CS 3402 UTEC - Digits in German

Nicho Galagarza, Jorge
de Lama Zegarra, Jose
Tenazoa Ramírez, Renzo

December, 2020

Abstract

A C++ program for parsing the German cardinal numbers grammar is implemented in the present project. We develop a top-down parser which performs rightmost derivation for each input string and provides error messages signaling the corresponding unrecognized and misplaced tokens. We conclude that the grammar of the cardinal numbers in German can be expressed as a Context-Free Grammar without ambiguity and therefore any of its strings can successfully be subjected to the related parsing task.

Keywords: Parser, Context-Free Grammar, Chomsky's Normal Form

1 Introduction

In the present project, we implement a parser for the Context-Free Grammar (CFG) that describes the syntax for the German cardinal numbers. In order to do so, we apply a rightmost derivation of the grammar i.e. on a given input, we constantly replace the rightmost token that we can identify on the input string. This is possible when the CFG at hand is non-ambiguous.

1.1 Context-Free Grammars

Most commonly abbreviated "CFG" (as it is going to be done in the present project), a Context-Free Grammar is a means for describing a certain class of languages that can be defined recursively. For example, the language $L = \{n \in \mathbb{N} : 0^n 1^n\}$ is associated to the following CFG:

$$S \to 01|\epsilon$$
$$S \to 0S1$$

In a more formal manner, there are four important components in the definition of a CFG [1]:

- A finite set of symbols.
- A finite set of variables.
- A start symbol, which is the variable upon which we can start executing our derivation.
- A finite set of productions or rules associated to a set of symbols.

1.2 Chomsky Normal Form

A grammar is said to be in Chomsky Normal Form (abbreviated "CNF") when all of it's production rules are of the form:

$$A \to BC|\epsilon, \text{ or}$$
$$A \to \alpha, \text{ or}$$
$$S \to \epsilon$$

Where α is a terminal symbol.

Transforming the grammar production rules to fit this form is beneficial for simplifying the labour of the parser, as it only needs to take (in the worst case, numerous) binary decisions to identify the correspondent tokens.

1.3 Parsers

A parser, or syntactic analyzer, is a device that performs parsing, which is, in a very basic sense, determining if a string conforms to the rules of a given formal grammar. It is the second stage of a compiler's front end. Therefore, each word should already be associated to a syntactic category, for the compiler to associate the corresponding structure for the program [2].

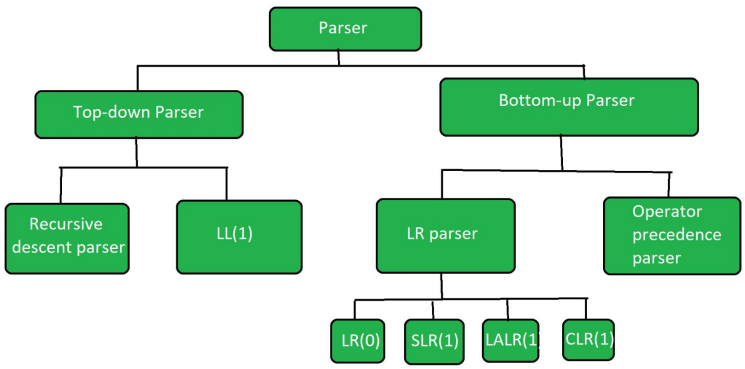

Figure 1: Categories of parsers

Furthermore, parsers can be subdivided into two main categories: top-down and bottom-up parsers, whereby we can subdivide each and find the most popular for each: LL(1) and LR parsers respectively. These are 2 popular approaches (both being left-to-right parsers, referencing the direction of scanning) which, aside from the categories they each belong, primarily distinguish from each other by the order of derivation for the input string: LL parsers perform leftmost derivations while LR parsers do so with rightmost ones [2].

It is worth mentioning that LL and LR parsers can't derive all the CFG's, which is why the languages that can be recognized are called LL grammars and LR grammars. The former are of great practical interest due to their ease to construct, which is why many programming languages are designed to be LL(1). However, LR parsers are far more powerful as they incorporate a larger set of languages.

2 Methodology

A parser was implemented in C++ language. It was determined that this would be a top-down right-to-left parser, which performs a rightmost derivation on the input string, in order to test a different approach from the already known classic parsers. After built, some tests were ran to verify its correctness, which are shown in the Results section.

2.1 Code explanation

In order to start describing the way it was implemented so it is clear, we can start by introducing the main.cpp file:

```
1   #include "parser.h"
2
3   using namespace std;
4
5   int main() {
6       string numero;
7       cout << "Insertar numero en aleman: "; cin >>  numero;
8       if(parser(numero)){
9           cout << "¡Numero Aceptado!" << endl;
10      } else {
11          error(numero);
12      }
13      return 0;
14  }
```

The main file receives the input string for the CFG and calls the parser boolean function on it, which will return true if the input belongs to the grammar and false otherwise, with the corresponding call to the error function (called in line 11 of the figure) in order to identify what token was unidentified or misplaced so that the error occurred.

Now, into the parser.h file, as part of the setup, we describe the working of the utility

function parser:

```
bool scanner (string& numero,string token) {
    int cont = 0;
    int s = (int)numero.size()-1;
    for(int j = (int)token.size()-1; j >= 0; j--){
        if(numero[s] == token[j]){
            cont++;
        }
        s--;
    }

    if(cont == token.size()){
        for(int i = token.size(); i > 0;i--){
            numero.pop_back();
        }
        return true;
    } else {
        return false;
    }
}
```

The basic functionality of this function is altering the string *numero* it receives as its parameter if and only if the string *token* can be recognized as its suffix. If that is the case, then that suffix is deleted from numero.

We proceed to describe the functioning of parser, the key function for the whole procedure:

```
bool parser(string& numero) {
return  Z2(numero) ||
        Z7(numero) ||
        Z8(numero) ||
        Z9(numero) ||
        Z10(numero) ||
        Z11(numero) ||
        Z12(numero) ||
        Z13(numero) ||
        Z14(numero) ||
        U(numero);
}
```

Here, parser represents the initial state, which will try to derive from the CFG, with every possible belonging state, the string at hand. Each "Z-function" is a boolean utility function that represents the corresponding production rule with the associated state. Any will return true if it found a rightmost derivation for the string, and false otherwise.

For clarity purposes, let us illustrate the usage of two distinct production rules: Z11 and Z1. The former representing a non-terminal state and the latter a terminal state. Let us also assume we received the string "ein" on input and parser function just called Z11 along all the other functions.

```
    bool Z1 (string& numero){
        if (scanner(numero,"ein") || scanner(numero,"zwei")
        || scanner(numero,"drei") || scanner(numero,"vier")
        || scanner(numero,"fünf") || scanner(numero,"sechs")
        || scanner(numero,"sieben") || scanner(numero,"acht")
        || scanner(numero,"neun")){
            return true;
        } else {
            return false;
        }
    }
    bool Z11 (string& numero){
        if(Z1(numero) || Z2(numero) || Z7(numero) || Z8(numero)) {
            return Z5(numero) || Z9(numero);
        } else {
            return false;
        }
    }
```

Z11 will systematically verify if string "ein" can be given as input any of the production rules so that any of them returns true. If it starts by Z1, the latter will call the scanner function on the string and verify if it can fit a token on it. In one of its calls, scanner will be given the input string as an input alongside the token "ein". It is here where it will return true, hence Z1 will return true as well as Z11, hereby making the original parser function return true as well, concluding the program.

3 Results

In the next images, we can see the 3 tests input completed.

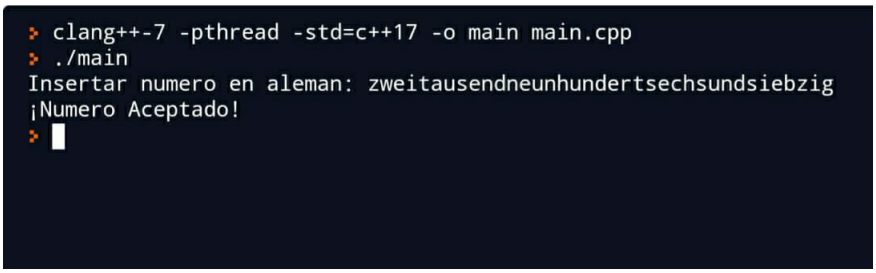

Figure 1: Test 1 successful - Done!

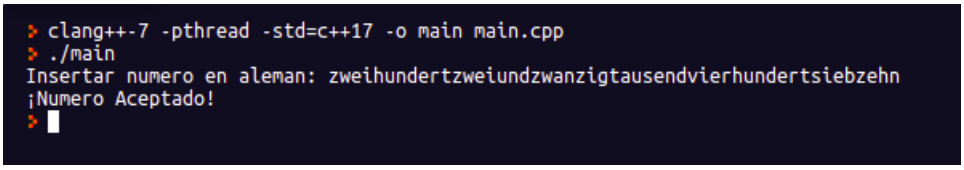

Figure 2: Test 2 successful - Done!

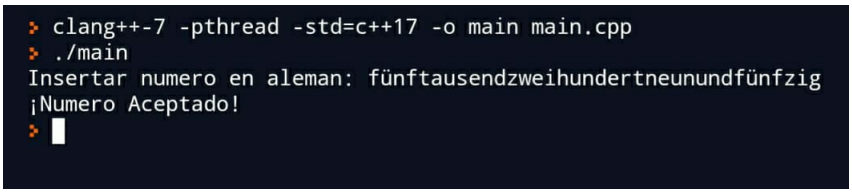

Figure 3: Test 3 successful - Done!

In the next images, we can our tests completed.

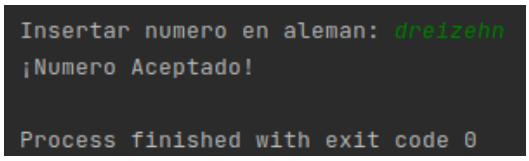

Figure 4: Test 4 succesful - Done!

In the following case, we write an incorrect entry. So, we print that the number was rejected, an exit(), and the token that failed.

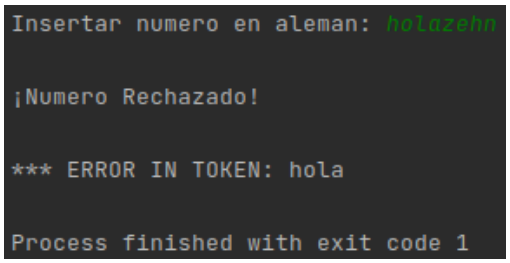

Figure 5: Test 5 failed - Done!

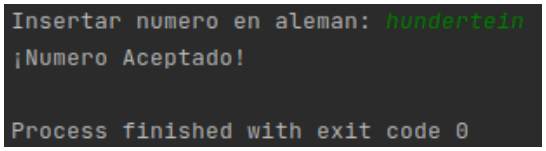

Figure 6: Test 6 successful - Done!

4 Conclusions

We can conclude that a non-standard (namely, top-down and right-to-left) parser can be implemented for recognizing the CFG of the German cardinal numbers. Further work could extend the functionality of this parser in order to make it able to perform arithmetic operations on numbers separated by symbols. It could also be modified to recognize efficiently a broader set of cardinal numbers in different languages.

References

[1] Hopcroft, J. E., Ullman, J. D. (2006). *Introduction to automata theory, languages, and computation*. Third Edition. Reading, Mass.: Addison-Wesley.

[2] Cooper, K., Torczon, L. (2012).*Engineering a compiler*. Second Edition. Reading, Mass.: Morgan Kaufmann