Universidade do Minho

Departamento de Informática

Informatics Dictionary

Pedro Mendes (a79003)

July 8, 2019

Abstract

Sometimes the languages at our disposal are not the most appropriate means of solving the problems we face, in these situations a *Domain Specific Language* is at times the best solution.

Contents

Chapter 1

Introduction

This project aims to create a simple and intuitive language to describe a dictionary and later annotate texts with.

First we'll analyse the problem, see what needs to be implemented and what challenges need to be overcome to implement said features. Then there will be an overview of the technologies used and finally the implementation done. The 2 main tools that were be used for this job were *yacc* and *flex* which operate on *Domain Specific Languages* them selves.

Chapter 2

Problem

The problem this program intends to solve is the following, the informatics department wants to create a dictionary of commonly used words, associating with each of them a *meaning*, the *English name* and a list of *synonyms*. This dictionary is read in conjunction with texts (that may or may not contain a title) and annotates them with footnotes explaining the words that are defined in the dictionary.

To solve this problem, a DSL[1] needs to be defined where the dictionary can be stored as well as texts to annotate. It's also important that the language is user friendly as it is intended to be human readable and writeable.

Another problem that needs to be taken into account is UTF-8 encoding. Finding words in a text means separating words by spaces but also take into account punctuation and other non-alphanumeric characters.

[1] Domain Specific Language

Chapter 3

Solution

3.1 Definition of the SATI language

Similarly to how an imperative program is split in two, declarations and instructions, this language is split between dictionary (*Dicl*) and texts (*Texts*), separated by one or more ‘\%’.

⟨*sati*⟩ → ⟨*dicl*⟩ ’%’ ⟨*texts*⟩
  | ’%’ ⟨*texts*⟩
  | ⟨*dicl*⟩ ’%’

3.1.1 Dictionary

The dictionary is a collection of words and each word is composed of 4 parts.

⟨*dicl*⟩ → ⟨*word*⟩
  | ⟨*word*⟩ ⟨*dicl*⟩

The word to find in the dictionary *WD*, it’s meaning *Meaning* the *English Name* and finally either a list of synonyms or a single synonym.

The list of synonyms contains synonyms separated by `','` and the last one on the list may or may not be followed by a comma.

⟨*word*⟩ → ⟨*wd*⟩ ‘:’ ⟨*meaning*⟩ ‘|’ ⟨*englishName*⟩ ‘|’ ‘[’ ⟨*synonyms*⟩ ‘]’ ‘;’
  | ⟨*meaning*⟩ ‘|’ ⟨*englishName*⟩ ‘|’ ⟨*synonym*⟩ ‘;’

⟨*synonyms*⟩ → ⟨*synonym*⟩ ‘,’
  | ⟨*synonym*⟩
  | ⟨*synonym*⟩ ‘,’ ⟨*synonyms*⟩

⟨*wd*⟩ → ⟨*ID*⟩

⟨*meaning*⟩ → ⟨*ID*⟩

⟨*englishName*⟩ → ⟨*ID*⟩

⟨*synonym*⟩ → ⟨*ID*⟩

As an example, a dictionary can be written like this:

```
Encapsulamento : Um mecanismo da linguagem para
                 restringir o acesso aos componentes
                 de um objecto.
```

```
 | Encapsulation | Modularidade
 ;

 Imutabilidade : Uma propriedade de informação que
  implica que esta não pode ser alterada.
| Imutability
| [ Constante, Inalteravel, ]
;
```

3.2 Texts

The texts section is composed of texts that may or may not have a title, the title is used to name the LaTeXchapter, if no title is given then *Untitled X* will be used where *X* is the number of untitled texts parsed so far.

The texts are surrounded by double quotes " and their title is text preceding the text. The language specification for presenting the texts is the following:

⟨*texts*⟩ → '"' ⟨*text*⟩ '"'
| ⟨*texts*⟩ '"' ⟨*text*⟩ '"'
| ⟨*texts*⟩ ⟨*title*⟩ '"' ⟨*text*⟩ '"'
| ⟨*title*⟩ '"' ⟨*text*⟩ '"'

⟨*text*⟩ → ⟨*TEXT*⟩

⟨*title*⟩ → ⟨*TEXT*⟩

And a sample text section could be:

```
POO "O encapsulamento permite uma maior modularidade e organização do código."
"Em programação é muito importante o single responsability principle."
```

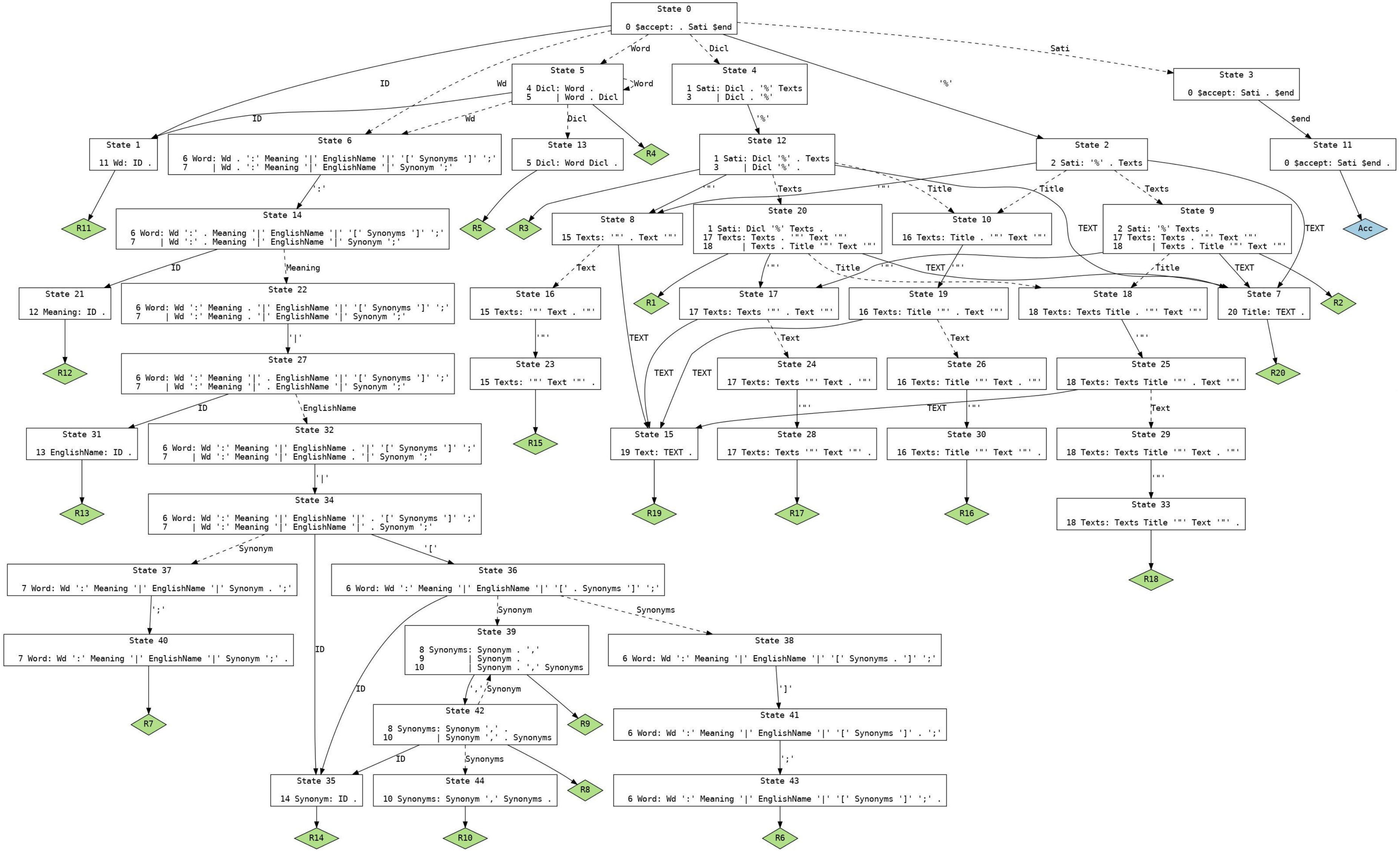
State 0
0 $accept: . Sati $end
Word
Dicl
Sati
ID
Wd
'%'
State 5
4 Dicl: Word .
5 | Word . Dicl
Word
State 4
1 Sati: Dicl . '%' Texts
3 | Dicl . '%'
State 3
0 $accept: Sati . $end
ID
Wd
Dicl
'%'
$end
State 1
11 Wd: ID .
State 6
6 Word: Wd . ':' Meaning '|' EnglishName '|' '[' Synonyms ']' ';'
7 | Wd . ':' Meaning '|' EnglishName '|' Synonym ';'
State 13
5 Dicl: Word Dicl .
R4
State 12
1 Sati: Dicl '%' . Texts
3 | Dicl '%' .
State 2
2 Sati: '%' . Texts
State 11
0 $accept: Sati $end .
':'
'"'
Texts
Title
'"'
Title
Texts
R11
State 14
6 Word: Wd ':' . Meaning '|' EnglishName '|' '[' Synonyms ']' ';'
7 | Wd ':' . Meaning '|' EnglishName '|' Synonym ';'
R5
R3
State 8
15 Texts: '"' . Text '"'
State 20
1 Sati: Dicl '%' Texts .
17 Texts: Texts . '"' Text '"'
18 | Texts . Title '"' Text '"'
State 10
16 Texts: Title . '"' Text '"'
TEXT
State 9
2 Sati: '%' Texts .
17 Texts: Texts . '"' Text '"'
18 | Texts . Title '"' Text '"'
TEXT
Acc
ID
Meaning
Text
'"'
Title
'"'
TEXT
'"'
Title
TEXT
State 21
12 Meaning: ID .
State 22
6 Word: Wd ':' Meaning . '|' EnglishName '|' '[' Synonyms ']' ';'
7 | Wd ':' Meaning . '|' EnglishName '|' Synonym ';'
State 16
15 Texts: '"' Text . '"'
R1
State 17
17 Texts: Texts '"' . Text '"'
State 19
16 Texts: Title '"' . Text '"'
State 18
18 Texts: Texts Title . '"' Text '"'
State 7
20 Title: TEXT .
R2
'|'
'"'
TEXT
Text
Text
'"'
R12
State 27
6 Word: Wd ':' Meaning '|' . EnglishName '|' '[' Synonyms ']' ';'
7 | Wd ':' Meaning '|' . EnglishName '|' Synonym ';'
State 23
15 Texts: '"' Text '"' .
TEXT
TEXT
State 24
17 Texts: Texts '"' Text . '"'
State 26
16 Texts: Title '"' Text . '"'
State 25
18 Texts: Texts Title '"' . Text '"'
R20
ID
EnglishName
'"'
'"'
TEXT
Text
State 31
13 EnglishName: ID .
State 32
6 Word: Wd ':' Meaning '|' EnglishName . '|' '[' Synonyms ']' ';'
7 | Wd ':' Meaning '|' EnglishName . '|' Synonym ';'
R15
State 15
19 Text: TEXT .
State 28
17 Texts: Texts '"' Text '"' .
State 30
16 Texts: Title '"' Text '"' .
State 29
18 Texts: Texts Title '"' Text . '"'
'|'
'"'
R13
State 34
6 Word: Wd ':' Meaning '|' EnglishName '|' . '[' Synonyms ']' ';'
7 | Wd ':' Meaning '|' EnglishName '|' . Synonym ';'
R19
R17
R16
State 33
18 Texts: Texts Title '"' Text '"' .
Synonym
'['
R18
State 37
7 Word: Wd ':' Meaning '|' EnglishName '|' Synonym . ';'
State 36
6 Word: Wd ':' Meaning '|' EnglishName '|' '[' . Synonyms ']' ';'
';'
Synonym
Synonyms
ID
State 39
8 Synonyms: Synonym . ','
9 | Synonym .
10 | Synonym . ',' Synonyms
State 40
7 Word: Wd ':' Meaning '|' EnglishName '|' Synonym ';' .
State 38
6 Word: Wd ':' Meaning '|' EnglishName '|' '[' Synonyms . ']' ';'
ID
','
Synonym
']'
R7
State 42
8 Synonyms: Synonym ',' .
10 | Synonym ',' . Synonyms
R9
State 41
6 Word: Wd ':' Meaning '|' EnglishName '|' '[' Synonyms ']' . ';'
ID
Synonyms
';'
State 35
14 Synonym: ID .
State 44
10 Synonyms: Synonym ',' Synonyms .
R8
State 43
6 Word: Wd ':' Meaning '|' EnglishName '|' '[' Synonyms ']' ';' .
R14
R10
R6

Figure 3.1: The Parser's Automata generated by *yacc*

And the full SATI file can be something like this.

```
Encapsulamento : Um mecanismo da linguagem para
                 restringir o acesso aos componentes
                 de um objecto.
               | Encapsulation
               | Modularidade
               ;

               Imutabilidade : Uma propriedade de informação que
                implica que esta não pode ser alterada.
              | Imutability
              | [ Constante, Inalteravel, ]
              ;
              %%
              POO "O encapsulamento permite uma maior modularidade e organização do código."
              "Em programação é muito importante o single responsability principle."
```

3.2.1 Lexing

To obtain all the literals and terminal tokens (ID and TEXT) a lexer was written in *flex*. Here the same two zones (*Dicl* and *Texts*) were used, where *Dicl* is the INITIAL state and *Texts* is TEXTS state. When the sequence of '%' is found the lexer switches state and returns the '%'. In this manner we can see what characters can not be part of each of the terminal symbols. For the ID we see that the symbols % ; , " [ ] : | cannot be used. And for TEXT '"' cannot be used, the latter can be fixed by replacing all '"' with '``', as LaTeXinterprets these as double quotes.

```
%option yylineno
%x TEXTS
ID_SEP [^\n %;,"\[\]:|]
TEXT_SEP [^\n "]
%%
{ID_SEP}[^%;,"\[\]:|]*{ID_SEP}   { yylval.str = strdup(yytext); return ID; }
(%)+                             { BEGIN TEXTS; return yytext[0]; }
<TEXTS>{TEXT_SEP}[^"]*{TEXT_SEP} { yylval.str = strdup(yytext); return TEXT; }
<*>[%;,"\[\]:|]                  { return yytext[0]; }
<*>.|\n                          { ; }
%%
```

3.3 Architecture

The architecture is split in two parts, the parsing of the dictionary and the parsing mutation of the input texts.

To achieve this, the following structures were designed: A word struct that represents each word in the dictionary, and the main Sati struct that stores the dictionary as well as some extra information.

The Word struct is pretty self-explanatory. It contains the same fields as the ones described by the language. The word itself, its meaning, the English name, and a list of synonyms.

The Sati struct is a bit more complex. First and foremost it has a dictionary that stores the Words associated with the word (String). Then it has the current_word field, (this will be explained more in depth in SubSection 3.3.1), but it keeps track of the word that is being currently parsed.

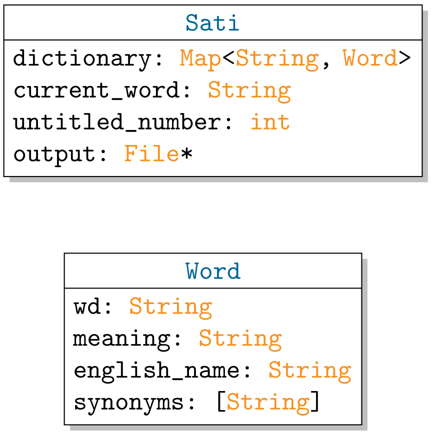

Figure 3.2: Structures

Next comes the untitled_number, this serves to count the number of untitled posts in order to produce the behaviour described in Section 3.2. And finally, the output field, which stores the file where to output the content, this is the standard output, by default, but can be changed with the flags documented in Chapter 4.

3.3.1 Parsing the dictionary

To parse the dictionary 2 elements of the Sait struct are used, the dictionary and the current_word, when a new word is added it's added to the dictionary and the key is stored in current_word, then, when a meaning, English name or synonym is added the current_word is used to access the dictionary and update corresponding field.

In Figure 3.3 current_word is abbreviated to cw and dictionary to dict for brevity.

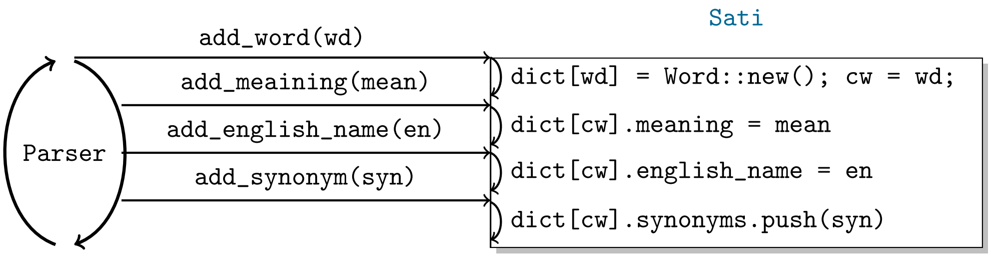

Figure 3.3: Dictionary building flowchart

3.3.2 Parsing and annotating the texts

After the dictionary is parsed, the texts start being produced. This procedure is relatively simple, a text is sent along with it's title to the Sati module and every occurrence of a word is changed to include a footnote with the information in the dictionary.

The outcome of this function is the production of a LaTeXchapter titled either "*Untitled X*" (where *X* is the untitled_number) or the passed title.

To find the words that needed to be annotated the text was split on spaces, this came with another problem, sometimes words are followed by punctuation instead of spaces and thus the 'dictionary' can't find it. For example, "word!" is not in the dictionary but "word" is, to solve this I made use of *Rust*'s String library, which allows, amongst other things, the trimming the start and end of a string of non alphanumeric characters in a UTF-8 aware way[1].

And finally, the texts are flushed to the file after being parsed, as to not allocate unnecessary memory.

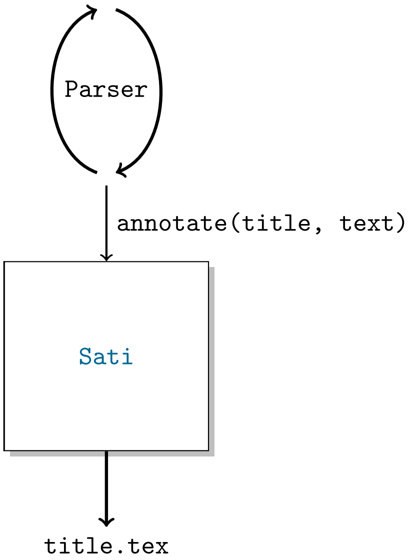

Figure 3.4: Annotating a text using the -s flag

[1] Rust Docs

Chapter 4

Usability

4.1 Flags

The program has a few flags documented in it's man page. These allow the user to control the input and output of the program.

- `-o | --output` *`file`*: Redirects the output the file specified as parameter.
- `-i | --input` *`file`*: Redirects input to the file specified as parameter.
- `-s | --split`: Outputs each text in a separate file, with the same name as the chapter title.
- `-n | --no-header`: Suppresses the output of the LaTeX headers needed for a compiling document.
- `-h | --help`: Shows the commands usage.

A concise manual page was also written to help document the program, it can be seen through the `man sati` command after installing the program. Which is also handled by the makefile written (`make install`). Because this project requires the `Rust` compiler to build a platform method of installation was also provided in the makefile (`make rust`).

4.2 Errors

The library currently reports all error messages with the line in which they occurred, and aborts execution once any error is found.

Currently the errors that can be reported on are:

- Input/Output errors, when a write or read fails or when a file can't be opened.
- Syntax errors detected by the grammar.
- Redefinition of a word in the dictionary.
- Adding a second meaning to a word.
- Adding a second description to a word.

The last two errors never happen due to the nature of the language, they will be syntax errors before being logic errors, but the library code developed is still prepared to deal with these.

Chapter 5

Conclusion

In conclusion, *yacc* is a very powerful tool for writing and maintaining context independent grammar, clearly separating and performing the parsing in a self-contained and defined place leaving other language logic to be distributed into other models. In conjunction with *flex*, which provides a means to lex the different language tokens, the process of creating a DSL was completely painless.

The resulting software is able to parse the language and produce a usable latex document that can be easily integrated in other documents.

Appendix A

GIC

```
%union { char* str; }
%token <str> ID
%token <str> TEXT
%type <str> Wd Synonym Meaning EnglishName Dicl Word Synonyms Text Title
%%
Sati : Dicl '%' Texts
     | '%' Texts
     | Dicl '%'
     ;

Dicl : Word
     | Word Dicl
     ;

Word : Wd ':' Meaning '|' EnglishName '|' '[' Synonyms ']' ';'
     | Wd ':' Meaning '|' EnglishName '|' Synonym ';'
     ;

Synonyms : Synonym ','
         | Synonym
         | Synonym ',' Synonyms
         ;

Wd : ID                              { add_word($1); }
   ;

Meaning : ID                         { add_meaning($1); }
        ;

EnglishName : ID                     { add_english_name($1); }
            ;

Synonym : ID                         { add_synonym($1); }
        ;


Texts : '"' Text '"'                 { annotate($2); }
      | Title '"' Text '"'           { annotate_with_title($1, $3); }
      | Texts '"' Text '"'           { annotate($3); }
      | Texts Title '"' Text '"'     { annotate_with_title($2, $4); }
      ;

Text : TEXT                          { $$=$1; }
     ;

Title : TEXT                         { $$=$1; }
      ;
%%
```

Appendix B

Flex

```
%option yylineno
%x TEXTS
ID_SEP [^\n %;,"\[\]:|]
TEXT_SEP [^\n "]
%%
{ID_SEP}[^%;,"\[\]:|]*{ID_SEP}   { yylval.str = strdup(yytext); return ID; }
(%)+                             { BEGIN TEXTS; return yytext[0]; }
<TEXTS>{TEXT_SEP}[^"]*{TEXT_SEP} { yylval.str = strdup(yytext); return TEXT; }
<*>[%;,"\[\]:|]                  { return yytext[0]; }
<*>.|\n                          { ; }
```