Parser Generators and Grammar Filter for Context Free Grammars in Scala

Panagiotis Karagiannis PKARAGIA@UCSC.EDU

Konstantinos Zampetakis KZAMPETA@UCSC.EDU

Bitbucket Repository: `https://apropos_@bitbucket.org/apropos_/projectpl.git`

1. Abstract

In this project we programmed grammar filters as well as a parser generator for context free grammars. More precisely, we implemented a parser generator which, given a grammar generates a function that is a parser. When this parser is given a program to parse, it produces a derivation for that program, or an error indication if the program contains a syntax error and cannot be parsed. We implemented generators for two kinds of parsers, namely, a top down parser and a CYK parser. Moreover, we implemented an interesting function that filters out harmless production rules from context free grammars. The implementation of the aforementioned programs was made in Scala, which is a functional language suited well for this kind of applications.

2. Introduction

We first begin with the definition of the Context-free grammar (CFG):

Definition 1 *A context-free grammar $\mathcal{G}$ is defined by the* 4*-tuple* $\mathcal{G} = (\mathcal{N}, \Sigma, P, S)$*, where:*

- *$\mathcal{N}$ is a finite set of non-terminal symbols.*
- *Σ is a finite set of terminal symbols.*
- *P is a finite set of production rules P contains (a, b) tuples such that $a \in \mathcal{N}$ and $b \in (\mathcal{N} \cup \Sigma)^*$*
- *$S \in \mathcal{N}$ is the start symbol in $\mathcal{N}$.*

Example of CFG

$\mathcal{N} = \{S, X, A, B\}$

$\Sigma = \{a, b\}$

$P = \{(S, AB), (S, XB), (X, AS),$
$(A, aA), (A, a), (B, b)\}$

$S = S$

A popular notation for context-free grammars is the Backus–Naur form (BNF). Converting the previous example to BNF we get the following:

Example in BNF form

$S \to AB|\, XB$

$X \to AS$

$A \to aA|\, a$

$B \to b$

Once a formal grammar is defined, one interesting question is if we can construct an algorithm recognizing every sentence belonging in this grammar. The certificate of the recognition will be the *abstract syntax tree* (AST), that is the exact derivation of this sentence from the grammar rules.

Definition 2 *A parser is a function $\mathcal{P}$ that takes as input a pair $< \mathcal{G}, s >$ where $\mathcal{G}$ is a formal grammar and s is a sentence and:*

- *Decides if s belongs to $\mathcal{G}$.*
- *If s belongs to $\mathcal{G}$ returns a derivation of s from $\mathcal{G}$.*

Parsing a general formal grammar can be a challenging task. However, if we restrict ourselves to context free grammars, we can construct efficient parsers as we will see in the following sections.

3. Harmless Production Rules

By definition a production rule is harmless if it does not contribute to any ambiguities in the grammar. A specific kind of a harmless production rule is a “blind alley” rule: namely, a rule for which it is impossible to derive a string of terminal symbols. For example, we have added a pair of blind alley rules in our original grammar:

$$S \to AB|\, XB$$
$$X \to AS$$
$$A \to aA|\, a$$
$$B \to b$$
$$\boxed{\begin{array}{l} Y \to aR \\ R \to bY \end{array}}$$

Clearly, by starting at either of the boxed rules it is impossible to produce a final string consisting only of terminal symbols. In general, blind alley rules do not affect the language or parse trees generated, therefore they only constitute "noise", that we would like to filter out of the context free grammar. Next, we present an algorithm that given the production rules of a context free grammar, P, filters out the blind alley rules and returns a new list of rules P'.

Algorithm 1 Solution to Filtering Blind Alley Rules

```
 1: function FilterBlindAlleys($P$)                ▷ $P$ - Rules
 2:     Define $P'$=empty
 3:     for $r \in P$ do
 4:         Add $r$ to $P'$ if the right hand side of $r$
 5:         consists only of terminal symbols
 6:     end for
 7:
 8:     while a rule is added to $P'$ do
 9:         for $r \in P$ do
10:             Add $r$ to $P'$ if the right hand side of $r$ exists
11:             as a combination of the left hand sides of
12:             the rules in $P'$
13:         end for
14:     end while
15:
16:     return $P'$
17: end function
```

At first, our algorithm performs one pass through the list of production rules P and adds to P' the rules whose right hand side consist only of terminal symbols. Next, the algorithm iterates through all the rules of P checking if there exist rules such that their left hand side can be found as a combination of the right hand sides of the rules in P'. If such a rule $r \in P$ is found, then clearly r is not a blind alley rule, since its right hand side can be expanded in such a way that includes only terminal symbols. If we added at least one rule to the list P' then we perform the last step again since we might find another rule which can be expanded to include only terminal symbols, otherwise, we stop the execution of the algorithm returning P'.

As far as the Scala implementation of this algorithm is concerned, a point that is worth mentioning is the elegant functional implementation of the while loop that spans lines $8-14$. More precisely, we first implemented a generic procedure that computes a fixed point of any function f:

```scala
def compute_fixed_point[A](x:A,
    f:A=>A):A= {
  if (x == f(x) ) {x}
  else {compute_fixed_point(f(x), f )}
}
```

Then, we implemented a function f to perform lines $9-12$ of the algorithm and used the `compute_fixed_point` to essentially compute the "fixed point" of f. The function f received as input the list of rules P', hence, if no new rule was added to P' then we would return P' (the fixed point), else we would recompute f on the new input $f(P')$.

4. Top Down Left to Right DFS Parser

As the name suggests, since the parser is top down, it starts searching for a derivation from the start symbol and it stops either when all rules have been examined or when a valid parse tree is found. The high level idea of how our parser works is summarized in **Algorithm 2**.

Algorithm 2 High Level Idea of Top Down Parsing

```
 1: function TopDown
 2:         ▷ Where rhs- right hand side, lhs - left hand side
 3:
 4:     Start with the rules of a given symbol,
 5:     say $X$ (first this is the start symbol)
 6:
 7:     If the rules for $X$ are empty and all of the
 8:     elements of the fragment have been examined
 9:     return An acceptable derivation if it exists,
10:     otherwise
11:     return None
12:
13:     Extract the leftmost grammar rule for $X$
14:     for each symbol of the rhs of the rule do
15:         If symbol is terminal and matches the header of
16:         the fragment move to the next token of the
17:         fragment and examine the remaining rules of
18:         symbol
19:
20:         If symbol is terminal and does not match the
21:         header of fragment return None
22:
23:         If symbol is non-terminal then goto line 2 and
24:         set $X$ to be the current symbol
25:     end for
26:
27:     return $P'$
28: end function
```

As we can see from our algorithm, for any given symbol the parser will fist extract the corresponding rules from the grammar. Then, in a left to right order, it will try to expand each rule, ultimately, performing a DFS of the grammar rules. If parsing the fragment is possible then the parser returns an AST for the fragment, otherwise it returns `None`. The fact that the parser is left to right has the drawback that it cannot handle left recursion of the form $B \to Ba$ because it will infinitely loop trying to expand symbol B. Similarly, indirect left recursion of the form $B \to Ta$, $T \to Ba$ will cause the same problem. Moreover, in case of grammar ambiguities our parser implementation simply returns the first acceptable derivation of the fragment. Finally, this parser performs an exhaustive search up until the first acceptable derivation is found, therefore, it is considered very inefficient. Parsers such as the LL and the Early Parser, use more sophisticated techniques, which enable them to reduce dramatically the running time of top down parsing (Altshuler).

Figure 1 depicts an example derivation for the fragment $aaabbb$. Since this fragment belongs in our language we see that our parser successfully finds a correct derivation. It is worth noting, that our parser had to discard many partial solution before deriving the AST for the fragment.

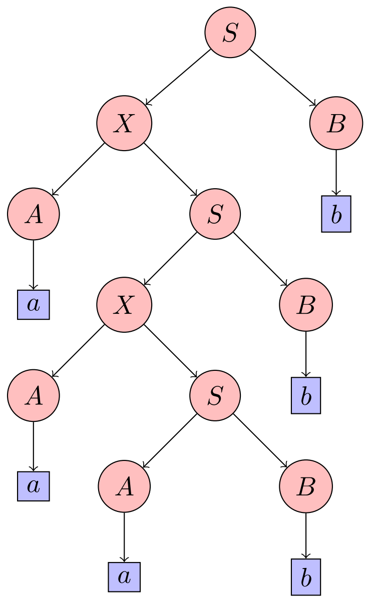

Figure 1. AST for the fragment $aaabbb$

The most challenging part of implementing the generator for the top down parser was that we had to use mutually recursive functions which prevented us from testing the code until it was completely finished. Nevertheless, Scala's expressiveness allowed us to write this parser relatively fast using about 100 lines of code.

5. CYK Parser

The Cocke–Younger–Kasami (CYK) algorithm is a bottom-up dynamic programming parser for context free grammars. The standard version of CYK operates only on context-free grammars given in Chomsky normal form (CNF), however since every context free grammar can be converted efficiently to CNF form, the algorithm is capable of parsing general CFG's. The big advantage of the CYK algorithm is its efficiency, as its run-time is $O(n^3|\mathcal{G}|)$, where n is the size of the sentence to be parsed and $|\mathcal{G}|$ is the number of non terminal symbols in our grammar. No algorithm can do better for general context-free grammars, although there are faster algorithms on more restricted grammars (LR languages can be parsed in linear time, (Russell, 1995)).

The key idea of this algorithm is the following: given the set of all possible non-terminal symbols, for all sub-strings of length k, derive all the possible non-terminal symbols for every sub-string of length $k+1$ by considering all possible partitions of this string into two subs-strings. To have a quick reference to sub-strings of sentence s, we introduce indices before and after every letter as shown in the figure below:

$${}_i\underbrace{w_{i+1}\cdots w_k}\,\underbrace{w_{k+1}\cdots w_j}$$

Figure 2. Partitioning a sentence to two sub-strings

Now, we define a function which returns all possible non-terminal symbols that can generate the sub-string defined from index i until index j.

Formally, we define a function:

$$\text{SYM} : \{0, 1, \cdots, n\}^2 \longrightarrow 2^{\mathcal{N}}$$

(where $2^{\mathcal{N}}$ is the powerset) with the following recursive formula:

$$\boxed{\text{SYM}(i,j) = \bigcup_{i<k<j} \sigma\big[\text{SYM}(i,k)\text{SYM}(k+1,j)\big]}$$

where n is the length of the sentence s, $\mathcal{N}$ is the set of non-terminal symbols in our grammar and σ is a function that takes the concatenation of two non-terminal symbols and returns the set of non-terminals in the left hand side of all rules that have the input of the function as their right hand side (for example for the grammar we defined above we have $\sigma(AB) = S$, $\sigma(AS) = X$ and $\sigma(AA) = \emptyset$).

The program will accept the sentence s, if $S \in \text{SYM}(0, n)$, where S is the start symbol of our grammar. As in every dynamic program, we can use memoization to speed up the performance of the algorithm. In this case, we can compute the function SYM by increasing i, j and filling an upper triangular matrix by columns. Notice that in order to fill entry (i, j) of the matrix we simply need to know the value of the function in entries $(i, k), (k, j) \forall i < k < j$. This observation is depicted in the figure below only for a single value of the variable k:

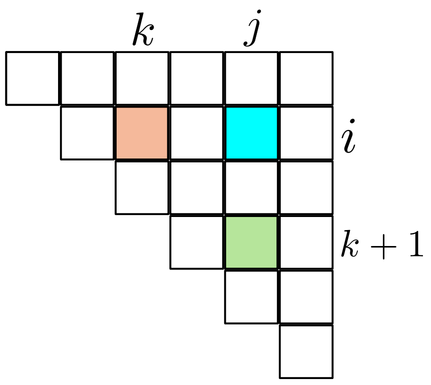

Figure 3. Filling the (i, j) cell

Next we present the pseudocode for the CYK algorithm:

Algorithm 3 CYK Dynamic Programming Recognizer

```
1:  function CYK($frag, G$)
2:                      ▷ Where $frag$ -fragment, $G$ -grammar
3:
4:      Define $n = length(frag)$
5:      for $j \in \{1 \dots n\}$ do
6:          $table[j-1, j] = \{R | R \rightarrow t_{j-1,j} \in P\}$
7:          for $i \in \{j-2 \dots 0\}$ do
8:              for $k \in \{i+1 \dots j-1\}$ do
9:                  $table[i, j] = table[i, j]$
10:                 $\cup \{R | R \rightarrow BC \in P \,\&\, B \in table[i, k]$
11:                 $\&\, C \in table[k, j]\}$
12:             end for
13:         end for
14:     end for
15:     if $table[0, n] = S$ then
16:         return True
17:     else
18:         return False
19:     end if
20: end function
```

To obtain a derivation of s from our grammar we can trace back our SYM matrix, using backpointers. We could also store parse trees as we fill in the matrix, but that would be less space efficient (Davis).

6. Implementation in Scala

In order to represent our grammar we used various Scala features. The basic component of a CFG is the grammar rules, but before we could define the rules we had to create a representation for the right hand side of every rule. More precisely, we represented the right hand side of a rule as a `sealed trait` which corresponds to Java Interfaces and provides a good alternative for enumerations. A sealed trait may not be directly extended unless the inheriting class is in the same source file. That allows the compiler to warn us of exhaustive match cases. Further, we used `final case` classes in order to represent terminal and non terminal symbols, the rules and ultimately the grammar. We choose to use final classes since they should never be extended. This is shown in the following code snippet:

```scala
 //definition of the RHS
 sealed trait RHS
 final case class T(t:String)extends RHS
 final case class NT(nt:Symbol)extends RHS

//definition of Rule
final case class Rule(left:NT,
     right:List[RHS])

 //definition of Grammar
 final case class CFG(start:NT,
     rules:List[Rule])
```

Scala is a programming language that is general purpose but at the same time concise, elegant and type-safe. Throughout our program we were able to express succinctly complex ideas by using pattern matching combined with recursion. Moreover, the ability to create partially applied functions that we could pass as variables to other functions was one of Scala's greatest advantages when compared to many OO programming languages.

Finally, we used the `sbt` build tool that allowed us to organize and test our project. Therefore, in our bitbucket repository we have adopted the `sbt` folder hierarchy and we have separated source from test files. Therefore, source files can be found in `Code/src/main/scala`, whereas, test files can be found at `Code/src/test/scala`. In order to test the files, we simply have to initiate `sbt` from the command line and type `test`. There exist in total 3 source files that consists of approximately 400 lines of code, while, there are also corresponding test files that consist of approximately 200 lines of code.

References

Altshuler, Dor. Cky and early parsing algorithms.

Davis, Ernest. Cyk-parse algorithm with tree recovery.

Russell, Norvig. *Artificial Intelligence: a moder approach*. Alan Apt, 1995.