Universiteit Leiden

Opleiding Informatica

An On-Line Parsing Algorithm for conjunctive grammars

Name: Michel Rensen

Date: 13/06/2014

1st supervisor: Marcello Bonsangue
2nd supervisor: Hendrik Jan Hoogeboom

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1 Introduction

Context-free grammars are a powerful tool to generate context-free languages. These grammars may be used for the monitoring of programs with procedures. Conjunctive grammars are an extension of context-free grammars. With conjunctive grammars it is possible to monitor a program with procedures simultaneously on separate specifications.

The language $\{a^n b^n c^n | n \geq 0\}$ is one of the better known examples of a language that cannot be generated by a context-free grammar. This language can be generated by a conjunctive grammar.

In this paper we present an on-line parsing algorithm for conjunctive grammars. We first present a matching algorithm which is based on derivatives for regular expressions, modified to work on conjunctive grammars. Then we show that the step from our matching algorithm to a parsing algorithm is a small step.

On-line parsing makes it possible to extend the given input string, while the algorithm has already done some calculations. Therefore the input string does not have to be known at the beginning of the algorithm.

In section 2 we discuss the definitions of grammars and the associated languages. In section 3 we define some other terms and functions that we need for the algorithm. Then in section 4 we introduce the derivatives and present the matching algorithm. We show the working of the algorithm with some examples in section 5. In the final section we show how the matching algorithm can be extended to a parsing algorithm on the basis of one of the examples.

2   Grammars and Languages

In this paper we write the empty set as 0 or $\emptyset$ and the empty word as 1 or Λ.

2.1   Conjunctive Grammars

Conjunctive grammars are an extension to context-free grammars [1]. With the additional possibility to use conjunctions in productions. Conjunctive grammars can generate more languages than context-free grammars can.
To assure that our parsing algorithm is decidable, we use the Greibach Normal Form (GNF). We know that every Context-free grammar can be transformed into a Greibach Normal Form [2] while still being able to generate the same languages. A similar result for conjunctive grammars is not known, but we restrict the grammars accepted by our algorithm to have their productions written in a similar form.
Because our algorithm is based on derivations for regular expressions, the productions of the grammars have to be written in a similar way. Therefore we define a conjunctive term.

Definition 2.1. A conjunctive term (CT) ε is a string of the form

$$\varepsilon ::= 0|1|\sigma|X|\varepsilon + \varepsilon|\varepsilon \cdot \varepsilon|\varepsilon \& \varepsilon$$

where σ is an alphabet symbol and X is a variable.

We call a conjunctive term which has the same form as the Greibach Normal Form a Greibach conjunctive term.

Definition 2.2. A Greibach conjunctive term (GCT) γ is a string of the form

$$\gamma ::= 0|1|\gamma + \gamma|\sigma \cdot \varepsilon|\gamma \& \gamma$$

where σ is an alphabet symbol and ε is a conjunctive term.

The only difference between a general conjunctive term and a Greibach conjunctive term is that in a GCT a concatenation always starts with a σ.
All GCT's are also CT's. All subexpressions of a GCT are also a CT.

Definition 2.3. A conjunctive grammar is a 4-tuple $G = (V, \Sigma, S, P)$, where

- V is a set of variables

- Σ is the alphabet, a finite set of terminal symbols

- $S \in V$ is the start variable

- P is a set of productions, where $P : V \to GCT(V \cup \Sigma)$

We see that P is a function from V to $GCT(V \cup \Sigma)$. Therefore we can use $P(X)$ to refer to the production of the variable X. Given the production $X \to \varepsilon$, $P(X) = \varepsilon$.

2.2 Languages

First we define the languages generated by conjunctive terms. Because all GCT's are CT's, this also defines the languaged generated by GCT's.

Definition 2.4. Given a conjunctive term ε, $L(\varepsilon)$ is the least subset of Σ^* such that

$$L(0) = \emptyset$$
$$L(1) = \{\Lambda\}$$
$$L(\sigma) = \{\sigma\}$$
$$L(X) = L(P(X))$$
$$L(\varepsilon \cdot \varepsilon') = L(\varepsilon) \cdot L(\varepsilon')$$
$$L(\varepsilon + \varepsilon') = L(\varepsilon) \cup L(\varepsilon')$$
$$L(\varepsilon \& \varepsilon') = L(\varepsilon) \cap L(\varepsilon')$$

S is the start symbol of a variable. Therefore the language generated by a conjunctive grammar G is the same languages as the language that is generated by S.

Definition 2.5. Given a conjunctive grammar $G = (V, \Sigma, S, P)$, the language generated by the grammar G is

$$L(G) = L(S)$$

Conjunctive grammars can generate any context-free language. With the addition of the conjunction, even more languages can be generated. The following example shows a language that cannot be generated with a context-free grammar, but can be generated with a conjunctive grammar.

Example 2.6. The following conjunctive grammar generates the language $\{a^n b^n c^n | n \geq 0\}$:

$$S \to (aAB \& aDbC) + 1$$
$$A \to aA + 1$$
$$B \to bBc + 1$$
$$C \to cC + 1$$
$$D \to aDb + 1$$

Note that this grammar is correctly written in the Greibach Normal Form as defined in Definition 2.2, as every string of concatenated symbols and variables starts with a symbol.

3   Other Terms and Functions

Before we can start working with the derivatives, we first need to define a few terms and functions.
We start with the definition of equivalence on conjunctive terms.

Definition 3.1. $\varepsilon \equiv \varepsilon'$ iff $L(\varepsilon) = L(\varepsilon')$

Now we define the $\leq$-operator on conjunctive terms.

Definition 3.2. $\varepsilon' \leq \varepsilon$ iff $\varepsilon \equiv \varepsilon'' + \varepsilon'$

We can easily see that $L(\varepsilon') \subseteq L(\varepsilon)$ if $\varepsilon' \leq \varepsilon$.

3.1   Nullability

By [3] the nullability of variables in CFG's is defined as

Definition 3.3. Given a context-free grammar,

1. Every variable A for which there is a production $A \to \Lambda$ is nullable

2. If $A_1, A_2, ..., A_k$ are nullable variables (not necessarily distinct), and

$$B \to A_1 A_2 ... A_k$$

   is a production, then B is nullable

As our algorithm only accepts grammars, whose productions are in the *Greibach Normal Form*, the second definition is impossible as every right side of a production starts with a symbol in Σ or is 0 or 1. Therefore we only look at the first definition.

Definition 3.4. We say that a variable X in a conjunctive grammar G is nullable if $P(X)$ is nullable. Whether a Greibach conjunctive term γ is nullable, is defined inductively as follows:

- 1 is nullable

- $\gamma_1 + \gamma_2$ is nullable if γ_1 is nullable or γ_2 is nullable

- $\gamma_1 \& \gamma_2$ is nullable if γ_1 is nullable and γ_2 is nullable

It then follows that a variable X is nullable iff $1 \leq P(X)$.

3.2   Empty Word Checking

The function $o(\varepsilon)$ checks whether Λ is in the language generated by a given conjunctive term. This function is needed to find out if a given string is actually accepted by a given grammar.

The following rules calculate the whether Λ is in the language of a given CT ε:

$$o(0) = 0$$
$$o(1) = 1$$
$$o(\sigma) = 0$$
$$o(X) = 1 \text{ iff } X \text{ is nullable (Definition 3.4)}$$
$$o(\varepsilon + \varepsilon') = MAX\{o(\varepsilon), o(\varepsilon')\}$$
$$o(\varepsilon \cdot \varepsilon') = MIN\{o(\varepsilon), o(\varepsilon')\}$$
$$o(\varepsilon \& \varepsilon') = MIN\{o(\varepsilon), o(\varepsilon')\}$$

We prove this function with a proof by induction on the structure of ε. That means that the theorem holds, if it holds for all possible structures of ε as defined in Definition 2.1.

Theorem 3.5. *The function $o(\varepsilon)$ checks whether $\Lambda \in L(\varepsilon)$, therefore*

$$o(\varepsilon) = \begin{cases} 1 & \Lambda \in L(\varepsilon) \\ 0 & \Lambda \notin L(\varepsilon) \end{cases}$$

Proof. Proof by induction on the structure of ε:

Keeping the same order as Definition 2.1, we start with ε being of the form 0. The rule we use to prove this structure is: $o(0) = 0$.

$$o(0) = 0 = \begin{cases} 1 & \Lambda \in \emptyset \\ 0 & \Lambda \notin \emptyset \end{cases} = \begin{cases} 1 & \Lambda \in L(0) \\ 0 & \Lambda \notin L(0) \end{cases}$$

The following possible structure is $\varepsilon ::= 1$. The associated rule is: $o(1) = 1$.

$$o(1) = 1 = \begin{cases} 1 & \Lambda \in \{\Lambda\} \\ 0 & \Lambda \notin \{\Lambda\} \end{cases} = \begin{cases} 1 & \Lambda \in L(1) \\ 0 & \Lambda \notin L(1) \end{cases}$$

If ε is a $\sigma \in \Sigma$, the rule is: $o(\sigma) = 0$.

$$o(\sigma) = 0 = \begin{cases} 1 & \Lambda \in \{\sigma\} \\ 0 & \Lambda \notin \{\sigma\} \end{cases} = \begin{cases} 1 & \Lambda \in L(\sigma) \\ 0 & \Lambda \notin L(\sigma) \end{cases}$$

For the structure of $\varepsilon \in V$ we have the rule $o(X) = 1$iff$X$ is nullable. To prove this, we need to use Definition 3.4. Note that both 1 and Λ are notations for the empty word.

$$o(X) = 1 \text{ iff } X \text{ is nullable } = 1 \text{ iff } (1 \leq P(X))$$
$$= \begin{cases} 1 & 1 \leq P(X) \\ 0 & \text{else} \end{cases} = \begin{cases} 1 & 1 \in L(P(X)) \\ 0 & \text{else} \end{cases}$$
$$= \begin{cases} 1 & 1 \in L(X) \\ 0 & \text{else} \end{cases} = \begin{cases} 1 & \Lambda \in L(X) \\ 0 & \Lambda \notin L(X) \end{cases}$$

If the CT has a +, we use the rule: $o(\varepsilon + \varepsilon') = MAX\{o(\varepsilon), o(\varepsilon')\}$.

$$o(\varepsilon + \varepsilon') = MAX\{o(\varepsilon), o(\varepsilon')\} = \begin{cases} 1 & o(\varepsilon) = 1 \vee o(\varepsilon') = 1 \\ 0 & o(\varepsilon) = 0 \wedge o(\varepsilon') = 0 \end{cases}$$

$$= \begin{cases} 1 & \Lambda \in (L(\varepsilon) \cup L(\varepsilon')) \\ 0 & \Lambda \notin (L(\varepsilon) \cup L(\varepsilon')) \end{cases} = \begin{cases} 1 & \Lambda \in L(\varepsilon + \varepsilon') \\ 0 & \Lambda \notin L(\varepsilon + \varepsilon') \end{cases}$$

For concatenation we use the rule: $o(\varepsilon \cdot \varepsilon') = MIN\{o(\varepsilon), o(\varepsilon')\}$.

$$o(\varepsilon \cdot \varepsilon') = MIN\{o(\varepsilon), o(\varepsilon')\} = \begin{cases} 1 & o(\varepsilon) = 1 \wedge o(\varepsilon') = 1 \\ 0 & o(\varepsilon) = 0 \vee o(\varepsilon') = 0 \end{cases}$$

$$= \begin{cases} 1 & \Lambda \in L(\varepsilon) \cdot L(\varepsilon') \\ 0 & \Lambda \notin L(\varepsilon) \cdot L(\varepsilon') \end{cases} = \begin{cases} 1 & \Lambda \in L(\varepsilon \cdot \varepsilon') \\ 0 & \Lambda \notin L(\varepsilon \cdot \varepsilon') \end{cases}$$

Finally we have to prove the theorem for the new structure in conjunctive grammars. For the conjunction we use the rule: $o(\varepsilon \& \varepsilon') = MIN\{o(\varepsilon), o(\varepsilon')\}$.

$$o(\varepsilon \& \varepsilon') = MIN\{o(\varepsilon), o(\varepsilon')\} = \begin{cases} 1 & o(\varepsilon) = 1 \wedge o(\varepsilon') = 1 \\ 0 & o(\varepsilon) = 0 \vee o(\varepsilon') = 0 \end{cases}$$

$$= \begin{cases} 1 & \Lambda \in L(\varepsilon) \& L(\varepsilon') \\ 0 & \Lambda \notin L(\varepsilon) \& L(\varepsilon') \end{cases} = \begin{cases} 1 & \Lambda \in L(\varepsilon \& \varepsilon') \\ 0 & \Lambda \notin L(\varepsilon \& \varepsilon') \end{cases}$$

$\square$

4 Matching Algorithm

4.1 Derivatives

We use the following rules to calculate the σ-derivative of a CT for any $\sigma \in \Sigma$ [4].

$$0_\sigma = 0$$
$$1_\sigma = 0$$
$$\sigma_\sigma = 1$$
$$\sigma'_\sigma = 0 (\sigma' \neq \sigma)$$
$$X_\sigma = P(X)_\sigma$$
$$(\varepsilon + \varepsilon')_\sigma = \varepsilon_\sigma + \varepsilon'_\sigma$$
$$(\varepsilon \cdot \varepsilon')_\sigma = \varepsilon_\sigma \cdot \varepsilon' + o(\varepsilon) \cdot \varepsilon'_\sigma$$
$$(\varepsilon \& \varepsilon')_\sigma = \varepsilon_\sigma \& \varepsilon'_\sigma$$

The derivatives of strings are defined as

$$\varepsilon_\Lambda = \varepsilon$$
$$\varepsilon_{\sigma w} = (\varepsilon_\sigma)_w$$

The σ-derivative calculates the conjunctive term that is left if a σ is removed from the front of the conjunctive term. The language of a σ-derivative of a conjunctive term contains all words w for which $\sigma w \in L(\varepsilon)$. Therefore we find the following theorem. We prove it with a proof by induction on the structure of ε. We start with this lemma

Lemma 4.1. *For a Greibach conjunctive term γ it holds that for all σ*

$$L(\gamma_\sigma) = \{w | \sigma w \in L(\gamma)\}$$

Proof. We proceed by induction on the structure of γ.
In case $\gamma = 0$
$$0_\sigma = 0 \text{ thus } L(0_\sigma) = L(0) = \emptyset = \{w | \sigma w \in L(0)\}$$

In case $\gamma = 1$
$$1_\sigma = 0 \text{ thus } L(1_\sigma) = L(0) = \emptyset = \{w | \sigma w \in L(1)\}$$

In case $\gamma = \gamma_1 + \gamma_2$ we know that $(\gamma_1 + \gamma_2)_\sigma = \gamma_{1\sigma} + \gamma_{2\sigma}$ thus

$$L((\gamma_1 + \gamma_2)_\sigma) = L(\gamma_{1\sigma} + \gamma_{2\sigma}) = L(\gamma_{1\sigma}) \cup L(\gamma_{2\sigma})$$
$$= \{w | \sigma w \in L(\gamma_1)\} \cup \{w | \sigma w \in L(\gamma_2)\}$$
$$= \{w | \sigma w \in L(\gamma_1) \cup L(\gamma_2)\} = \{w | \sigma w \in L(\gamma_1 + \gamma_2)\}$$

In case $\gamma = \sigma' \varepsilon$ we know that $(\sigma' \varepsilon)_\sigma = \sigma'_\sigma \varepsilon + o(\sigma') \varepsilon_\sigma$ and $o(\sigma') = 0$ thus

$$L((\sigma' \varepsilon)_\sigma) = L(\sigma'_\sigma \varepsilon)$$
1) if $\sigma' = \sigma$ then $L(\sigma'_\sigma \varepsilon) = L(1\varepsilon) = L(\varepsilon) = \{w | \sigma w \in L(\sigma \varepsilon)\} = \{w | \sigma w \in L(\sigma' \varepsilon)\}$
2) if $\sigma' \neq \sigma$ then $L(\sigma'_\sigma \varepsilon) = L(0\varepsilon) = L(0) = \{w | \sigma w \in L(\sigma' \varepsilon)\}$

In case $\gamma = \gamma_1 \& \gamma_2$ we know that $(\gamma_1 \& \gamma_2)_\sigma = \gamma_{1\sigma} \& \gamma_{2\sigma}$ thus

$$L((\gamma_1 \& \gamma_2)_\sigma) = L(\gamma_{1\sigma} \& \gamma_{2\sigma}) = L(\gamma_{1\sigma}) \cap L(\gamma_{2\sigma})$$
$$= \{w|\sigma w \in L(\gamma_1)\} \cap \{w|\sigma w \in L(\gamma_2)\}$$
$$= \{w|\sigma w \in L(\gamma_1) \cap L(\gamma_2)\} = \{w|\sigma w \in L(\gamma_1 \& \gamma_2)\}$$

$\square$

Now we can prove the same theorem for conjunctive terms.

Theorem 4.2. *Given a conjunctive term ε, the language generated by the σ-derivative of ε is*

$$L(\varepsilon_\sigma) = \{w|\sigma w \in L(\varepsilon)\}$$

Proof. Proof by induction on the structure of ε:

Again we start with the possible structure of $\varepsilon ::= 0$. We use the rule: $0_\sigma = 0$.

$$L(0_\sigma) = L(0) = \emptyset = \{w|\sigma w \in L(0)\}$$

If ε has the structure 1, we use the rule: $1_\sigma = 0$:

$$L(1_\sigma) = L(0) = \emptyset = \{w|\sigma w \in L(\{\Lambda\})\} = \{w|\sigma w \in L(1)\}$$

In case ε is a $\sigma \in \Sigma$, we have two rules. We use $\sigma_\sigma = 1$ if ε and σ are the same alphabet symbol. We use $\sigma'_\sigma = 0(\sigma' \neq \sigma)$ if they are different. Thus we have two cases if ε is of the form σ.

  1) if $\sigma' = \sigma$ then $L(\sigma'_\sigma) = L(1) = \{\Lambda\} = \{w|\sigma w \in \{\sigma\}\} = \{w|\sigma w \in L(\sigma')\}$
  2) if $\sigma' \neq \sigma$ then $L(\sigma'_\sigma) = L(0) = \emptyset = \{w|\sigma w \in \{\sigma'\}\} = \{w|\sigma w \in L(\sigma')\}$

In case ε is a variable X we have

$$L(X_\sigma) = L(P(X)_\sigma)$$
$$= \{w|\sigma w \in L(P(X))\} \text{ by Lemma 4.1}$$
$$= \{w|\sigma w \in L(X)\}$$

For the logical-or, we use the rule: $(\varepsilon + \varepsilon')_\sigma = \varepsilon_\sigma + \varepsilon'_\sigma$.

$$L((\varepsilon + \varepsilon')_\sigma) = L(\varepsilon_\sigma + \varepsilon'_\sigma) = L(\varepsilon_\sigma) \cup L(\varepsilon'_\sigma)$$
$$= \{w|\sigma w \in L(\varepsilon)\} \cup \{w|\sigma w \in L(\varepsilon')\}$$
$$= \{w|\sigma w \in (L(\varepsilon) \cup L(\varepsilon')\}$$
$$= \{w|\sigma w \in L(\varepsilon + \varepsilon')\}$$

When ε has a concatenation, we use the rule: $(\varepsilon \cdot \varepsilon')_\sigma = \varepsilon_\sigma \cdot \varepsilon' + o(\varepsilon) \cdot \varepsilon'_\sigma$. We have two

cases:

1) if $o(\varepsilon) = 0$ then $L((\varepsilon \cdot \varepsilon')_\sigma) = L(\varepsilon_\sigma \cdot \varepsilon') = L(\varepsilon_\sigma) \cdot L(\varepsilon')$

$$= \{u|\sigma u \in L(\varepsilon)\} \cdot \{v|v \in L(\varepsilon')\}$$
$$= \{uv|\sigma u \in L(\varepsilon) \wedge v \in L(\varepsilon')\}$$
$$= \{uv|\sigma uv \in L(\varepsilon \cdot \varepsilon')\}$$
$$= \{w|\sigma w \in L(\varepsilon \cdot \varepsilon')\}$$

2) if $o(\varepsilon) = 1$ then $L((\varepsilon \cdot \varepsilon')_\sigma) = L(\varepsilon_\sigma \cdot \varepsilon' + \varepsilon'_\sigma) = L(\varepsilon_\sigma) \cdot L(\varepsilon') \cup L(\varepsilon'_\sigma)$

$$= \{u|\sigma u \in L(\varepsilon)\} \cdot \{v|v \in L(\varepsilon')\} \cup \{w|\sigma w \in L(\varepsilon')\}$$
$$= \{uv|\sigma u \in L(\varepsilon) \wedge v \in L(\varepsilon')\} \cup \{w|\sigma w \in L(\varepsilon')\}$$
$$= \{uv|\sigma uv \in L(\varepsilon \cdot \varepsilon')\} \cup \{w|\sigma w \in L(\varepsilon')\}$$
$$= \{w|\sigma w \in L(\varepsilon \cdot \varepsilon')\} \cup \{w|\sigma w \in L(\varepsilon')\}$$
$$= \{w|\sigma w \in L(\varepsilon \cdot \varepsilon') \vee \sigma w \in L(\varepsilon')\}$$
$$= \{w|\sigma w \in L(\varepsilon \cdot \varepsilon')\}, \text{ because } \Lambda \in L(\varepsilon)$$

When ε is an conjunction, we use the rule: $(\varepsilon \& \varepsilon')_\sigma = \varepsilon_\sigma \& \varepsilon'_\sigma$.

$$L((\varepsilon \& \varepsilon')_\sigma) = L(\varepsilon_\sigma \& \varepsilon'_\sigma) = L(\varepsilon_\sigma) \cap L(\varepsilon'_\sigma) = \{u|\sigma u \in L(\varepsilon)\} \cap \{v|\sigma v \in L(\varepsilon')\}$$
$$= \{w|\sigma w \in (L(\varepsilon) \cap L(\varepsilon'))\} = \{w|\sigma w \in L(\varepsilon \& \varepsilon')\}$$

$\square$

4.2 Matching Algorithm

The matching algorithm needs to check if a given string w is in the language generated by a given grammar G or not. We define our algorithm as

Definition 4.3. Given a conjunctive grammar G and a string w,

$$w \in L(G) \text{ iff } o(S_w) = 1$$

First we compute the derivative of the given string w, which consecutively computes the derivatives of the seperate symbols. As $S_w = \{u|wu \in L(S)\}$, if $\Lambda \in L(S_w)$, $w \in L(S)$. So we check if Λ is in the remaining string, therefore $w \in L(G)$ iff $o(S_w) = 1$.

5 Examples

To give a better idea of the working of the matching algorithm, we give a few examples in which we completely work out the algorithm for a given conjunctive grammar and a given string.

Example 5.1. Given the following conjunctive grammar G

$$S \to (aAB \& aDbC) + 1$$
$$A \to aA + 1$$
$$B \to bBc + 1$$
$$C \to cC + 1$$
$$D \to aDb + 1$$

This is the conjunctive grammar given in example 2.6. Given the word $w = abc$ we compute the algorithm as follows:

$$S_w = S_{abc} = (S_a)_{bc} = ((S_a)_b)_c$$

$$
\begin{aligned}
S_a = P(S)_a &= ((aAB \& aDbC) + 1)_a = (aAB \& aDbC)_a + 1_a \\
&= ((aAB)_a \& (aDbC)_a) + 0 = (a_a AB + o(a) \cdot (AB)_a) \& (a_a DbC + o(a) \cdot (DbC)_a) \quad (1) \\
&= (1AB + 0(AB)_a) \& (1DbC + 0(DbC)_a) = AB \& DbC
\end{aligned}
$$

$$(S_a)_b = (AB \& DbC)_b = (AB)_b \& (DbC)_b$$

To make it clearer, we compute $(AB)_b$ and $(DbC)_b$ separately.

$$
\begin{aligned}
(AB)_b &= A_b B + o(A) B_b = (A_b B + 1 B_b) = P(A)_b B + P(B)_b \\
&= (aA + 1)_b + (bBc + 1)_b = ((aA)_b) + 1_b) + ((bBc)_b + 1_b) \\
&= ((a_b A + o(a) A_b) + 0) + ((b_b Bc + o(b)(Bc)_b) + 0) \\
&= (0A + 0A_b) + (1Bc + 0(Bc)_b) = 0 + Bc = Bc
\end{aligned}
$$

$$
\begin{aligned}
(DbC)_b &= D_b bC + o(D)(bC)_b = D_b bC + 1(bC)_b = P(D)_b bC + (b_b C + o(b)C_b) \\
&= (aDb + 1)_b bC + (1C + 0C_b) = ((aDb)_b + 1_b)bC + (C + 0) \\
&= ((a_b Db + o(a)(Db)_b) + 0)bC + C = (0Db + 0(Db)_b)bC + C = 0bC + C = C
\end{aligned}
$$

$$(S_a)_b = (AB)_b \& (DbC)_b = Bc \& C$$

$$
\begin{aligned}
((S_a)_b)_c &= (Bc \& C)_c = (Bc_c) \& C_c = B_c c \& C_c = (P(B)_c c + o(B)c_c) \& P(C)_c \\
&= ((bBc + 1)_c c + 1c_c) \& (cC + 1)_c = (((bBc)_c + 1_c)c + c_c) \& (c_c C + 1_c) \\
&= (((bBc)_c + 0) + 1) \& (1C + 0) \\
&= ((bBc)_c + 1) \& C = 1 \& C
\end{aligned}
$$

$$
\begin{aligned}
S_w = S_{abc} &= ((S_a)_b)_c = 1 \& C \\
o(S_w) &= o(1 \& C) = MIN\{o(1), o(C)\} = MIN\{1, 1\} = 1
\end{aligned}
$$

Out of definition 4.3, we know that $w \in L(G)$ iff $o(S_w) = 1$. We just calculated that $o(S_{abc}) = 1$, so $abc \in L(G)$.

Example 5.2. Using the same grammar G as given in example 5.1 we now use the matching algorithm on the string aa.

$$S_w = S_{aa} = (S_a)_a$$

As we already calculated it in (1) we know that

$$S_a = AB\&DbC$$

$$(S_a)_a = (AB\&DbC)_a = (AB)_a\&(DbC)_a$$

Again we compute $(AB)_a$ and $(DbC)_a$ separately.

$$
\begin{aligned}
(AB)_a &= A_aB + o(A)B_a = P(A)_aB + 1B_a = P(A)_aB + P(B)_a \\
&= (aA + 1)_a + (bBc + 1)_a = ((aA)_a + 1_a) + ((bBc)_a + 1_a) \\
&= ((a_aA + o(a)A_a) + 0) + ((b_aBc + o(b)(Bc)_a) + 0) \\
&= (1A + 0A_a) + (0Bc + 0(Bc)_a) = A + 0 = A
\end{aligned}
$$

$$
\begin{aligned}
(DbC)_a &= D_abC + o(D)(bC)_a = P(D)_abC + 1(bC)_a \\
&= P(D)_abC + (b_aC + o(b)C_a) = (aDb + 1)_a + (0C + 0C_a) \\
&= ((aDb)_a + 1_a) + 0 = (a_aDb + o(a)(Db)_a) + 0 \\
&= 1Db + 0(Db)_a = Db
\end{aligned}
$$

$$S_w = S_{aa} = (S_a)_a = (AB)_a\&(DbC)_a = A\&Db$$

$$
\begin{aligned}
o(S_w) &= o(A\&Db) = MIN\{o(A), o(Db)\} \\
&= MIN\{1, MIN\{o(D), o(b)\}\} = MIN\{1, MIN\{1, 0\}\} = MIN\{1, 0\} = 0
\end{aligned}
$$

As $w \in L(G)$ iff $o(S_w) = 1$ we know that $o(S_{aa}) = 0$, so $aa \notin L(G)$.

6 Parsing

A parsing algorithm should not only know if an input string is accepted by the language of a grammar, but also how. We can represent this with a parse tree. The step from our matching algorithm to a parsing algorithm is not very big. For example 5.1 we can deduce the parse tree in figure 6. By keeping track of the usage three of the derivatives, we know how the string is parsed by our algorithm.

We need to keep track of the following derivative rules:

1. $\sigma_\sigma = 1$

2. $X_\sigma = P(X)_\sigma$

3. $(\varepsilon \cdot \varepsilon')_\sigma = \varepsilon_\sigma \cdot \varepsilon' + o(\varepsilon) \cdot \varepsilon'_\sigma$

With the first rule we know that at that moment the σ in the input string is from this precise moment. With the second rule we have to check which part of the production of X is actually used for the input string.

We only need to keep track of the third rule when $o(\varepsilon) = 1$. This is only possible if ε is a variable. This variable has a production of the form $\varepsilon \to \varepsilon' + 1$ and the variable goes to 1 in this branch.

When a variable goes to a conjunctive term with an conjunction, we can split that conjunction and show that in both sides of the conjunction there is a path to the input string.
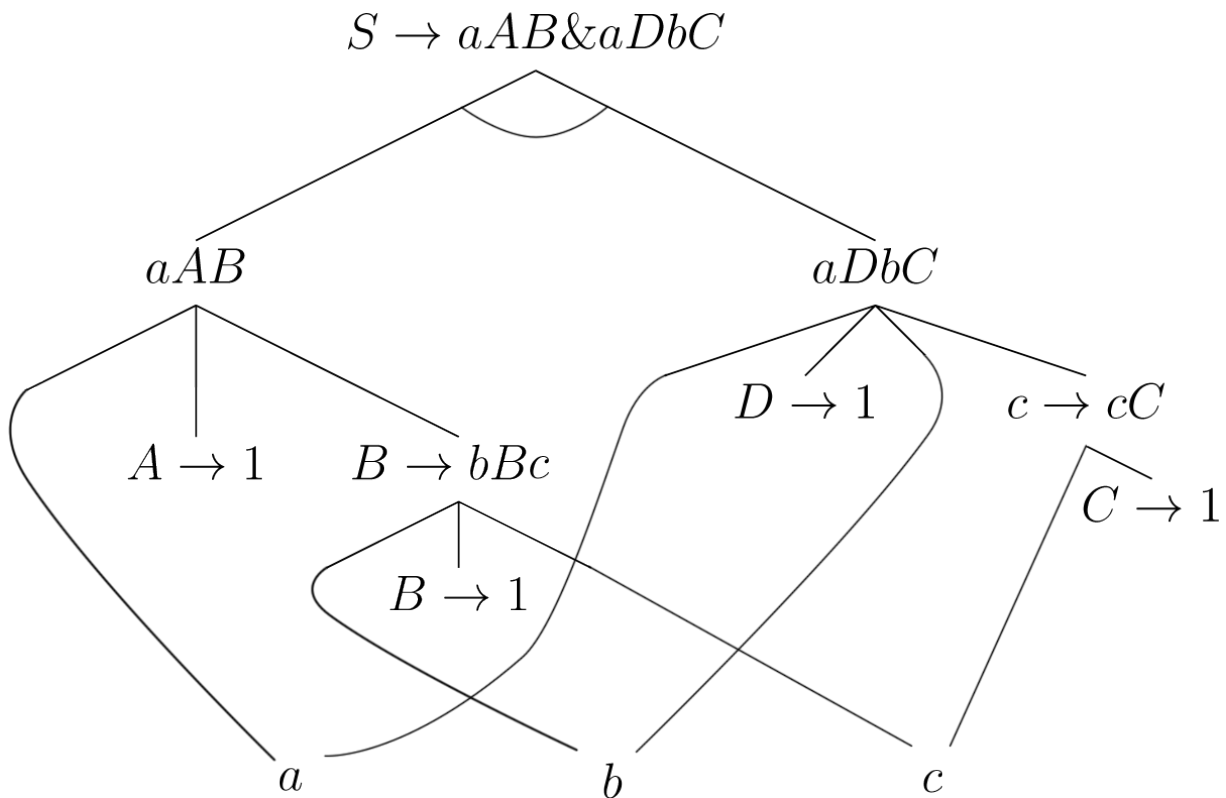

Figure 1: Parse tree for the string abc according to the grammar given in example 5.1.

14

7 Conclusion

Our algorithm calculates the derivatives of strings by separately calculating the derivatives of single symbols. This makes it possible to extend the string, while the derivatives of a part of the string have already been calculated. In this way the string does not have to be known at the start of our algorithm.

We also know that our algorithm is decidable. Because of the Greibach Normal Form, we know that every derivative can calculate a conjunctive term.

For further research it would be useful to eliminate the Greibach Normal Form, as it is not yet known if conjunctive grammars in Greibach Normal Form and general conjunctive grammars generate the same languages.

It may also be helpful to calculate the complexity of our algorithm.

It is interesting to see if our algorithm could be extended to Boolean Grammars, in which productions could also have *negation*. These grammars can generate even more languages, for example $\{ww|w \in \Sigma^*\}$.

References

[1] Alexander Okhotin: Conjunctive and Boolean grammars: The true general case of the context-free grammars. Computer Science Review 9: 27-59 (2013)

[2] Sheila A. Greibach: A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. J. ACM 12(1): 42-52 (1965)

[3] John C. Martin: Introduction to Languages and the Theory of Computation, fourth edition, McGraw-Hill (2011)

[4] Scott Owens, John H. Reppy, Aaron Turon: Regular-expression derivatives re-examined. J. Funct. Program. 19(2): 173-190 (2009)