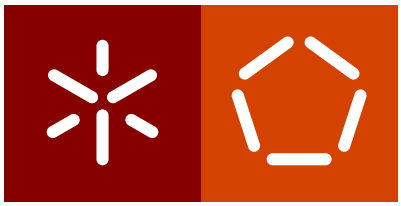
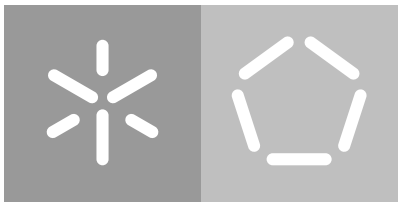


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Jorge Miguel Sol Ferreira

Syntax-Directed Editor Generator

November 2017



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Jorge Miguel Sol Ferreira

Syntax-Directed Editor Generator

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor Pedro Rangel Henriques

November 2017

ACKNOWLEDGEMENTS

First of all I would like to thank Professor Pedro Rangel Henriques for his support during the duration of this dissertation.

I would also like to thank my family for being always supportive and being always there for me.

ABSTRACT

The goal of the master's thesis work here reported is to develop a system capable of generating a syntax-directed editor (SDE) for any given language definition. A SDE is a type of source code editor that knows the programming language grammar and uses this knowledge to guide the editing and the execution of a program. This type of editing ensures that a program is syntactically correct.
The editor is intended to provide both syntax-directed editing as well as manual text editing. A meta-language must be created to describe the grammar of the editor's target language. The meta-language will provide annotations to change the display of the text in the editor. That specification, written in the referred meta-language will be the input to generate templates for the syntax directed editor.
The SDE generator is available in a standalone JAR application as well as a web version.

RESUMO

O objetivo do trabalho de mestrado aqui relatado é desenvolver um sistema capaz de gerar um editor dirigido pela sintaxe (SDE) para uma qualquer linguagem. Um SDE é um editor de texto que tem conhecimento da gramática da linguagem e usa esse conhecimento para guiar o utilizador na edição e execução do programa assegurando assim que o programa esteja sintaticamente correto.
O editor será capaz de dar ao utilizador a habilidade de editar o programa tanto através de comandos dirigidos pela sintaxe como digitando o programa. Para isso é preciso criar uma meta linguagem para descrever a gramática da linguagem-alvo do dito editor. A linguagem permite ao utilizador anotar a gramática de modo a mudar a aparência do texto no editor. Esta meta linguagem seria depois usada para gerar os templates necessários para o editor dirigido pela sintaxe.
O gerador de SDE está disponível numa versão JAR standalone e numa versão web.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1

INTRODUCTION

This report describes the project carried out to fulfil the thesis requirement to conclude the 2nd year of the Master's Degree in Software Engineering of the Department of Informatics, School of Engineering, University of Minho.

1.1 BACKGROUND

Developing good programs is a difficult task. It requires effort and time from the user to analyze the program, design the algorithm that solves it, and to effectively writing the program.
A simple text editor is not enough to provide the assistance required for such tasks. Developing software using only a text editor is slow, tedious and error prone. We need editors to incorporate tools that make the development easier and less time consuming. These tools can range from the traditional debuggers to smart editors aimed to increase the readability of the program and to speed up their writing.
Manually typing is not the only way to develop. Syntax-directed editing, or structure editing, is a mechanism that being aware of the language grammar provides the user with commands to edit the programs strictly following the underlying syntax. These commands offered by the editor interface ensures the syntactic correctness of a program. This type of editing is useful for the programmer and it is important to expand it.
Even though these language dependent editors are advantageous it is not practical nor easy to develop one editor for every existing language as right now there exists a considerable number of programming languages and there are always new ones being developed.
This is why it is important to develop programs capable of generating such syntax-directed editors (SDE).

1.2 RESEARCH HYPOTHESIS

Given a programming language grammar written in a meta language it is possible to generate a syntax directed editor (SDE) for that language. The resulted editor will be intuitive and easy to use and will ease the target language learning process for a beginner user.

1.3 PROJECT DESCRIPTION, OBJECTIVES

The goal of this project is to create a system capable of generating a syntax directed editor (SDE) for any programming language given its grammar.
This will be achieved by creating a meta-language. This meta-language will describe the programming language grammar.
The generator will then take the meta-language and will produce the respective templates to be used by the SDE.
The resulted SDE will allow the user to write the program by editing both the program text and the program tree.

Chapter 2 will describe what is a text editor as well as features that are usually associated with it. It will also present the notion of SDE and structure editing. Lastly it will define what are SDE generators and it will present several examples of SDE generators. In Chapter 3 will be present the requirements of the SDE generator and its architecture. Chapter 4 will describe the implementation. In Chapter 5 the web version of the SDE generator will be discussed. Finally Chapter 6 will present the use of the system.

2

STATE OF THE ART

In this chapter will be presented the definition of source code editors. Next it will be discussed what it is a SDE and how the edition is made in those editors. After it will be explained what is a language workbench and several examples will be displayed. Finally several examples of existing SDE generators will be show.

2.1 TEXT EDITORS

A text editor is a program used to write text files. It is used both for mundane tasks, such as simply writing an email, and for more complex ones, such as writing a program.
Source code editors are enhanced text editors designed to create and edit programs source code. To help the user to write code more efficiently source code editors are able to provide many features, such as syntax highlighting and code completion. Notepad++ [1] and Sublime Text [2] are examples of well-known source code editors.
A source code editor can be included in an integrated development environment (IDE). This piece of software, in addition to edit source code, is also able to compile, deploy and debug software packages.
However the program representation and manipulation in these editors are still textual and the programming support supplies single features rather than making the impression of a thoroughly designed, specialized tool for program editing (Minör, 1992).

2.1.1 *Editing Features*

Features that allow the user to read, navigate and edit source code and that ease the programming task are so important that it is to be expected from a source code editor to provide them. The most common features present in source code editors are the following:

1 https://notepad-plus-plus.org/
2 https://www.sublimetext.com/

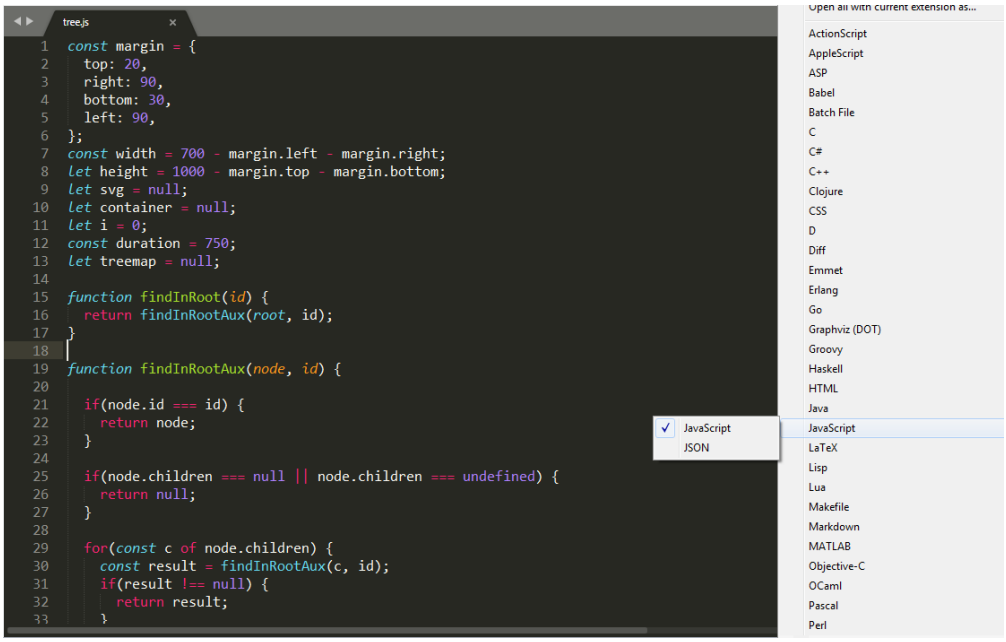


Figure 1: Slublime Text syntax highlighting options

Syntax Highlighting

Syntax highlighting allows faster reading of the code by highlighting keywords such as 'IF' and 'WHILE'; identifiers and literals are usually printed in different colors. Syntax highlighting also includes braces matching, i.e., it highlights the opening and the respective closing bracket. This allows an easier way to interpret nested instructions. Syntax highlighting is expected to be provided by every IDE and source code editor. As keywords vary from language to language source code editors allow the user to choose the adequate syntax highlighting (Figure 1).

Code Completion

Code completion provides a list with valid options that the user can choose while he's typing. This speeds up the coding process as the user doesn't have to write the full name. It also liberates the user from inspecting other files to lookup up the elements. For example, a user can see the available methods of a Java class after typing the class name. Documentation can be present in the menu list alongside their respective method or function.

Figure 2 depicts how code completion works on Netbeans[3]. The user calls the code completion by pressing *ctrl+space*[4] then navigates to the option he wants to insert and then

3 https://netbeans.org/
4 Notice that in some environments, code completion is always on and the user has no need to call it.

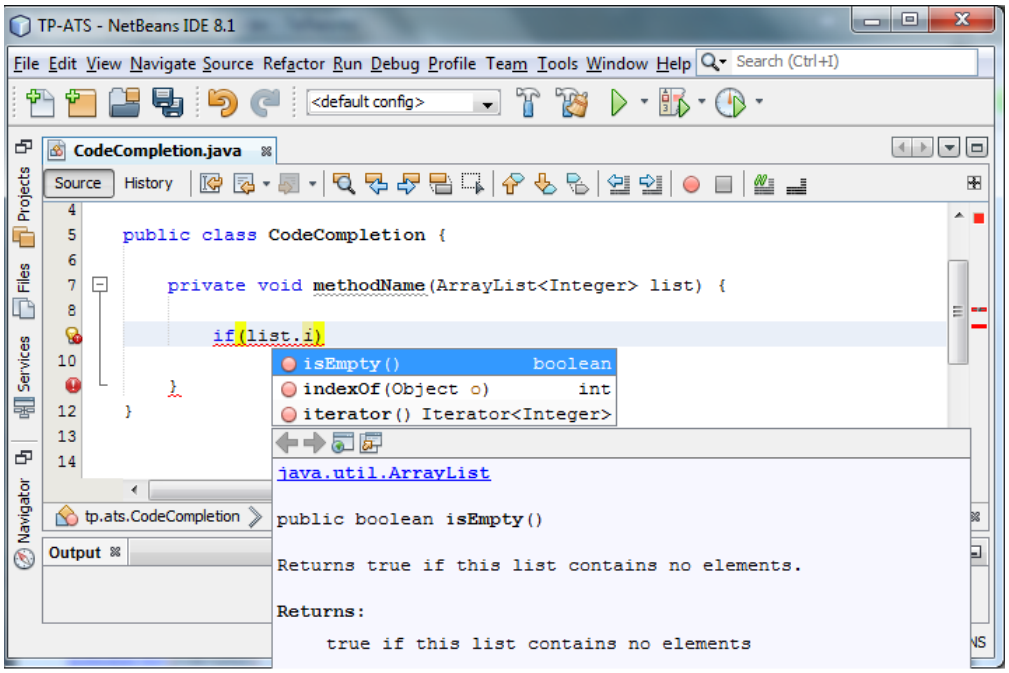


Figure 2: Netbeans code completion

presses enter to select it. If there is only one valid option, the list is not showed and the option is automatically inserted.

Code formatting

Some text editors offer an option to automatically format a program. This is a useful tool as consistent indentation in a program makes it comprehensible and easier to read and maintain. It is included in most IDEs which allow the user to define his own coding style, for example the user can choose the size of tabs, where to place spaces and define the brace positioning (Figure 3).

2.2 SYNTAX-DIRECTED EDITORS

A syntax-directed editor (SDE) (Teitelbaum and Reps, 1981; Lunney and Perrott, 1988; Zelkowits, 1984; Reps, 1982) is a type of source code editor that knows the programming language grammar and uses this knowledge to guide the editing and the execution of a program. This ensures that a program is always syntactically correct.
A program is constructed top-down by inserting either a template or a phrase on placeholders at cursor position.
A template is a pattern of code consisting of keywords, punctuation and non-terminals.

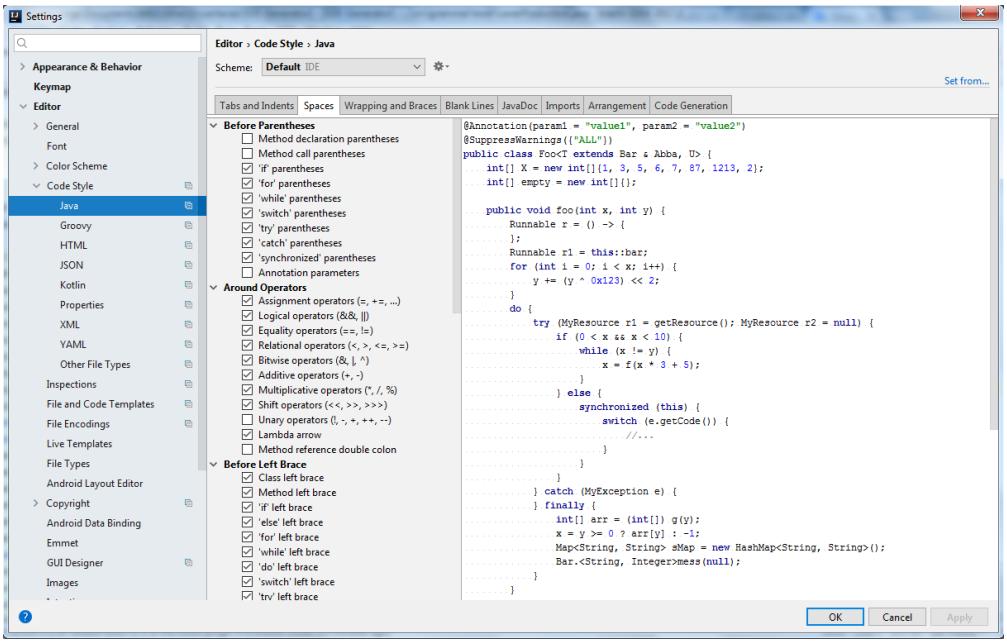


Figure 3: Coding Styling Settings in IntelliJ

These templates are predefined and correspond to the grammar rules of the language. Because a template cannot be changed it ensures that it contains no error. The following example represents a template for the if conditional statement, where *condition* and *statement* are placeholders.

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

Listing 2.1: Example of a IF conditional template

These placeholders can be expanded to continue the writing of the program by choosing the next valid template or phrase to insert.
A phrase is a user typed piece of text, for example the name of a user-defined variable, a numerical value or a string.
All modifications of the program occur at the current position of the editing cursor. The cursor can only be positioned on a template or on a placeholder. Inside a phrase the cursor can be anywhere.

```
program example{
    declarations
        var -> boolean;
        <variable-declaration>
    statements
        if(a > 1)
```

```
        then{
            <statement-list-1>
        }
        <statement-list-2>
}
```

Listing 2.2: SDE

In Listing 2.2, < statement-list-1>, <statement-list-2> and <variable-declaration> are placeholders yet to be expanded. It can also be seen that a variable declaration template was fully expanded to *var -> boolean* as well as the the conditional statement in the if statement to *a>1*.
Usually in a SDE the program is internally represented as an abstract syntax tree (AST).
Compared to text editors, a SDE requires less typing from the user as most of the writing during programming is by typing keywords and punctuation and these elements are included in the predefined templates. Another advantage of using a SDE is that a user is not required to extensively know the programming language syntax because the editing is guided by it.

2.3 Text Editors Generators

Even though source code editors and SDEs are powerful tools that eases the development of programs, it is not practical to develop them for every language as it can be time-consuming and error-prone. This is why it is important to develop language independent editors.

2.3.1 *Language Workbenches*

Language workbenches, a term coined by Martin Fowler (2005), are tools aimed to implement new languages as well as theirs IDEs. In addition to ease the development of languages, they also make language-oriented programming environments practical.
According to Fowler, a language workbench needs to have the following characteristics:

- Users can freely define new languages which are fully integrated with each other.
- The primary source of information is a persistent abstract representation.
- A Domain-Specific Language (DSL) is defined in three main parts: schema, editor(s), and generator(s).
- The users manipulate the DSL through a projectional editor

- A language workbench can persist incomplete or contradictory information in its abstract representation.

Language workbenches can be referred to as being textual (xText and Spoofax), graphical (MetaEdit+ and DOME) or projectional (JetBrains MPS) (Erdweg et al., 2013).
In projectional language workbenches (Voelter and Pech, 2012; Fowler) all text, symbols and graphics are projected. They manipulate the abstract representation of the program and offer the programmer ways to edit the abstract model structure directly. With this type of program editing, they resemble to SDEs.

xText

xText is a textual language workbench currently developed by Eclipse under the Eclipse TMF (Textual Modeling Framework) project.
It allows the user to specify a language (ranging from small Domain-Specific Languages to full-blow General Purpose Languages (Eysholdt and Behrens, 2010)) and their respective tools.
xText generates the following artifacts (Efftinge and Völter, 2006):

- a set of abstract syntax tree (AST) classes represented as an eclipse model framework (EMF) based metamodel
- a parser that can read the textual syntax and returns an EMF-based AST (model).
- a number of helper artifacts to embed the parser in an openArchitectureWare workflow
- an Eclipse editor that provides syntax highlighting, code completion, code folding, a configurable outline view and static error checking for the given syntax.

```
grammar org.example.domainmodel.Domainmodel with
                                        org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Domainmodel :
    (elements+=Type)*;

Type:
    DataType | Entity;

DataType:
    'datatype' name=ID;

Entity:
```

```
    'entity' name=ID ('extends' superType=[Entity])? '{'
        (features+=Feature)*
    '}';

Feature:
    (many?='many')? name=ID ':' type=[Type];
```

Listing 2.3: Grammar defined in xText

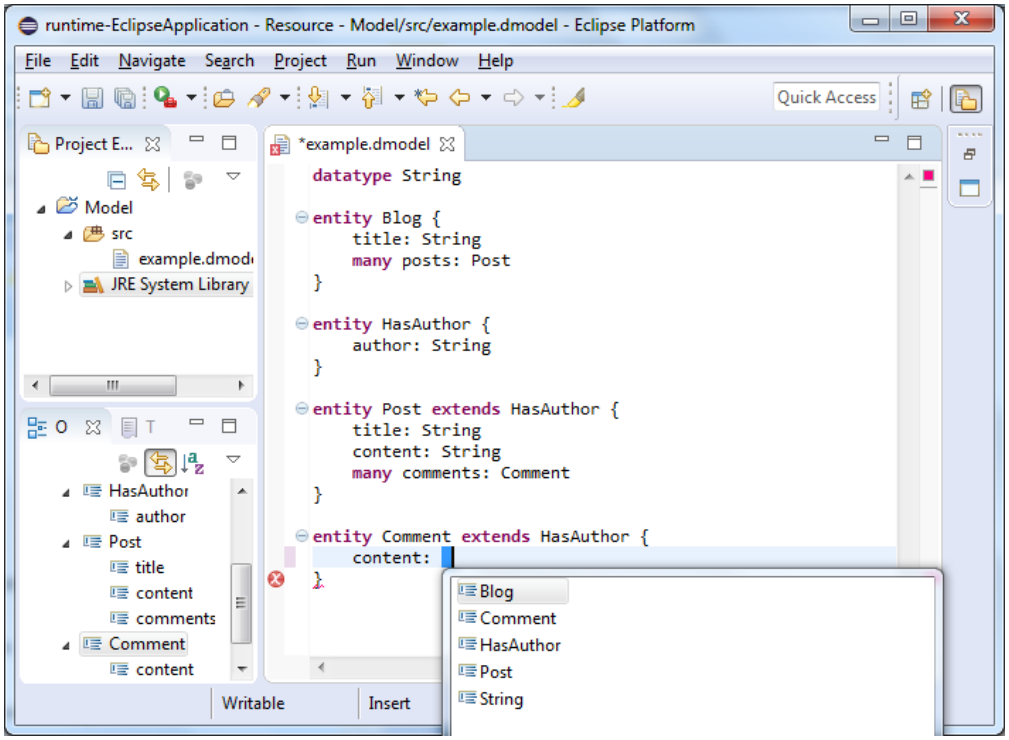


Figure 4: xText generated editor

Figure 4 shows the editor that was generated for the language defined in Listing 2.3. The editor is based on the eclipse IDE and contains features such as syntax highlighting and code completion.

Spoofax

Spoofax[5] is an Eclipse based workbench that provides syntax definition, program transformation, code generation, and declarative specification of IDE components (Völter and Visser, 2010). In Spoofax the grammars are specified using the modular Syntax Definition Formalism (SDF). SDF grammars are highly modular, combine lexical and context-free syntax into one formalism, and can define concrete and abstract syntax together in production

5 http://www.metaborg.org/en/latest/

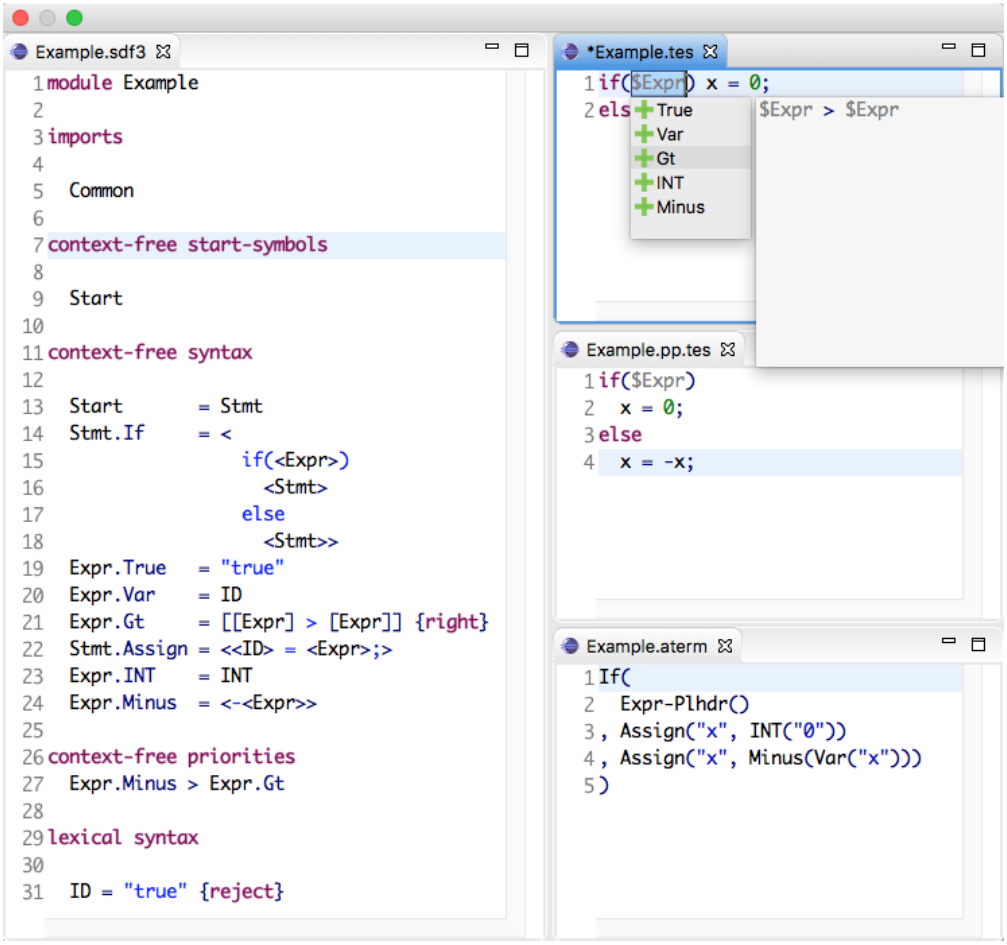


Figure 5: Spoofax Editor

rules. Spoofax uses the Stratego program transformation language to describe the semantics of a language.

Figure 5 shows on the left side a language definition in Spoofax. On the right side shows a program of that language being edited as well as its pretty printed version and its abstract syntax tree.

MetaEdit+

MetaEdit+ [6] (Tolvanen et al., 2007; Tolvanen and Rossi, 2003; met, b,a) is an environment for developing and using domain-specific modeling languages. MetaEdit+ allows building modeling tools and generators fitting to specific application domains. It provides a meta modeling language and tool suite for defining the method concepts, their properties, associated rules, symbols, checking reports and generators. MetaEdit+ uses the GOPPRR (Graph-

6 http://www.metacase.com/mwb/

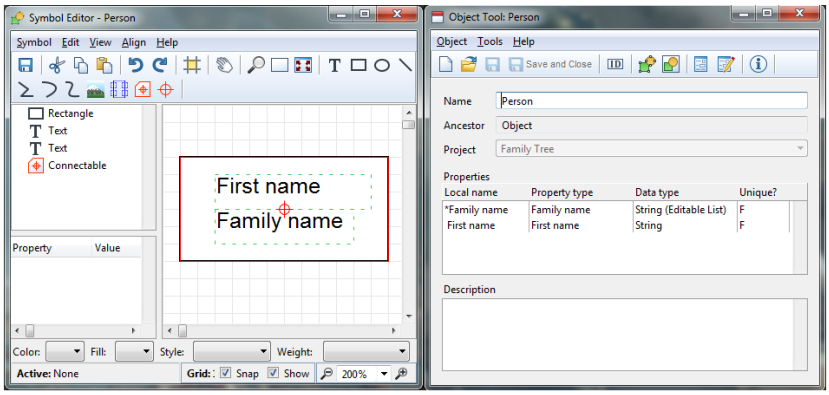


Figure 6: Object Definition in MetaEdit+

Object-Property-Port-Role-Relationship) metamodeling framework. Each of the name of the GOPPRR is called a metatype.

- **Graph:** collection of objects, relationships, roles and bindings of these
- **Object:** main elements of graphs
- **Relationship:** connection between two or more objects
- **Role:** specifies how an object participates in a relationship
- **Port:** optional specification of a specific part of an object to which a role can connect
- **Property:** describing characteristic associated with the other types, such as a name, an identifier or a description.

First the user creates the objects of the modeling language. Figure 6 depicts the creation of a object person and its symbol. A person object has two properties, first name and family name. It will be represented by a box with the first name and family name inside it.

The diagram editor is depicted in figure 7

JetBrains MPS

JetBrains MPS [7] (Pech et al., 2013) is a language workbench based on a projectional editor. Firstly the user defines the abstract syntax of the language by creating concepts. This resembles object-oriented programming as the concepts are similar to classes. Each concept represents a node of the AST.

7 https://www.jetbrains.com/mps/

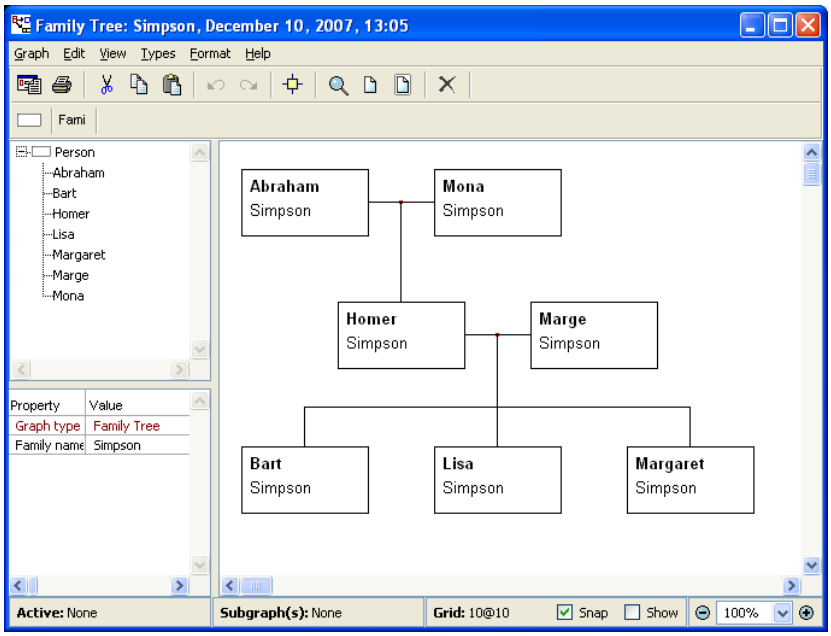


Figure 7: Diagram Editor in MetaEdit+

Figure 8 depicts the creation of the concept *Script* in MPS which, as well as the *Library* script, can be the root of the language. It has one child of the concept type *CommandList* and can have zero or more children of the type *RoutineDefinition*. Concepts can also have properties, references to other concepts in the AST and alias, which is a string that will be recognized by MPS as a representation of the concept.

Just like classes, concepts can be defined abstract and other concepts can extend them. The *CommandList* concept can have one or more children of the abstract concept *AbstractCommand*. The right side of the editor in Figure 9 shows the concepts that extend the *AbstractCommand* concept.

After defining the structure of the language the user defines the editor for the concepts. Here the user defines how the concepts are displayed in the editor. Figure 10 shows the definition of the editor for the *IfStatement* concept. The cells delimited by % represents the children concepts of the concept. In the bottom of the editor the user defines the styling for each cell such as indent style and font size. The cell *?[- (...) -]* represents that the cell is optional.

The resulted editor can be seen in Figure 11. The user can continue the editing by either typing or hitting *ctrl+space* and choosing the appropriate option.

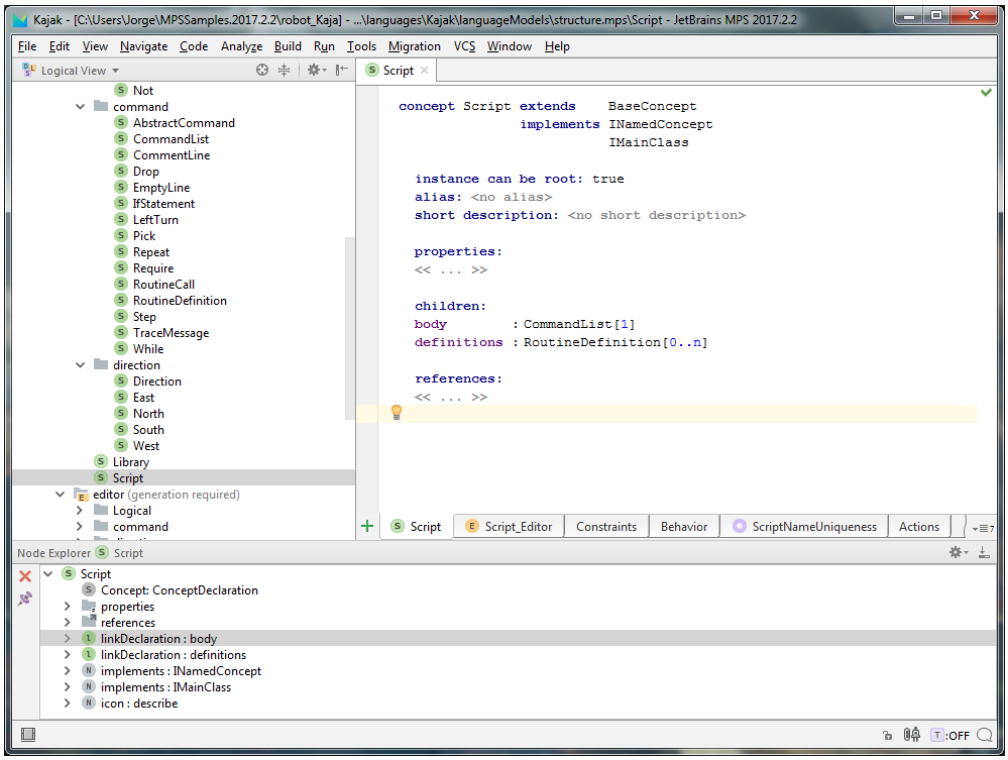


Figure 8: Concept Definition in MPS

2.3.2 *SDE generators*

Many syntax editors generators have been built in the past years:

In Arefi et al. (1990) is described a syntax directed editor generator for visual programming languages. The user defines the syntax and semantics of the programming language and the generator will output its syntax-directed editor. Figure 12 depicts a generated editor.

ASF+SDF Meta-environment

The ASF+SDF Meta-environment (van den Brand et al., 2001) allows the user to write language definitions. It provides the user with a parser, prettyprinter, syntax-directed editor, debugger, and interpreter or compiler for the language specification.

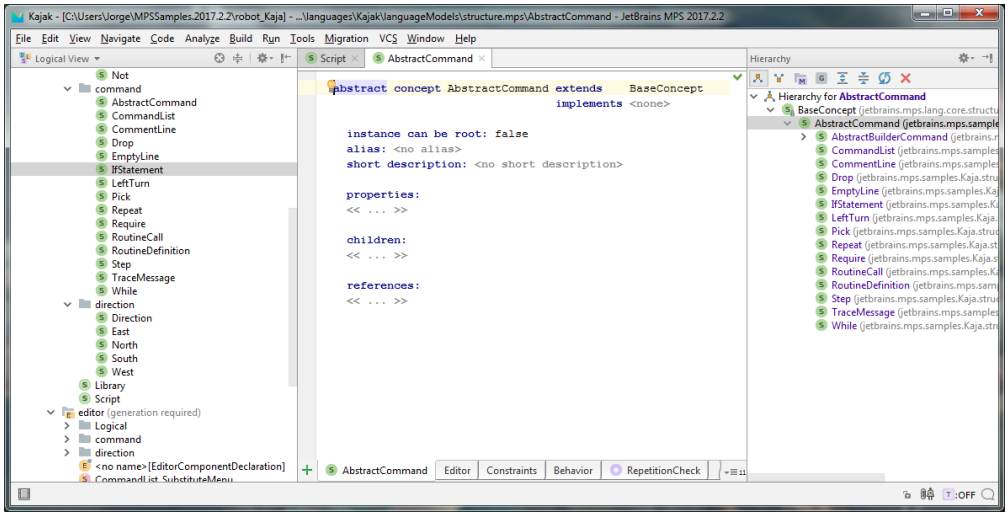


Figure 9: Abstract Concept in MPS

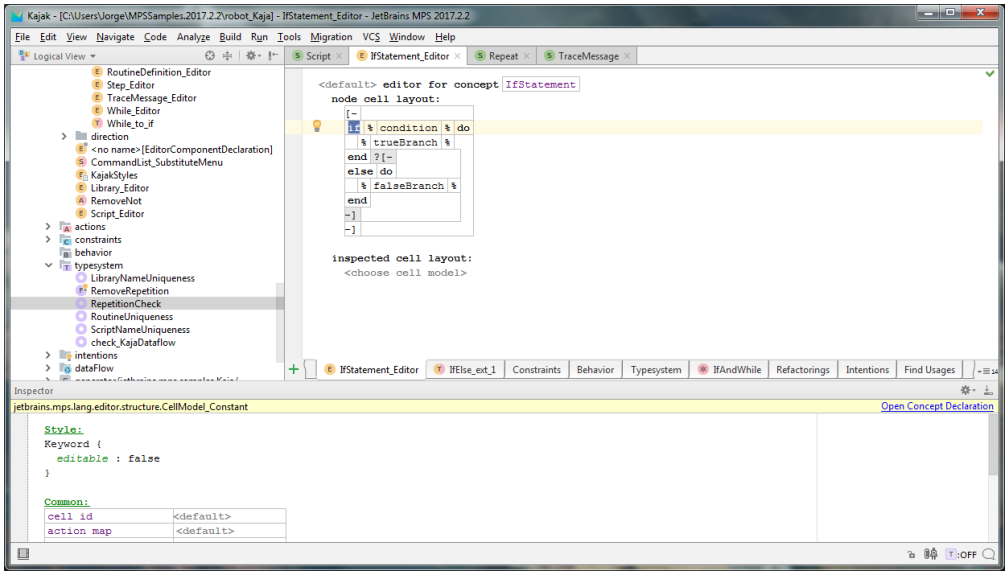


Figure 10: Editor Definition in MPS

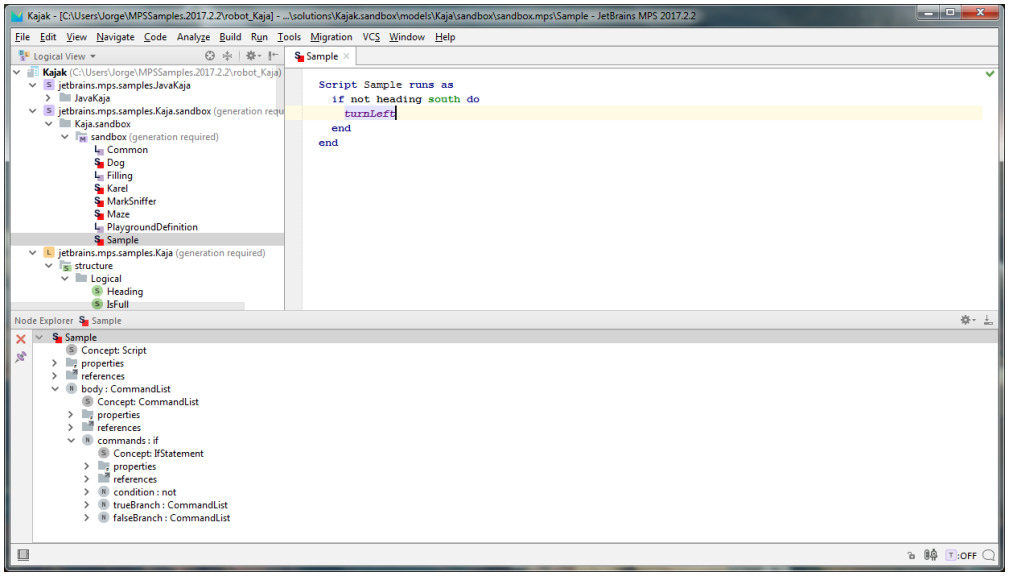


Figure 11: MPS Resulted Editor

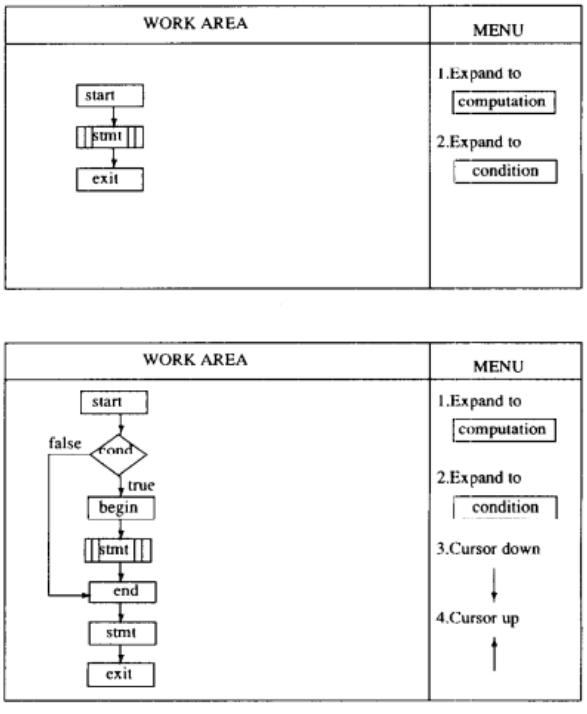


Figure 12: Resulted Editor

Synthesizer Generator

In the Synthesizer Generator (Reps and Teitelbaum, 1984, 2012) the user provides an attribute-grammar specification that includes rules defining the abstract syntax, attribution, display format and concrete input syntax. This specification is written in the Synthesizer Specification Language (SSL). The editor generated represents the program as an attributed derivation tree. The Synthesizer Generator has also been used to create proof checkers and text-formatting editors.

In Listing 2.4 the lexemes of the language are defined. The lexemes are defined in the following form:

```
phylum-name : lexeme-name < regular-expression >;
```

It declares that all strings generated by the given regular-expression are in the named phylum.

```
  WHITESPACE:     < [\ \t\n] >;
  AGENDA:         < "AGENDA:" >;
  END:            < "END." >;
  GROUP:          < "***(" >;
  ENDGROUP:       < "*)"   >;

  NUMERO:         < [0-9\.\-\(\)]+ >;
  TEXTO:          < [a-zA-Z\/\~\:\@\.\-\_\?\!0-9\ ]+ >;
```

Listing 2.4: Lexemes

The concrete input syntax is displayed in listing 2.5

```
AG ::= (AGENDA Id ItemList END)
         {$$.t = Agenda(Id.t, ItemList.t);};

  ItemList ::= (Item) {$$.t = (Item.t :: ItemListNil);}
             |  (Item ItemList) {$$.t=(Item.t::ItemList$2.t);}
             ;

  Item ::= ('+' Id Num Num Mor EM U)
           {$$.t = Single(Id.t, Num$1.t, Num$2.t, Mor.t, EM.t, U.t);}
         | (GROUP Id ItemList ENDGROUP)
                {$$.t = Group(Id.t, ItemList.t);}
         ;

  Id ::= (TEXTO) {$$.t = Identifier(TEXTO);};
  Mor ::= (TEXTO) {$$.t = Morada(TEXTO);};
  EM ::= (TEXTO) {$$.t = Email(TEXTO);};
```

```
U ::= (TEXTO) {$$.t = Url(TEXTO);};
Num ::= (NUMERO) {$$.t = Numero(NUMERO);};
```

Listing 2.5: Concrete Syntax

```
AG    {syn ag t;};
Item    {syn item t;};
ItemList  {syn itemlist t;};
Id    {syn identificador t;};
Mor     {syn morada t;};
EM    {syn email t;};
U     {syn url t;};
Num     {syn num t;};

ag ~ AG.t;
item ~ Item.t;
itemlist ~ ItemList.t;
identificador ~ Id.t;
morada ~ Mor.t;
email ~ EM.t;
url ~ U.t;
num ~ Num.t;
```

Listing 2.6: Association Rules

The display form of a term is determined by unparsing declarations which determine the textual representation of the term, the selectable components of the term and the default editing modes of those selectable components.

```
ag : Agenda [ @ ::= "AGENDA: " @
            "     Registos: " itemlist.acount
            "%t%n" @ "%b%nEND." ];

itemlist : ItemListNil  [ @ ::= ]
         | ItemListPair [ @ ::= @ ["%n"] @ ];

item : ItemNull [ @ ::= "<item>" ]
     | Single   [ @ ::= "--> " @
                  " : " @ " : " @ " : "
                  @ " : " @ " : " @ ]
     | Group    [ @ ::= "*** [" @ "]%t%t%n" @ "%b%b%n***%n" ];

identificador : IdentNull  [ @ ::= "<identificador>" ]
              | Identifier [ @ ::= @ ]
              ;

morada : MoradaNull  [ @ ::= "<morada>" ]
```

```
        | Morada      [ @ ::= @ ]
        ;

email : EmailNull  [ @ ::= "<email>" ]
      | Email      [ @ ::= @ ]
      ;

url : UrlNull  [ @ ::= "<url>" ]
    | Url      [ @ ::= @ ]
    ;
num : NumNull [ @ ::= "<numero>" ]
    | Numero  [ @ ::= @ ]
    ;
```

Listing 2.7: Unparsing Rules

The grammar can be extended by associating attributes to a grammar production. The user defines the attributes in the following form:

```
phylumo {
        synthesized type attribute-name ;
        inherited type attribute-name ;
} ;
```

The keywords synthesized an inherited can be abbreviated as *syn* and *inh* respectively.

```
itemlist,item {inh INT bcount;
               syn INT acount;
               };

ag : Agenda {  itemlist.bcount = 0; };

itemlist : ItemListNil  { $$.acount = $$.bcount;}
                | ItemListPair { item.bcount = itemlist$1.bcount;
                                 itemlist$2.bcount = item.acount;
                                 itemlist$1.acount = itemlist$2.acount;
             };

item : ItemNull { $$.acount = $$.bcount; }
         | Single   { $$.acount = $$.bcount+1; }
         | Group    { itemlist.bcount = $$.bcount;
                          $$.acount = itemlist.acount; }
           ;
```

Listing 2.8: Attributes

The previous definition of a grammar will produce a generator with an initial state depicted in Figure 13.

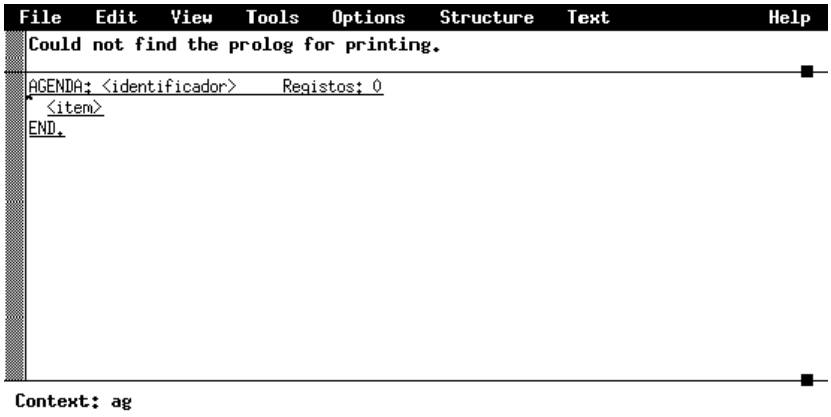


Figure 13: Synthesizer Generator Initial State

After selecting the <*item*> placeholder (Figure 14) we can either choose the context *Simples* (Figure 15) or the context *Grupo* (Figure 16).

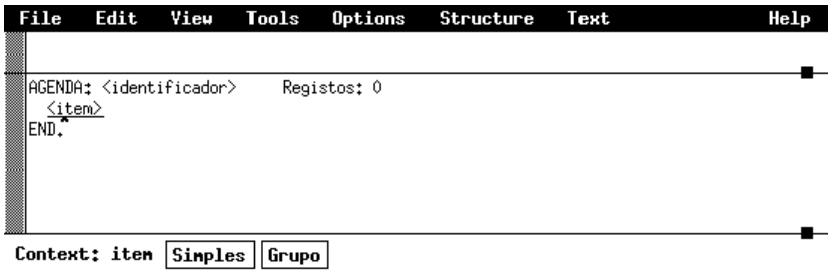


Figure 14: Item Selected

Figure 17 depicts the full text created with the editor.

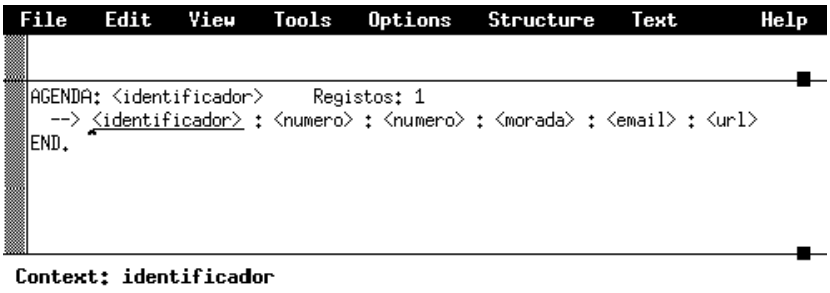


Figure 15: *Simples* Selected

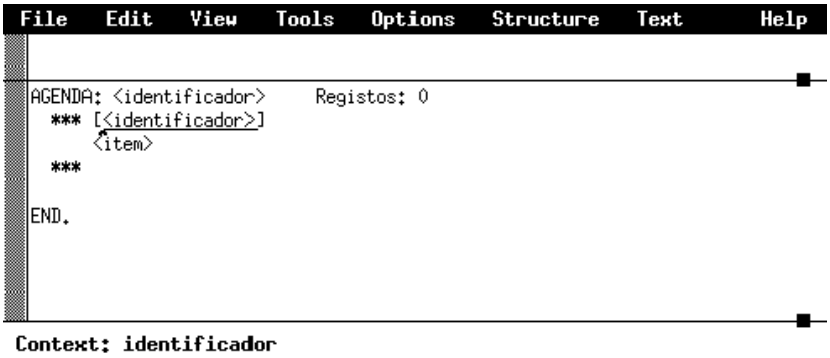


Figure 16: *Grupo* Selected

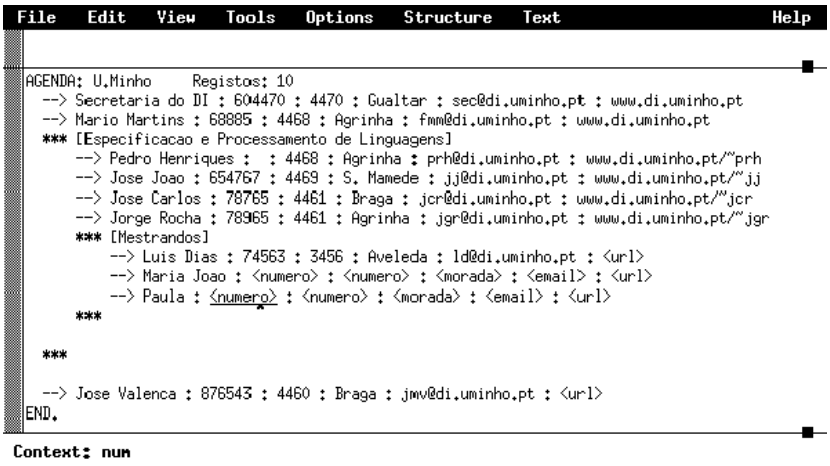


Figure 17: Final State

LISA

The LISA system [8] (Mernik et al., 2000; Henriques et al., 2005, 2002) generates an interpreter from an attribute grammar based language specifications. LISA generates two types of

8 https://labraj.feri.um.si/lisa/

editors, a language knowledgeable editor and a syntax-directed editor LISA also generates inspectors, debuggers and visualizers for the given language definition.

```
language Robot {
  lexicon {
      Command       left | right | up | down
      ReservedWord  begin | end
      ignore        [\0x0D\0x0A\ ]
  }

  attributes int *.inx; int *.iny;
             int *.outx; int *.outy;

  rule start {
    START ::= begin COMMANDS end compute {
      START.outx = COMMANDS.outx;
      START.outy = COMMANDS.outy;
      COMMANDS.inx = 0;
      COMMANDS.iny = 0;
    };
  }
  (...)
}
```

Listing 2.9: Language definition in LISA

Figure 18 shows the structure view of a program written in the language defined in Listing 2.9. Even though it is said that LISA provides a syntax-directed editor it is not present in the version available to download. It is only possible to view the structure of the program.

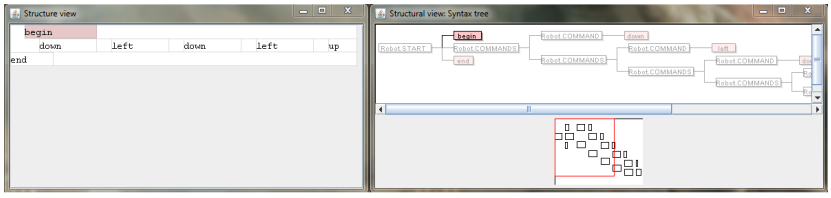


Figure 18: Structure view in LISA

2.4 SUMMARY

In the previous sections it was talked about the definition of source code editors as well as some of the features they include.
It was also presented the definition of a SDE and its advantages compared to a text-based

source code editor.
It was also addressed the importance of having systems that generate text editors. Then the definition of language workbenches was introduced as well as several examples of SDE generators were presented. The Diagen system was considered but didn't fit as a SDE generator as it is a diagram generator.

3

SDE GENERATOR: DESIGN

The main goal of this master's thesis is to develop a system able to generate syntax directed editors for any given language grammar. This grammar has to be defined in a meta language. The generated system shall provide a complete editing environment where the user can create and edit programs guided by the syntax tree of the language.

3.1 REQUIREMENTS

The generated editor must have the following features:

- Edit a program by syntax editing commands
- Edit a program by editing both the program tree and text without violating the syntax of the grammar
- Save and load valid text files to the editor
- Save and load the state of the edition without requiring the program to be syntactically correct

3.2 ARCHITECTURE

The user starts by providing the grammar to the system. The system will then generate the required templates to be used by the SDE to build the program tree. The parser and the lexer of the grammar will also be generated. When a text file is loaded the parser and lexer will be used to build the abstract tree which with the grammar templates will be used to generate the program tree and display it on the editor.

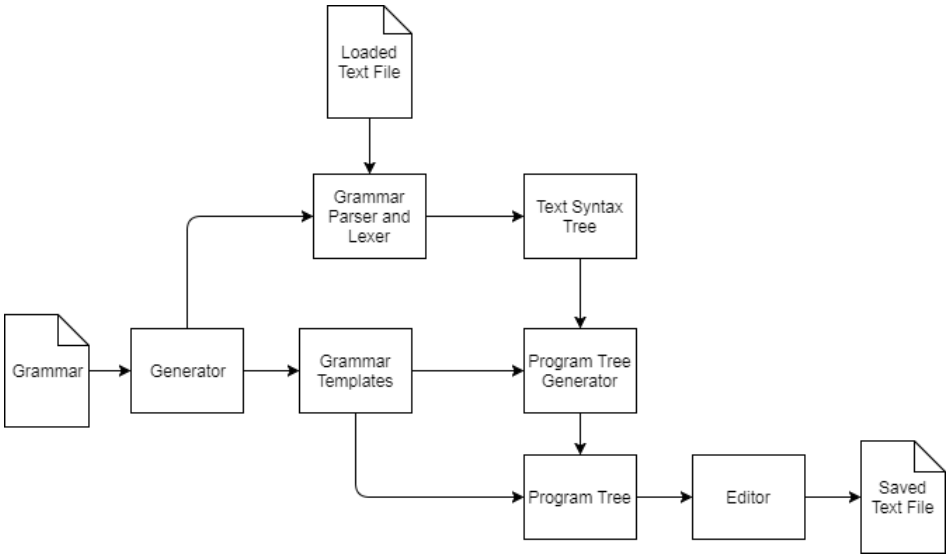


Figure 19: System Architecture

Figure 19 depicts the different components of the system. The Generator component is used to generate the grammar templates and the parser and lexer of the grammar. The Program Tree component represents the program in the editor and is modified by the user while editing. When loading a text file, the Program Tree Generator component will generate the program tree using the grammar templates and the text syntax tree.

4

SDE GENERATOR: DEVELOPMENT

This chapter will be used to describe the development of the SDE Generator. The development of the SDE generator was split unto four steps:

- Definition of the meta language
- Generation of the grammar templates
- Building the program tree
- Loading text files unto the editor

4.1 GRAMMAR

To generate the editor, the user has to supply a grammar defined in the meta language. This meta language is based on the ANTLR grammar definition with the particular feature that the user can annotate the production terms.

This annotations are optional so the system accepts pure combined ANTLR grammars. They can be placed either after a production term (Listing 4.2) or just before the grammar definition (Listing 4.1). The former type of annotation is applied to the production term immediately preceding the annotation, while the latter is applied globally on the grammar. A single production term can have multiple annotations (Listing 4.3).

The annotations may have arguments. Those that don't have arguments are written as @<*annotation name*> while the annotations with arguments are written as @<*annotation name*>=<*argument*>

```
@bc='\*'
@ec='*/'
grammar Gramatica;

programa : instrs
         ;
```

Listing 4.1: Annotations before grammar definition

```
instr : ifInstruction
      | whileInstruction
      | forInstruction
      | atr ';' @nl
      ;

ifInstruction : 'if' '(' cond ')' '{'@nlt instrs  '}' @nl elseCondition?
              ;

atr : ID@c=red '=' exp
    ;
```

Listing 4.2: Annotations in a grammar rule

```
whileInstruction : 'while' '(' cond ')' '{'  instrs '}' @c=red @nl
                 ;
```

Listing 4.3: Multiple annotations in a grammar rule

Without annotations, the text presented by the editor would be formatted as shown in Figure 20, while with the annotations presented in listing 4.2 the text is shown as in Figure 21.

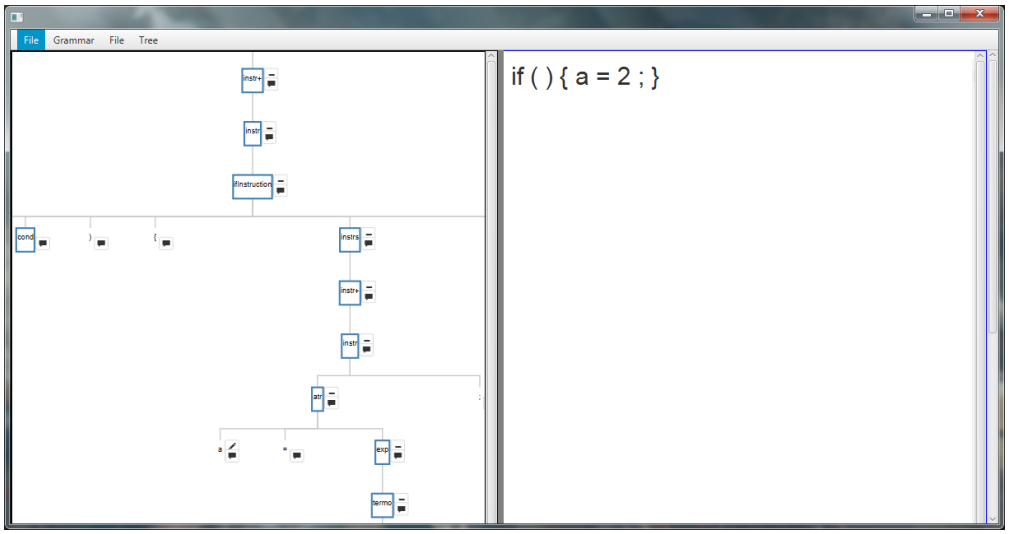


Figure 20: Editor without annotations

Table 1 displays all the annotations present in the meta language as well as their arguments and description.

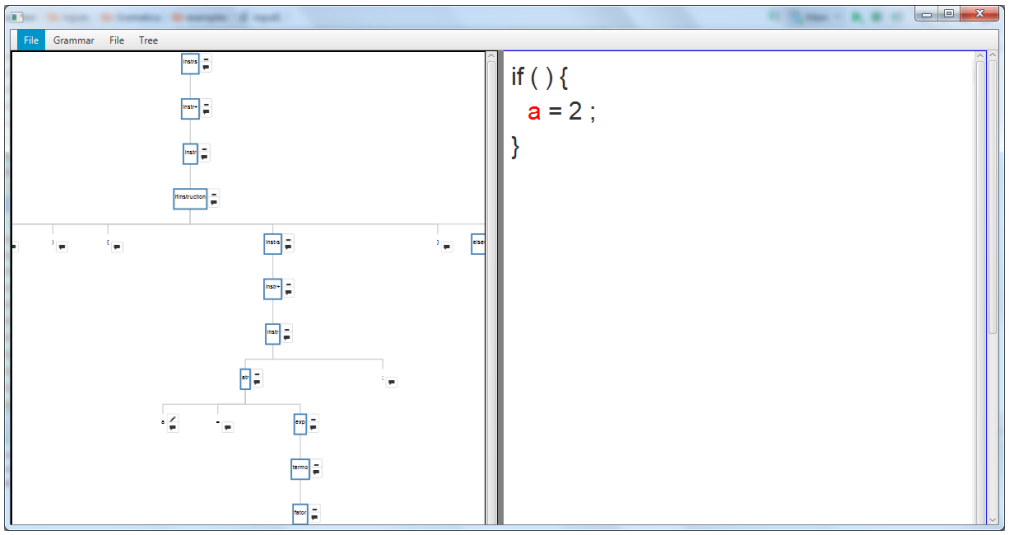


Figure 21: Editor with annotations

Annotation	Argument	Used to
@h	Text to be displayed	display a helpful tip on a tree node.
@nl	None	insert a new line in the right side of the editor.
@nlt	None	insert a new line followed by a tab.
@c	Color name	color a terminal node.
@bc	Char sequence in single quotes	define the beginning of a comment.
@ec	Char sequence in single quotes	define the ending of a comment.

Table 1: Annotations table

The annotations **@h**, **@nl**, **@nlt** and **@c** are placed after a production term, while the **@bc** and **@ec** annotations are placed before the definition of the grammar.

4.2 TEMPLATES

As referred previously, a SDE needs grammar templates to be able to build programs. In the SDE Generator the templates are built after the initial parsing of the grammar and are defined in JAVA classes. Templates depicts the grammar structure, its rules, productions, terms and annotations.

4.2.1 *Grammar Template*

The main class is the *Grammar* class (4.4) which represents the whole grammar. It stores the grammar name, the begin comment (@bc) and end comment (@ec) annotations as well as the parser and lexer rules templates.

```
public class Grammar implements Serializable {
    private String name;
    private ArrayList<ParserRule> parserRules;
    private ArrayList<LexerRule> lexerRules;
    private String beginComment;
    private String endComment;

   // (...)
}
```

Listing 4.4: Grammar class

4.2.2 *Rules Templates*

The *ParserRule* (4.5) and *LexerRule* (4.6) classes represent respectively a parser and a lexer rule of a grammar and contains the rule name and its productions.

```
public class ParserRule implements Serializable {
   private String name;
    private ArrayList<ParserProduction> productions;

   // (...)
}
```

Listing 4.5: ParserRule class

```
public class LexerRule implements Serializable {
    private String name;
    private String regex;
    private ArrayList<LexerProduction> productions;

   // (...)
}
```

Listing 4.6: LexerRule class

4.2.3 *Productions Templates*

Both *ParserProduction* (4.7) and *LexerProduction* (4.8) are simple classes that are only composed by a list of the productions terms.

```
public class ParserProduction implements Serializable {
```

```
    private ArrayList<ParserTerm> terms;

   // (...)
}
```

Listing 4.7: ParserProduction class

```
public class LexerProduction implements Serializable {
    private ArrayList<LexerTerm> terms;

   // (...)
}
```

Listing 4.8: LexerProduction class

4.2.4 *Terms Templates*

ParserTerm

A parser term (4.9) is characterized by its quantifier and by the new line (@nl or @nlt), color (@c) and helper (@h) annotations.

```
public abstract class ParserTerm implements Serializable {
    private String quantifier;
    private String newLine;
    private String color;
    private String helper;

   // (...)
}
```

Listing 4.9: ParserTerm class

The parser term is divided into three types: terminal, non-terminal and block.
The *Terminal* class, defined by its name

```
public class Terminal extends Term {
    private String name;
    private boolean editable;
    private boolean lexer;

  //(...)
}
```

Listing 4.10: Terminal class

The *Nonterminal* class is simply defined by its name which is a rule name.

```
public class Nonterminal extends Term {
    private String name;

  //(...)
}
```

Listing 4.11: Terminal class

The *Terminal* class, defined by its name

A block term is defined as a set of terms surrounded by parentheses. A block can be separated into alternatives by a vertical bar. In 4.12 *(attribute | function | ruleClauses)* is a block that has three alternatives.

```
form : (attribute | function | ruleClauses) '.' ;
```

Listing 4.12: Rule with a block term

It is represented by the *lst:blockClass* where the alternatives are lists of lists of *ParserTerm*.

```
public class Block extends Term {
    private ArrayList<ArrayList<ParserTerm>> alternatives;
    private boolean lexer;

  //(...)
}
```

Listing 4.13: Block class

LexerTerm

The *LexerTerm* class (4.14) is characterized by its quantifier and it is divided into *SimpleLexerTerm* and *LexerBlock* classes.

```
public abstract class LexerTerm implements Serializable {
    private String quantifier;

  //(...)
}
```

Listing 4.14: LexerTerm class

The *LexerSimpleTerm* class is defined by the name and by a boolean that checks if the term is non-terminal or not (if it points to a lexer rule or not)

```
public class LexerSimpleTerm extends LexerTerm {
    private String name;
    private boolean terminal;

  //(...)
}
```

Listing 4.15: LexerSimpleTerm class

Just like the *Block* class the *LexerBlock* is defined by a list of lists of alternatives.

```
public class LexerBlock extends LexerTerm {
    private ArrayList<ArrayList<LexerTerm>> alternatives;

  //(...)
}
```

Listing 4.16: LexerBlock class

4.2.5 *Lexer Regular Expressions*

After generating the lexer terms and productions templates the the lexer rule template is completed by composing the whole regular expression. This is done by joining all productions and terms template names. If the name refers to another lexer rule its regular expression is returned.

The regular expression of the ***ID*** lexer defined in Listing 4.17 rule will be:

```
([a-z]|[A-Z])(([a-z]|[A-Z])|[0-9])+
```

```
ID  : LETTER (LETTER|DIGIT)+
    ;
DIGIT : '0'..'9'
    ;
LETTER: 'a'..'z'
    |'A'..'Z'
    ;
```

Listing 4.17: Lexer Definition

Since *JavaScript* is used in the SDE generator ANTLR regular expressions need to be transformed to *JavaScript* regular expressions, for example the ANTLR interval of characters, *'a'..'z'*, is transformed to [a-z].

4.3 STRUCTURE OF THE PROGRAM TREE

The tree structure of the program built is represented in a JSON object where the structure of each node of the tree is displayed in Listing 4.18.

```
{
  "data": {
    "name": <name of the node>,
    "term": <Java object of the term>,
    "quantifier": <term quantifier>,
    "type": <type of the term, either normal or block>
  },
  "height": 0,
  "depth": 0,
  "parent": <id of the node>,
  "id": <node id>,
  "children": <node children>
}
```

Listing 4.18: JSON Tree Structure

When a node tree is left clicked the program checks if the tree node can be expanded. A node can be expanded if it isn't a terminal or it has the + or * quantifier.

When an expandable non terminal node is clicked the program will search on the grammar templates for the rule with the same name as the node. When found, it will add to the node children the production terms as new nodes of the tree.

4.4 TEXT LOADING

When the user loads a valid text file it has to be transformed into the JSON object described previously to be displayed in the program. To do that the syntax tree of the text is generated. Figure 22 depicts the syntax tree of the following text: **2*(2-5)**

```
grammar Exp;

eval: additionExp
    ;

additionExp: multiplyExp ( '+' multiplyExp | '-' multiplyExp )*
```

```
            ;

multiplyExp: atomExp ( '*' atomExp | '/' atomExp )*
            ;

atomExp :     Number
        |     '(' additionExp ')'
        ;

Number :     ('0'..'9')+ ('.' ('0'..'9')+)?
       ;
```

Listing 4.19: Example Grammar

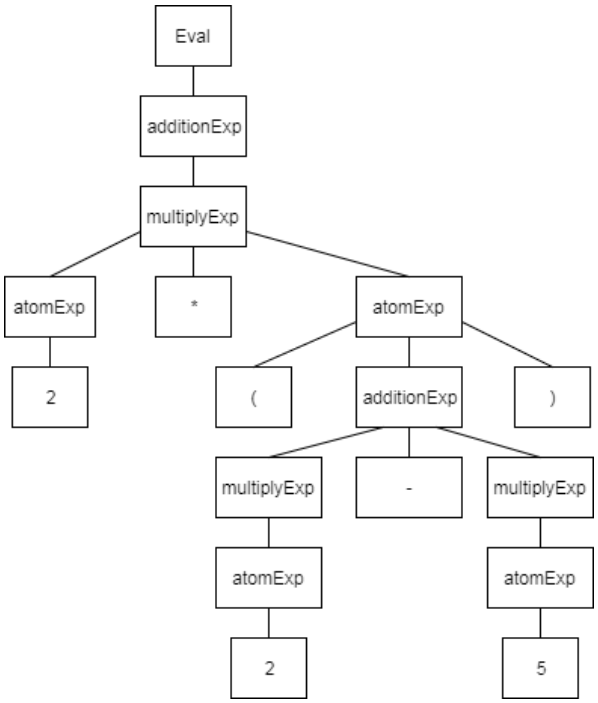


Figure 22: Text Syntax Tree

To generate the JSON program structure, firstly the node children will be compared with productions of the grammar rule that matched the parent node to find the adequate production. After finding it the process repeats for each child node.

A tree node and a grammar term match:

1. If the grammar term is non-terminal and the tree node name equals the term name
2. If the grammar term and the tree node is terminal and the term regular expression matches the name of the tree node

When a term and a node are matched the following term and node are compared. When a term has the **+** quantifier or the ***** quantifier if there was a match in the first node the next nodes are compared with the same term until there is no match. A block term is compared to a node by comparing its alternatives terms to the node, just like comparing a rule production.

If a node doesn't match with a term, the search breaks and starts over in the next rule production. When a term with the ***** quantifier or the **?** quantifier doesn't match with the node, the search doesn't break and it is added an empty node to the JSON to represent zero occurrences. A set of nodes won't match to a block term if it didn't match to any alternatives of the block.

Firstly the first child node of the tree, the ***additionExp*** node, is compared with the first term of the first production of the start rule, the ***additionExp*** term. Since they match and there is no more nodes nor terms in the production to be compared the search will start over with the children of the ***additionExp*** node and the first production of the ***additionExp*** rule.

Next the ***multiplyExp*** node will be compared with the ***multiplyExp*** term and there is a match. The block ***('+' multiplyExp | '-' multiplyExp)**** won't match with anything since there are no more nodes to be compared.

Next the ***atomExp*** node will be compared with the ***atomExp*** term and there is a match. The following nodes will match with the first alternative of the block.

After the execution of the algorithm the JSON object will be built and both the text and the tree will be displayed in the editor.

5

SDE GENERATOR: WEB APPLICATION

Since great part of the application was developed in JavaScript it made sense to develop a web version for the SDE generator.
In this chapter the architecture of the web application will be explained and the web application will be presented.

5.1 ARCHITECTURE

The web server has two endpoints to interact with the client, one for uploading the grammar the other to upload the text file.

When uploading the grammar (Figure 23) the client sends a post request with the grammar file. The server then generates the templates and the grammar parser and lexer files. To distinguish the produced files from different grammars they are saved in a folder whose name is the SHA-1 hash of the grammar file. The server then responds to the client with the grammar hash and the grammar templates in JSON format.

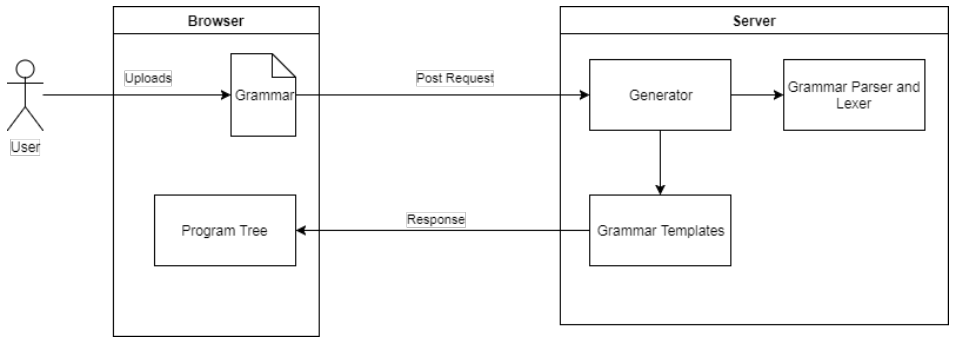


Figure 23: Loading a grammar

When uploading a text file (Figure 24) the client sends a post request with the text file and the hash of the grammar file received when uploading the grammar. The server locates the folder where the grammar parser and lexer are stored and generates the Text Syntax

Tree. The server responds to the client with the syntax tree in JSON format which with the grammar templates will be used by the client to build the program tree.

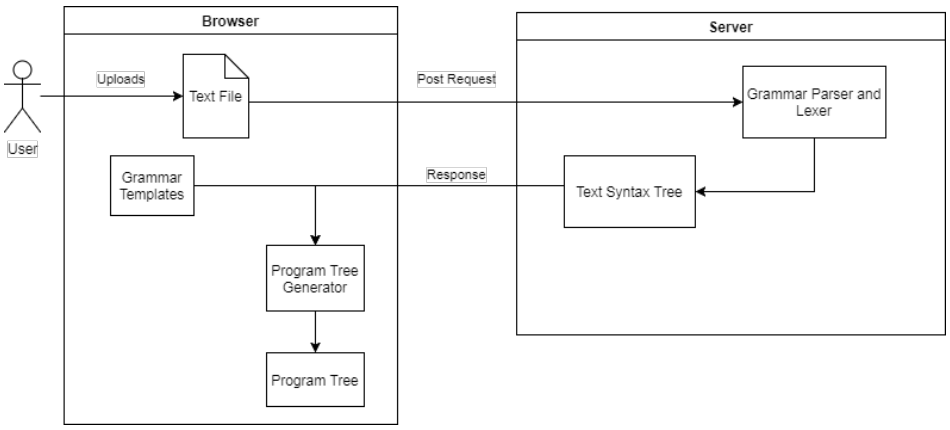


Figure 24: Loading a text file

5.2 WEBSITE

The web application resembles the Java application. The user can load the grammar and a text file and edits the program the same as the Java version. The web application can be accessed in https://sdegenerator.herokuapp.com/.

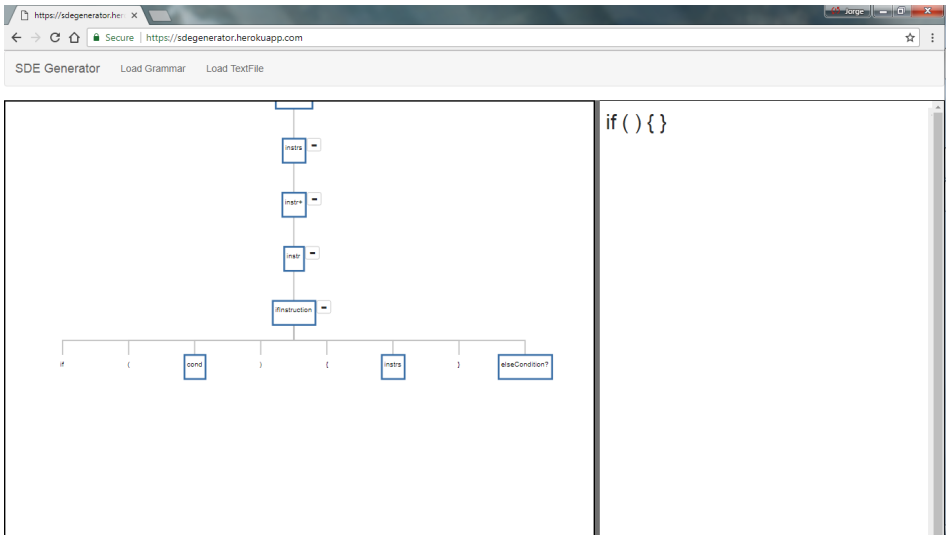


Figure 25: Web Application

6

SDE GENERATOR: USE OF THE SYSTEM

In this chapter it will be presented the environment where the user edits the program. It will be listed the commands available to the user for the edition of the program. It will also be explained how the *comment* and *helper* annotations influence the edition of the program.

6.1 SIMPLE EDITING

To edit the program the user just has to left click in the tree node and the respective production terms will appear in the level below. This type of editing is only possible if the rule has only one production.

```
whileInstruction : 'while' '(' cond ')' '{' @nlt instrs '}' @nl
                 ;
```

Listing 6.1: While rule

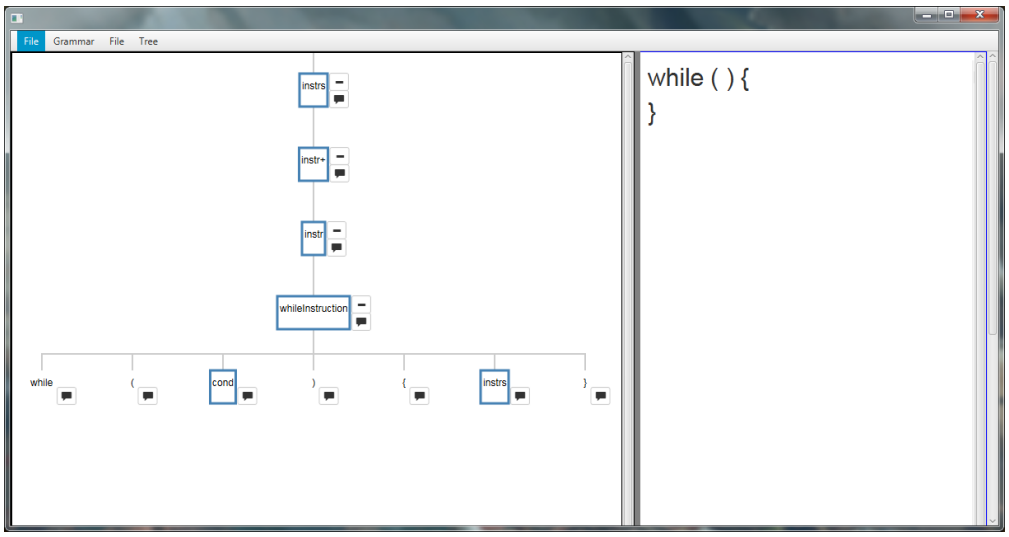


Figure 26: While condition in the editor

6.1.1 *Multiple Productions*

If the rule has more than one production the user instead of left clicking has to right click the node. After right clicking a list of available productions will be presented then the user chooses the desired option and keeps on building. Figure 27 shows the resulting list of the rule defined in listing 6.2.

```
instr : ifInstruction
      | whileInstruction
      | forInstruction
      | atr ';'
      ;
```

Listing 6.2: Inst rule

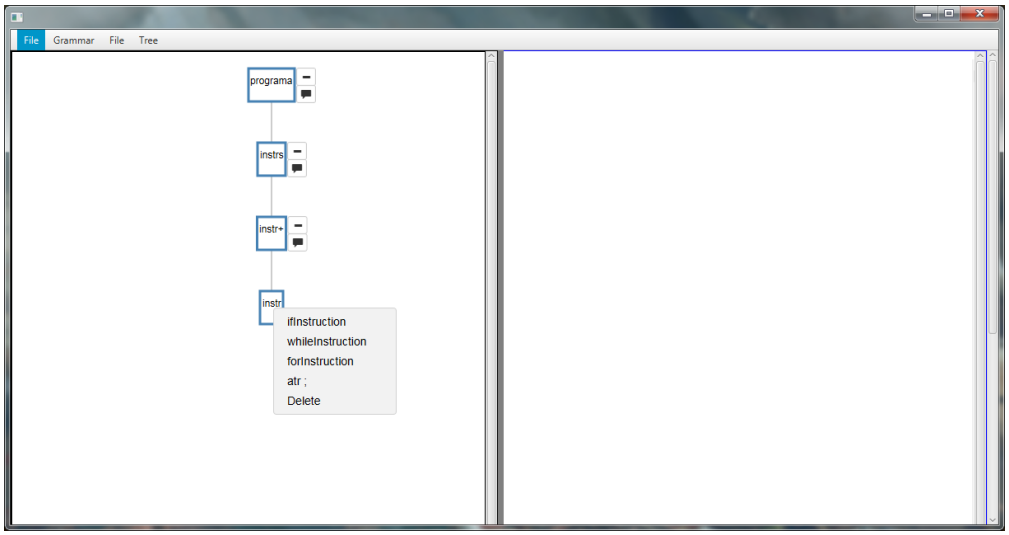


Figure 27: Multiple Productions in the editor

6.1.2 *Quantifiers*

Productions terms may be followed by quantifiers. There are three types of quantifiers, the '**+** ' quantifier (one or more occurrences), the '***** ' quantifier (zero or more occurrences) and the '**?** ' quantifier (zero or one occurrence).

'+' Quantifier

To add more than one occurrence to a term the user simply has to click the tree node has many times as he wants.

'' Quantifier*

Just as the '+ ' quantifier the user clicks as many times as he wants. If the user pretends zero occurrences right clicking the node will show an option *Empty* that adds an empty node to the tree.

'?' Quantifier

The user either left clicks the node or right clicks and chooses the *Empty* option.

6.1.3 *Block Editing*

Block editing is done the same way as editing rules with multiple productions where the right click list is composed by the block alternatives.

6.1.4 *Editing Terminals*

To edit the terminal terms the user clicks the pencil button next to the node and writes the terminal node. One of the features of this editor is the fact that is also possible to edit the terminal node in the text.
If the text inputted isn't valid the editor alerts the user marking the text red as show in Figure 29.

6.1.5 *Comments*

It is possible to assign comments to the tree nodes. To do that the user clicks in the comment button next to the tree node and writes the desired comment. If the *@bc* is defined, the comment will be written on the left side of the editor. Like editing terminals it is possible to edit the comments in both sides of the editor.

6.1.6 *Helper Annotations*

When the user defines the helper annotation to a term an exclamation point appears next to the correspondent tree node. When clicked displays the annotation. Unlike comments, it's not possible to edit the annotation.

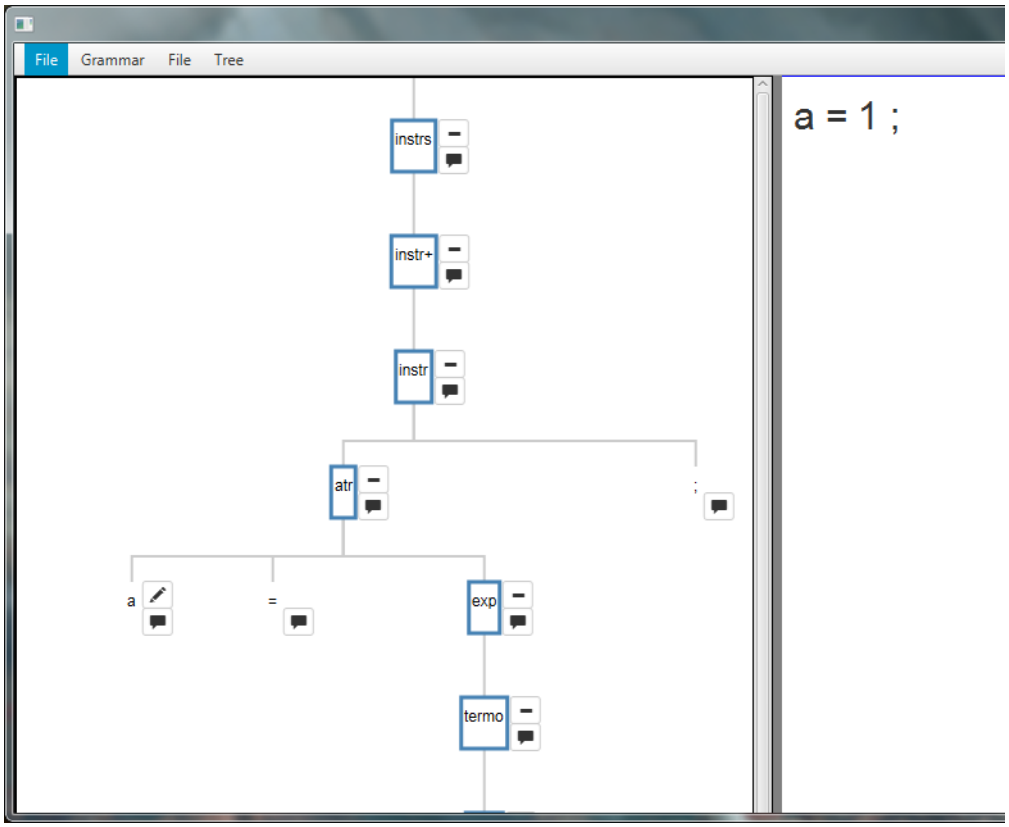


Figure 28: Valid terminal

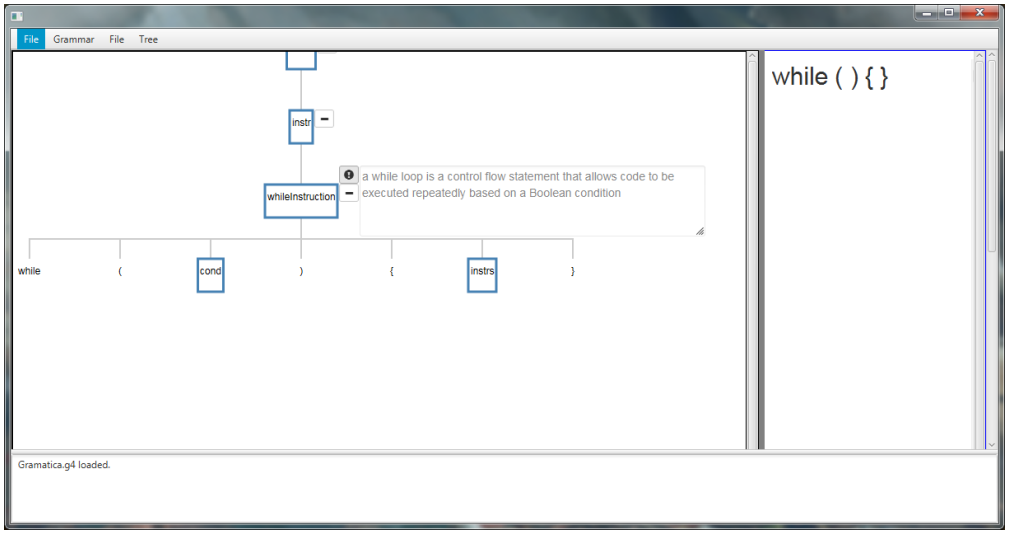


Figure 30: Helper Annotation

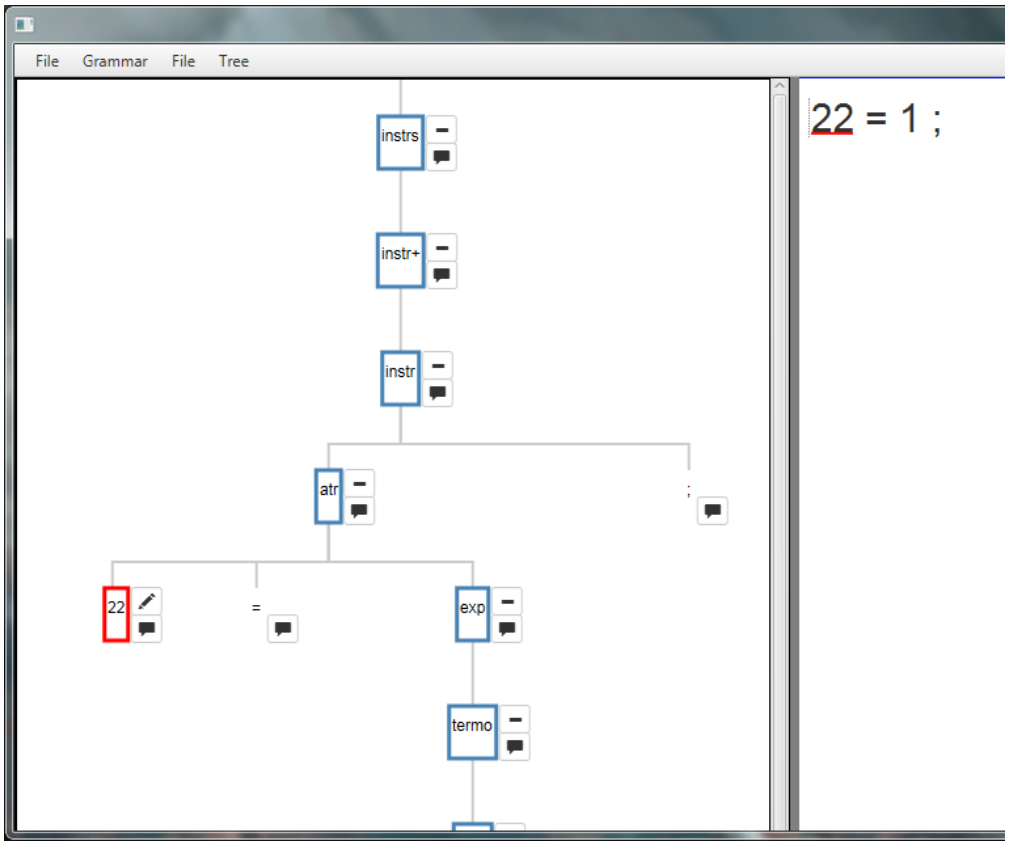


Figure 29: Invalid terminal

6.1.7 *Swap Operations*

It is possible to swap the position of the nodes that are children to the same node with the '+' or '*' quantifier. This allows the user to switch the order of the program.

6.2 LOADING AND SAVING

There are two ways to save and load programs. The user can choose between saving the editing tree or saving as a text file.

6.2.1 *Tree Saving and Loading*

This option saves the tree state as a serialized JAVA object. The grammar templates, the parser and the lexer are also saved so the serializable object can be loaded without requiring the grammar to be loaded. This option allows the program to be saved without being a valid program.

6.2.2 *Text Saving and Loading*

The editor only allows valid text files to be loaded unto the editor. To save to a text file the editor firstly validates the text and if it is valid saves the program.

7

CONCLUSION

By being aware of the language grammar, syntax-directed editing is a mechanism useful for a beginner user as it ensures the syntactic correctness of a program. Editors that allow this kind of editing are import as it spares a new user from knowing extensively the programming language grammar.

In this master's thesis it was developed a system capable of generating syntax-directed editors for a given language grammar. The grammar is written in a meta-language that allows the user to annotate the grammar the change the display of the program on the editor. The generated editor is intuitive and easy to use to a user new to the programming language. The editor is capable of loading text files of a determined grammar previously loaded unto the editor as well as saving the program as a text file if the program is syntactically correct. If the program is not correct it's possible to save the state of the edition to resume at another time.

Along this document it was documented the development of the SDE generator, from the definition of the meta language, used by the system to generate the SDE, to the generation of its templates and finally how these templates were used to allow the edition of the program. It was also described how the system is used.

The SDE generator is available both as a standalone Java application and as a web application.

BIBLIOGRAPHY

Metaedit+ workbench evaluation tutorial. https://www.metacase.com/support/45/manuals/evaltut/et-Title.html, a. Accessed: 2017-10-15.

Metaedit+ workbench user's guide. https://www.metacase.com/support/45/manuals/mwb/Mw.html, b. Accessed: 2017-10-15.

Farah Arefi, Charles E Hughes, and David A Workman. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3):349–360, 1990.

Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006.

Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

Martin Fowler. Projectional editing. http://martinfowler.com/bliki/ProjectionalEditing.html. Accessed: 2016-11-12.

Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.

Pedro Rangel Henriques, Maria Joao Varanda Pereira, Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Automatic generation of language-based tools. *Electronic notes in theoretical computer science*, 65(3):77–96, 2002.

Pedro Rangel Henriques, MJ Varanda Pereira, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using the lisa system. *IEE Proceedings-Software*, 152(2):54–69, 2005.

TF Lunney and RH Perrott. Syntax-directed editing. *Software Engineering Journal*, 3(2):37–46, 1988.

Marjan Mernik, Mitja Lenic, Enis Avdicauševic, and Viljem Zumer. Multiple attribute grammar inheritance. *Informatica*, 24(3):319–328, 2000.

Sten Minör. Interacting with structure-oriented editors. *International Journal of Man-Machine Studies*, 37(4):399–418, 1992.

Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains mps as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 165–168. ACM, 2013.

Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 169–176. ACM, 1982.

Thomas Reps and Tim Teitelbaum. The synthesizer generator. *ACM SIGSOFT Software Engineering Notes*, 9(3):42–48, 1984.

Thomas Reps and Tim Teitelbaum. *The synthesizer generator reference manual*. Springer Science & Business Media, 2012.

Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93. ACM, 2003.

Juha-Pekka Tolvanen, Risto Pohjonen, and Steven Kelly. Advanced tooling for domain-specific modeling: Metaedit+. In *Sprinkle, J., Gray, J., Rossi, M., Tolvanen, JP (eds.) The 7th OOPSLA Workshop on Domain-Specific Modeling, Finland*, 2007.

Mark GJ van den Brand, Arie van Deursen, Jan Heering, HA De Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A Olivier, Jeroen Scheerder, et al. The asf+ sdf meta-environment: A component-based language development environment. In *International Conference on Compiler Construction*, pages 365–370. Springer, 2001.

Markus Voelter and Vaclav Pech. Language modularity with the mps language workbench. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1449–1450. IEEE, 2012.

Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–304. ACM, 2010.

Marvin V Zelkowits. A small contribution to editing with a syntax directed editor. *ACM SIGSOFT Software Engineering Notes*, 9(3):1–6, 1984.