portugal)*, Mar. 2006a. (to be published).

Daniela da Cruz and Pedro Rangel Henriques. Liss compiler homepage. http://www.di.uminho.pt/ gepl/LISS, 2006b.

Daniela da Cruz and Pedro Rangel Henriques. LISS — a linguagem e o compilador. Relatório interno do CCTC, Dep.Informática / Univ. do Minho, Jan. 2007a. (to be published).

Daniela da Cruz and Pedro Rangel Henriques. Liss — the language and the compiler. In *Proceedings of the 1.st Conference on Compiler Related Technologies and Applications, CoRTA'07 — Universidade da Beira Interior, Portugal*, Jul 2007b.

P. Deransart and M. Jourdan, editors. *Attribute Grammars and their Applications*, Sep. 1990. INRIA, Springer-Verlag. Lecture Notes in Computer Science, nu. 461.

P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars: Main results, existing systems and bibliography. In *LNCS 341*. Springer-Verlag, 1988.

G. Filè. Theory of attribute grammars. (Dissertation) Onderafdeling der Informatica, Technische Hogeschool Twente, 1983.

M. C. Gaudel. Compilers generation from formal definitions of programming languages: A survey. In *Methods and Tools for Compiler Construction*, pages 225–242. INRIA, Rocquencourt, Dec. 1983.

Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer, New York, Heilderberg, Dordrecht, London, 2nd edition, 2012. ISBN 978-1-4614-4698-9. doi: 10.1007/978-1-4614-4699-6.

Niklas Holsti. Incremental interaction by syntax transformation. In *Compiler Compilers and Incremental Compilation – Proc. of the Workshop, Bautzen*, pages 192–210. Akademie der Wissenschaften der DDR, Institut für Informatik und Rechentechnik, Oct. 1986.

John E. Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 5 – Context-Free Grammars and Languages. Addison-Wesley, 3rd ed. edition, 2006. ISBN 0-321-46225-4.

Uwe Kastens. Attribute grammar as a specification method. In H. Alblas and B. Melichar, editors, *Int. Summer School on Attribute Grammars, Applications and Systems*, pages 16–47. Springer-Verlag, Jun. 1991a. LNCS 545.

Uwe Kastens. Attribute grammars in a compiler construction environment. In H. Alblas and B. Melichar, editors, *Int. Summer School on Attribute Grammars, Applications and Systems*, pages 380–400. Springer-Verlag, Jun. 1991b. LNCS 545.

Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmers text editing. In *CHI '05: HUMAN FACTORS IN COMPUTING*, pages 1557–1560. Press, 2005.

MI-students, Daniela da Cruz, and Pedro Rangel Henriques. Agile - a structured-editor, analyzer, metric-evaluator and transformer for attribute grammars. In Luis S. Barbosa and Miguel P. Correia, editors, *INForum'10 — Simposio de Informatica (CoRTA'10 track)*, pages 197–200, Braga, Portugal, September 2010. Universidade do Minho.

Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4.

Nuno Oliveira, Maria Joao Varanda Pereira, Pedro Rangel Henriques, Daniela da Cruz, and Bastian Cramer. Visuallisa: A visual environment to develop attribute grammars. *ComSIS – Computer Science an Information Systems Journal, Special issue on Advances in Languages, Related Technologies and Applications*, 7(2):266 – 289, May 2010. ISSN ISSN: 1820-0214.

Terence Parr. An introduction to antlr. http://www.cs.usfca.edu/ parrt/-course/652/lectures/antlr.html, Jun. 2005.

Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007. URL http://www.amazon.de/Complete-ANTLR-Reference-Guide-Domain-specific/dp/0978739256.

K. J. Räihä. Bibliography on attribute grammars. *SIGPLAN Notices*, 15(3):35–44, 1980.

Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989a.

Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual*. Texts and Monographs in Computer Science. Springer-Verlag, 1989b.

Thomas Reps, Tim Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 5(3): 449–477, 1983.

Thomas Reps, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language-based editors. *Communications of the ACM*, Sep. 1986.

S.D. Swierstra and H.H. Vogt. Higher order attribute grammars, lecture notes of the Int. Summer School on Attribute Grammars, Applications and Systems. Technical Report RUU-CS-91-14, Dep. of Computer Science / Utrecht Univ., Jun. 1991.

Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9), Sep. 1981.

H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. On the efficient incremental evaluation of Higher Order Attribute Grammars. Research Report RUU-CS-90-36, Dep. of Computer Science / Utrecht Univ., Dec. 1990.

William Waite and Gerhard Goos. *Compiler Construction*. Texts and Monographs in Computer Science. Springer-Verlag, 1984.

A

LISS CONTEXT FREE GRAMMAR

LISS (da Cruz and Henriques, 2007a) is an imperative programming language, defined by the Language Processing members (Pedro Henriques and Leonor Barroca) at UM for teaching purposes. It allows handling integers, sets of integers, dynamic sequences, complex numbers, polynomials, etc., etc (da Cruz and Henriques, 2007b,a, 2006a,b, 2005).

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

```
grammar LissGIC;

/* ****** Program ****** */

liss : 'program' identifier body
     ;


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

/* ****** Declarations ****** */

declarations : variable_declaration* subprogram_definition*
             ;

/* ****** Variables ****** */

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

```

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

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

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

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

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

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

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

range : minimum '..' maximum
      ;

minimum : number
        | identifier
        ;

maximum : number
        | identifier
        ;

step :
     | up_down number
     ;

up_down : 'stepUp'
        | 'stepDown'
        ;

satisfy :
        | 'satisfying' expression
        ;

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

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

succ_or_pred : succ_pred identifier
             ;

succ_pred : 'succ'
          | 'pred'
          ;

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

```

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

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

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

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

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

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

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

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

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

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



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

```
string : STR
       ;

number : NBR
       ;

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


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

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

ID : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')* //removi o uso do signal '-' conflitos com os valores do signal
   ;

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

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


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

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

lissGIC.g4

Auxiliary results which are not main-stream; or

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

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.