Generating efficient table driven parsers for non context free languages

Maurice Gittens <maurice at gittens dot nl>

June 20, 2025

Abstract

This paper introduces a slight modification of the Noam Chomsky's phrase structure grammars called dotted grammars. The grammars will be shown to preserve the generative capabilities of unrestructed phrase structure grammars. In addition it will be shown that dotted grammars allow the mechanical construction of efficient table driven parsers for non context free languages.

Contents

Chapter 1

Preliminaries

1.1 Introduction

This paper introduces a slightly modified version of phrase structure grammars (called deterministic dotted grammars) which allow efficient table driven parsers to be generated. LL(1) and LR(1) grammars are shown to be proper subsets of the class of deterministic dotted grammars. The class of languages accepted by deterministic grammars include at least some context sensitive languages. The implementation of a parser generator for deterministic dotted grammars (which generates C++) is also briefly introduced. To the knowledge of the author no similar of table driven parsers exists in literature.

1.2 Acknowledgments

In this section I would like to acknowledge the supportive comments and critical reading of the following people. Please let it be understood that any errors, omitions, improper use of the English language etc. are all my fault.

- Dick Grune <dick@cs.vu.nl> suggested corrections for early versions of this document. A real motivator.
- John Shutt <jshutt@owl.wpi.edu> spotted a number of errors in the definition of dotted grammars.
- Elena Mauro <elena_mauro@hotmail> suggested corrections for different chapters in this document

1.3 Changes

- January 24 2003; fix a few typographical errors in the section on multi-dot grammars
- December 5 2002; Improve my use of the English language
- April 13 2002; Added corrections pointed out by Elena Mauro <elena_mauro@hotmail.com>
- February 26 2002; Found out about the spell checker in lyx :-). Minor cleanups.

- August 8 2001; Reintroduce the chapter on multi-dot grammars with some corrections. Some people asked for this.
- June 6 2001; An example of parse table generation process; more corrections
- April 24 1997; Corrected some typos
- April 20 1997; Removed multi-dot grammars from this paper. Add a parse table for the recording grammar example.
- April 131997; Added an example recording grammar; corrected a few inaccuracies. Added a section for acknowledgments.
- April 11 1997; Added corrections to chapter 3 by John Shutt <jshutt@owl.wpi.edu>
- May 03 1997; Start using cdot to represent dots in production rules. Corrected a few errors in example parse tables. Corrections for recording grammars.
- March 23 1997; Original version at http://www.gits.nl/grammar.html

Maurice Gittens

December 2002

Chapter 2

Phrase structure grammars

This section presents a definition of Noam Chomsky's phrase structure grammars. Furthermore some terminology, properties and a typical example are also presented.

2.1 Definitions

An unrestricted phrase structure grammar (PSG) is defined as a 4 tuple (N, T, S, P) where:

- N is a nonempty finite set of nonterminal symbols
- T is a non empty finite set of terminal symbols
- S element of N is a distinguished symbol called the start symbol
- P is a set of rewriting rules called production rules of the form:
  $\alpha \rightarrow \beta$
  $\alpha \in (N \cup T)+$containing at least one nonterminal and $\beta \in (N \cup T)*$.

2.2 Sentences and languages

A sentence x for a *PSG G* is an element of $T*$ for which there exists a finite sequence ω_1, ω_2, ... , ω_n and strings$A_i, B_i, C_i, D_i (1 \leq i \leq n-1)$over $(N \cup T)*$ such that:

$\omega_1 = S$

$\omega_i = A_i C_i B_i \quad (1 \leq i \leq n-1)$

$\omega_{i+1} = A_i D_i B_i$ $1 \leq i \leq n-1$ and$C_i \rightarrow D_i$ C_i is an element of P.

$\omega_n = x$

The sequences $\omega_1, \omega_2, ..., \omega_n$ and $\omega_n, ..., \omega_1$ are respectively called a generation sequence and a reduction sequence of a sentence x. [ref. 7] An element $\omega_i (1 \leq i \leq n)$ of these sequences is called a sentential form of G. Thus a sentence for a grammar G may be defined as a sentential form of G consisting solely of terminal symbols. The set of sentences which is generated by a *PSG G* is called the language defined by G. The

language defined by G is denoted by $L(G)$. A sentence is called ambiguous for a *PSG* G if it is generated by more than one generation sequence. A phrase structure grammar is called ambiguous if it generates at least one ambiguous sentence.

2.3 An example

As an example we consider the phrase structure grammar defined by the production rules:

S $\rightarrow$ aSBC

S $\rightarrow$ aBC

CB $\rightarrow$ BC

aB $\rightarrow$ ab

bB $\rightarrow$ bb

bC $\rightarrow$ bc

cC $\rightarrow$ cc

Capital letters represent nonterminals while other alphabetic characters represent terminals. The start symbol is represented by the letter S. This grammar has been shown to generate the language $a^n b^n c^n$ (n $\geq$ 1).

A derivation of the string *aaaabbbbcccc* by this grammar follows:

Sentential form	Production rules
S	S $\rightarrow$ aSBC
aSBC	S $\rightarrow$ aSBC
aaSBCBC	S $\rightarrow$ aSBC
aaaSBCBCBC	S $\rightarrow$ aBC
aaaaBCBCBCBC	aB $\rightarrow$ ab
aaaabCBCBCBC	CB $\rightarrow$ BC
aaaabBCCBCBC	CB $\rightarrow$ BC
aaaabBCBCCBC	CB $\rightarrow$ BC
aaaabBCBCBCC	CB $\rightarrow$ BC
aaaabBCBBCCC	CB $\rightarrow$ BC
aaaabBBCBCCC	CB $\rightarrow$ BC
aaaabBBBCCCC	bB $\rightarrow$ bb
aaaabbBBCCCC	bB $\rightarrow$ bb
aaaabbbBCCCC	bB $\rightarrow$ bb
aaaabbbbCCCC	bC $\rightarrow$ bc
aaaabbbbcCCC	cC $\rightarrow$ cc
aaaabbbbccCC	cC $\rightarrow$ cc
aaaabbbbcccC	cC $\rightarrow$ cc
aaaabbbbcccc	

As might be noted, an important problem faced when generating sentences with these grammars is the problem of deciding at which position of a sentential form rewriting should take place. Another issue is to decide which production rule such be used to rewrite a particular sentential form.

2.4 The Chomsky hierarchy

2.4.1 Unrestricted phrase structure grammars

These grammars have no restrictions on the forms of their production rules. These grammars are called type 0 grammars. The set of languages generated by these grammars is called the set of recursively enumerable languages.

2.4.2 Context sensitive grammars

Let $\alpha \rightarrow \beta$ be a production rule of a phrase structure grammar G. If for all production rules of G it holds that $|\alpha| \leq |\beta|$ then G is a context sensitive or monotonic grammar. Since no production rule shortens the length of sentential forms, a key property of these grammars is that the length of a sentential forms of a grammar G for a sentence $x \in L(G)$ is never greater than $|x|$. These grammars are called type 1 grammars. The languages defined by these grammars are called context sensitive languages.

2.4.3 Context free grammars

These phrase structure grammars have production rules of the form: $A \rightarrow \beta$ where $A \in N$ and $\beta \in (N \cup T)+$. Let G be a context free phrase structure grammar. It has been shown that every $x \in L(G)$ can be derived by applying no more than x production rules. These grammars are also called type 2 grammars. The languages generated by context free grammars are called context free languages.

2.4.4 Right linear grammars

Right linear grammars have production rules of the form:

1. $A \rightarrow x$
2. $A \rightarrow xA'$

where A and A' represent arbitrary nonterminals and x represents an arbitrary terminal symbol. These grammars are also called type 3 grammars. The languages generated by these grammars are called regular languages.

Type 3 languages are a proper subset of type 2 languages which are a proper subset of type 1 languages which are a proper subset of the recursively enumerable languages.

Chapter 3

Dotted grammars

3.1 Introduction

In general it has not proven to be an easy task to define languages using unrestricted phrase structure grammars. Non context free grammars, in general, do not allow simple notions of "flow of control" to be deduced from the grammars. To solve this problem, while maintaining the generative capacity of these grammars, a dot is introduced on both sides of the production rules of phrase structure grammars. The dot functions as an oracle which reveals the location in sentential forms where rewriting should take place.

3.2 Definition

A dotted grammar (DG) is defined as a *PSG* to which information which supports the notion of a 'current rewriting symbol', is added. More formally a *DG* is defined as a 4 tuple (N, T, S, P)

- N is a non empty finite set of nonterminal symbols
- T is a non empty finite set of terminal symbols
- S element of N is a distinguished symbol called the start symbol
- P is a set of rewriting rules called production rules of the form:
  $\omega_1 \cdot A\omega_2 \rightarrow \omega_3 \cdot \omega_4 \qquad A \in N$

$\omega_1, \omega_2, \omega_3$, and ω_4 are elements $(N \cup T)*$. The symbol A is called the subject of the production rule $\omega_1 \cdot A\omega_2 \rightarrow \omega_3 \cdot \omega_4$. $\omega_1 \cdot A\omega_2$ is called the left-hand side of the production rules while $\omega_3 \cdot \omega_4$ represents the right-hand side.

3.3 Sentences and languages for dotted grammars

3.3.1 Definition

A sentence x for a *DG* G is an element of $T*$ for which there exists a finite sequence ω_1 , ω_2, ... ω_n (n > 1) of dotted sentential forms. A dotted sentential form of a *DG* is

an element of $(N \cup T) * \cdot (N \cup T) *$.

$\omega_1 = \cdot S$

$\omega_i = B_i C_i \cdot D_i E_i F_i$

$\omega_n = x \cdot$

For $\omega_{i+1}$the following holds:

1. $D_i \in N$ then $\omega_{i+1} = B_i I_i \cdot J_i F_i$ and $p_i = C_i \cdot D_i E_i \rightarrow I_i \cdot J_i$ is a production rule where $(1 \leq i \leq n-1)$.
2. $D_i \in T$ then $w_{i+1} = B_i C_i D_i \cdot E_i F_i$where $(1 \leq i \leq n-1)$.

The set of sentences which is generated by a dotted grammar G is called the language defined by G. The language defined by G is denoted by $L(G)$. This definition for sentences of dotted grammars corresponds directly to the definition of sentences as these are generated by phrase structure grammars. In the case of dotted grammars however rewriting is restricted to the location in sentential forms specified by the dot. As a result it holds that for every dotted grammar G there exists a phrase structure grammar G' such that $L(G)$ is a subset of $L(G')$. The grammar G' is obtained by removing the dot from the production rules of G.

3.3.2 More terminology

Since dotted grammars are special case phrase structure grammars the terminology of phrase structure grammars is applicable to dotted grammars.

Let r = $\omega_1 \cdot A \omega_2 \rightarrow \omega_3 \cdot \omega_4$ be a production rule of a dotted grammar, r is called a *length decreasing* production rule if $|\omega_1 \omega_2| > |\omega_3 \omega_4|$, r is called *length increasing* if $|\omega_1 \omega_2| < |\omega_3 \omega_4|$. r is called *length preserving* otherwise.

A dotted grammar $G = (N, T, S, P)$ is said to be deterministic under a rule selection strategy f when f defines a one to one function from the set of sentential forms of G to the set of production rules P of G. Clearly a deterministic dotted grammar G allows one and only one derivation for all sentences in $L(G)$, so deterministic dotted grammars don not generate ambiguous sentences.

3.4 An example

The language $a^n b^n c^n (n > 0)$

Consider a DG for the language $a^n b^n c^n (n > 0)$. Capital letters are nonterminals, other letters are terminals and S is the start symbol.

$\cdot S \rightarrow a \cdot SBC$

$\cdot S \rightarrow a \cdot BC$

$\cdot CB \rightarrow BC \cdot$

$BC \cdot C \rightarrow \cdot BCC$

$BC \cdot B \rightarrow \cdot BBC$

$a \cdot B \rightarrow ab \cdot$

$b \cdot B \rightarrow bb\cdot$

b·C →cc·

c·C →cc·

Using this grammar the sentence aaaabbbbcccc is generated.

sentential form	production tule
·S	$\cdot S \rightarrow a \cdot SBC$
a·SBC	$\cdot S \rightarrow a \cdot SBC$
aa·SBCBC	$\cdot S \rightarrow a \cdot SBC$
aaa·SBCBCBC	$\cdot S \rightarrow a \cdot BC$
aaaa·BCBCBCBC	$a \cdot B \rightarrow ab\cdot$
aaaab·CBCBCBC	$\cdot CB \rightarrow BC \cdot$
aaaabBC·CBCBC	$\cdot CB \rightarrow BC\cdot$
aaaabBCBC·CBC	$\cdot CB \rightarrow BC \cdot$
aaaabBCBCBC·C	$BC \cdot C \rightarrow \cdot BCC$
aaaabBCBC·BCC	$BC \cdot B \rightarrow \cdot BBC$
aaaabBC·BBCCC	$BC \cdot B \rightarrow \cdot BBC$
aaaab·BBCBCCC	$b \cdot B \rightarrow bb\cdot$
aaaabb·BCBCCC	$b \cdot B \rightarrow bb\cdot$
aaaabbb·CBCCC	$\cdot CB \rightarrow BC\cdot$
aaaabbbBC·CCC	$BC \cdot C \rightarrow \cdot BCC$
aaaabbb·BCCCC	$b \cdot B \rightarrow bb\cdot$
aaaabbbb·CCCC	$b \cdot C \rightarrow bc\cdot$
aaaabbbbc·CCC	$c \cdot C \rightarrow cc\cdot$
aaaabbbbcc·CC	$c \cdot C \rightarrow cc\cdot$
aaaabbbbccc·C	$c \cdot C \rightarrow cc\cdot$
aaaabbbbcccc·	-

3.5 Parsing with dotted grammars

When one considers designing efficient parsers for non context free phrase structure grammars it is important to note that such a parser faces at least the following problems while parsing an input string:

1. deciding at which point in the current sentential form rewriting should take place
2. selecting a “proper” production rule with which to rewrite the current sentential form

3.5.1 Deciding where to rewrite

The main idea in the design of dotted grammars is to make the position in sentential forms at which rewriting should take place explicit. So we tell the parser at which point in the sentential form to rewrite so that the parser does not have to decide. Now the only problem the parser has to solve is the problem of selecting a production rule with which to rewrite the current sentential form.

Notice that in the example in paragraph 3.4 the dot identifies the position at which rewriting takes place? This is what we gain by adding a dot to the production rules. The remaining problem the parser should solve is the selection of a production rule with which to rewrite the current sentential form. We want this to be a constant time operation relative to the length of sentential forms, as to make efficient parsing of dotted grammars possible.

In conventional table driven parsers (like LL(1) and LR(1) parsers) the problem of deciding where to rewrite is solved by choosing to use left most derivations and right most derivations in reverse respectively. This choice allows LL(1) and LR(1) parsers to "know" at which point in a sentential form rewriting should take place.

Chapter 4

Left to right parsers for deterministic dotted grammars

4.1 Introduction to deterministic dotted grammars

Let $G = (N, T, S, P)$ be a dotted grammar. The production rules P of G may be partitioned into sets of production rules which have a common left-hand side. Such a set of production rules may be written as:

C·DE $\rightarrow$ I_1 ·J_1

C·DE $\rightarrow$ I_2·J_2

...

C·DE $\rightarrow$ I_k·J_k

The set of strings I_1·J_1, I_2·J_2, ... , I_k·J_k will be called the right-hand sides of C·DE denoted by RHS(C·DE). For each right-hand side $r \in RHS(C \cdot DE)$ we define a set of terminal symbols t called the selection set of r. The selection set of a right-hand side r is denoted by SelectionSet(r). When the selection sets for all left-hand sides in a dotted grammar G are disjoint we call G a *deterministic dotted grammar*.

4.2 Defining left to right parsers for deterministic dotted grammars

A left to right parser for a dotted grammars is a device which allows sentences to be derived in the following manner. A sentence x for a dotted grammar G is produced by a finite sequence $\omega_1, \omega_2, ..., \omega_n (n > 1)$ of detailed sentential forms. A detailed sentential form is a triple (X, Y, Z) where X and Z are elements of $T*$ and Y is an element $((N \cup T) * \cdot (N \cup T)*)$.

$\omega_1 = (, \cdot S, x)$

$\omega_i = (A_i, B_i C_i \cdot D_i E_i F_i, L_i G_i)$ L_i is a terminal.

$\omega_n = (x, y\cdot,)$
y is an element of $(N \cup T)*$, called a translation of x.

The symbol L_i is called the lookahead symbol. For ω_{i+1}one of the following holds:

1. The prediction step: $\omega_{i+1} = (A_i, B_i I_i \cdot J_i F_i, L_i G_i$ where $D_i \in N$ and $C_i \cdot D_i E_i \rightarrow I_i \cdot J_i$ is a production rule, and $L_i \in SelectionSet(I_i \cdot J_i)$.

2. The match step: $\omega_{i+1} = (A_i L_i, B_i C_i L_i \cdot E_i F_i, G_i)$ where $D_i \in T$ and $D_i = L_i$.

It will be possible to define different types of parsers based on the method used to generate selection sets for right-hand sides.

4.3 Parsing tables for two deterministic dotted grammars

It is well known that efficient table driven parsers can be defined for LL(1) and LR(1) grammars. This subsection gives example parsing tables for two deterministic dotted grammars.

4.3.1 A parsing table for $a^n b^n c^n (n > 0)$

A deterministic dotted grammar for the language $a^n b^n c^n (n > 0)$ follows.

·S → ·aS
a·S →a·aSBC
a·S →a·BC
BC·B →·BBC
C·CB →CBC·
BC·C →·BCC
b·CB →bBC·
a·B →a·b
b·B →b·b
b·C →b·c
c·C →c·c

This grammar allows the mechanical construction of the following parsing table.

left hand side/terminals	a	b	c
·S	·aS	-	-
a·S	a·aSBC	a·BC	-
C·CB	CBC·	CBC·	CBC·
BC·C	·BCC	·BCC	·BCC
BC·B	·BBC	·BBC	·BBC
b.CB	bBC.	bBC.	bBC.
a·B	a·b	a·b	a·b
b·B	b·b	b·b	b·b
b·C	b·c	b·c	b·c
c·C	c·c	c·c	c·c

4.3.2 A simple expression grammar

In this section a parsing table for the following expression grammar is presented. In the following grammar the identifier NUM is a terminal and other identifiers are non-terminals. The identifier expr is the start symbol.

·expr → ·term _expr

term·_expr →expr ·+ term _expr

term·_expr → expr ·

expr + term·_expr →term·_expr

·term →·factor _term

factor·_term →term ·* factor _term

factor·_term →term ·

term * factor·_term →factor·_term

·factor → ·NUM factor

NUM ·factor →factor ·

This grammar allows the mechanical construction of the following parsing table.

left hand sides/terminals	+	*	NUM
·*expr*	·term _expr	·term_expr	·term_expr
term·_expr	expr·+term _expr	expr·	expr·
expr+term·_expr	term·_expr	term·_expr	term·_expr
·term	·factor _term	·factor _term	·factor _term
factor·_term	term·	term·*factor _term	term·
term*factor ·_term	factor·_term	factor·_term	factor·_term
·factor	-	-	·NUM factor
NUM·factor	factor·	factor·	factor·

Chapter 5

Recording grammars

5.1 Introduction

This section defines the notion of a recording grammars. The machine model used by recording grammars is based on four stacks. Recording grammars are based on the idea of recording what has to be done in the future based on that which has been done in the past and "executing" or "playing back" the actions recorded at future times.
As will be illustrated with a simple example recording grammar, these grammars seem to reduce the number of rules needed to define a language as compared to dotted grammars.

5.2 Definition

A recording grammar (RG) is defined as a dotted grammar to which a special nonterminal symbol called a playback symbol is added. More formally a DG is defined as a 4 tuple (N, T, S, P)

- N is a non empty finite set of nonterminal symbols including a special symbol called the playback symbol denoted by #.
- T is a non empty finite set of terminal symbols
- S element of N is a distinguished symbol called the start symbol
- P is a set of rewriting rules called production rules of the form:
  $\omega_1 \cdot A\omega_2 \rightarrow \omega_3 \cdot \omega_4 \rightarrow \omega_5 \cdot \omega_6 \qquad A \varepsilon N, A \neq \#$

$\omega_1, \omega_2, \omega_3, \omega_4, \omega_5$, and $\omega_6 \in (N \cup T)$ * while w_1 do not ω_2 contain the playback symbol #. The symbol A is called the subject of the production rule $\omega_1 \cdot A\omega_2 \rightarrow \omega_3 \cdot \omega_4 \rightarrow \omega_5 \cdot \omega_6$. $\omega_1 \cdot A\omega_2$ is called the left-hand side of the production rules while $\omega_3 \cdot \omega_4$represents the right-hand side and $\omega_5 \cdot \omega_6$ represents the statement to be recorded.

5.3 Sentences and languages for recording grammars

5.3.1 Definition

A sentence w for a RG G is an element of $T*$ for which there exists a finite sequence ω_1 , ω_2, ... ω_n (n > 1) of recording sentential forms. A recording sentential form is a 4 tuple (W,X,Y,Z) where W and Z are elements of $T*$ while X and Y are elements of $((N \cup T) * \cdot (N \cup T)*)$.

$\omega_1 = (,\cdot S,\cdot,w)$

$\omega_i = (A_i, B_iC_i \cdot D_iE_iF_i, G_i \cdot H_i, I_i)$

$\omega_n = (w, x\cdot, y \cdot z,)$
x,y and z is are element of $(N \cup T)*$, called a translation of w.

For ω_{i+1}one of the following holds:

1. The prediction step: $\omega_{i+1} = (A_{i,}B_iJ_i \cdot K_iF_i, G_iL_i \cdot M_iH_i.I_i)$ where $D_i \in N, D_i \neq$# and $C_i \cdot D_iE_i \rightarrow J_i \cdot K_i \rightarrow L_i \cdot M_i$ is a production rule.
2. The playback instruction: $\omega_{i+1} = (A_i, G_i \cdot H_i, B_iC_i \cdot E_iF_i, I_i)$ where D_i=#.
3. The match step:$\omega_{i+1} = (A_iD_i, B_iC_iD_i \cdot E_iF_i, G_i \cdot H_i, J_i)$where $D_i \in T, I_i = D_iJ_i$.

As usual, the set of sentences which is generated by the recording grammar G is called the language defined by G. The language defined by G is denoted by $L(G)$.

5.4 An example recording grammar

Consider a recording grammar for the language $a^nb^nc^n(n > 0)$. Capital letters are nonterminals, other letters are terminals and S is the start symbol.

$\cdot S \rightarrow \cdot aS \rightarrow \cdot B\#$

$a \cdot S \rightarrow \cdot aS \rightarrow \cdot B$

$a \cdot S \rightarrow \cdot\# \rightarrow \cdot$

$\cdot B \rightarrow \cdot b \rightarrow \cdot c$

Using this grammar the sentence aaabbbccc is produced.

recording sentential form	production tule
(,·S,·, aaabbbccc)	·S → ·aS → ·B#
(,·aS,·B#,aaabbbccc)	accept a
(a,a·S,·B#,aabbbccc)	a.S →.aS → .B
(a,.aS,.BB#,aabbbccc)	accept a
(aa,a·S,·BB#,abbbccc)	a·S →·aS → ·B
(aa,·aS,·BBB#,abbbccc)	accept a
(aaa,a·S,·BBB#,bbbccc)	a·S →·# → ·
(aaa,·#,·BBB#,bbbccc)	playback
(aaa,·BBB#,·,bbbccc)	·B →·b → ·c
(aaa,·bBB#,·c,bbbccc)	accept b
(aaab,b·BB#,·c,bbccc)	·B →·b → ·c
(aaab,b·bB#,·cc,bbccc)	accept b
(aaabb,bb·B#,·cc,bccc)	·B →·b → ·c
(aaabb,bb·b#,·cc,bccc)	accept b
(aaabbb,bbb·#,·ccc,ccc)	playback
(aaabbb,·ccc,bbb·,ccc)	accept c
(aaabbbc,c·cc,bbb·,cc)	accept c
(aaabbbcc,cc·c,bbb·,c)	accept c
(aaabbbccc,ccc·,bbb·,)	-

5.5 A parse table for the example recording dotted grammar

A parse table for the recording grammar given above is shown here.

left hand side / terminals	a	b	c
·S	·aS → ·B#	-	-
a·S	·aS→·B	·# →·	·# →·
.B	-	·b →·c	-

This parse table guides the acceptance of the language intended with less production rules and smaller parse tables.

5.6 Some comments

What I like about the recording grammar in the example above is the fact that it accepts the language intended with significantly less production rules as compared to dotted grammars and plain vanilla Chomsky grammars. I hope that the power of "recording" and "playing back" will serve to simplify dotted grammars in general. Since I view grammars as a formal model for programming languages I hope that similar features in modern programming languages will prove to be an effective means to combat complexity in software systems. I also hope to find out more about the formal properties of these grammars. Please do not hesitate to help.

Chapter 6

How do we construct parsing tables for deterministic dotted grammars?

6.1 Introduction

My reading of the principle of procrastination is:

> Postpone until later that which you might never have to do at all.

To this I add:

> Do now that which you are absolutely certain you must do now.

Put another way, we postpone in those cases we do not know for certain what the most appropriate action is and we act swiftly in those cases we do know what the most appropriate action is. Using these principles a mechanism will be devised which allows the mechanical construction of parsers for deterministic dotted grammars.

In section 6.2 I first try to informally describe the process of generating parse tables for deterministic dotted grammars. In section 6.3 I will attempt a more formal (and hopefully accurate and readable) formulation of the same process.

6.2 Generating a parse table for the language $a^n b^n c^n (n > 0)$; step by step.

The process of generating parse tables for deterministic dotted grammars will be presented here using the language $a^n b^n c^n (n > 0)$ as an example. The process detailed for generating the parse tables is essentially the process used by the TINY parser generator introduced later in this paper.

1. Compute the initial terminal set for right-hand sides
2. selectively associate (or connect) the right-hand sides with left-hand sides
3. Compute the subsequent terminal sets and for left hands ides

4. Compute the selection sets for the right-hand sides.

The deterministic dotted grammar for the language $a^n b^n c^n (n > 0)$ is repeated here for reference:

·S → ·aS
a·S →a·aSBC
a·S →a·BC
BC·B →·BBC
C·CB →CBC·
BC·C →·BCC
b·CB →bBC·
a·B →a·b
b·B →b·b
b·C →c·c
c·C →c·c

6.2.1 Compute the initial terminal set for a right-hand side

First we designate that each left-hand side of a production rule is associated with a set of terminal symbols called the *terminal set* of the left-hand side. We also designate that each right-hand side of a production rule is associated with a set of terminal symbols called the *terminal set* of the right-hand side. If a right-hand side has a terminal symbol to right of the dot then this terminal symbol is an element of the terminal set for this right-hand side. Such a right-hand side is called a *leaf*. For right-hand sides which are not leaves the terminal sets are initially empty. The initial terminal sets for right-hand sides with non empty terminal sets follows.

left hand side	initial terminal set	righthandside
$\cdot S$	$\{a\}$	$\cdot aS$
$a \cdot S$	$\{a\}$	$a \cdot aSBC$
$a \cdot B$	$\{b\}$	$a \cdot b$
$b \cdot B$	$\{b\}$	$b \cdot b$
$c \cdot C$	$\{c\}$	$c \cdot c$
$b \cdot C$	$\{c\}$	$b \cdot c$

6.2.2 Connecting the right-hand sides

The essential point to note in this context is that the left-hand side of a production rule may be a substring of the right-hand side of a production rule. Let us note for example that the left-hand side of the production rule: a·B →a·b is a substring of the right-hand side of the production rule a·S →a·BC. From this it follows that whenever the string a·BC is substituted into a sentential form, it is possible that the rule a·B →a·b may be subsequently applied. However, in general it may be true for a given grammar that there exists more than one left-hand side of a production rule which is a substring of a given right-hand side of a production rule. In this case the right-hand side is connected to the longest left-hand side which is a substring of the particular right-hand side. If

there exist more than one "longest" left-hand side then a left-hand side is chosen based one its relative position in the grammar definition. The connected right-hand sides for our example grammar are:

production rule	connected left hand side
a·S →a·aSBC	a·S
a·S →a·BC	a·B

Before it is verified if a left-hand side is a substring of a right-hand size the dot is migrated to the right until there is not a terminal symbol to the right of the dot. By this rule the right-hand side of the production rule a·S →a·aSBC connects to its own left-hand side in our example grammar.

6.2.3 Computing subsequent terminal sets

Using the following procedure, the terminals sets for all all right hands sides and left-hand sides in the grammar is computed.

After applying the algorithm above on our example grammar, the terminal sets for our right-hand sides become are

left-hand side	terminal set	right-hand side
$\cdot S$	$\{a\}$	$\cdot aS$
$a \cdot S$	$\{a\}$	$a \cdot aSBC$
$a \cdot S$	$\{b\}$	$a \cdot BC$
$a \cdot B$	$\{b\}$	$a \cdot b$
$b \cdot B$	$\{b\}$	$b \cdot b$
$c \cdot C$	$\{c\}$	$c \cdot c$
$b \cdot C$	$\{c\}$	$b \cdot c$

The terminal set of a left-hand side is defined as the union of the terminal sets of the non leaf right-hand sides.

6.2.4 Compute the selection sets for the right-hand sides

After applying the procedure described in section 6.2.3 the terminal set of a right-hand side is called the selection set of that right-hand side. For deterministic dotted grammars it holds that all selection sets for a particular left-hand side must be disjoint. If, for a particular left-hand side, a selection set is the empty set then the corresponding right-hand side is designated the *default* right-hand side for this left-hand side. The default right hand is applied when no other right hand can be applied. If for a particular left-hand side more than one selection set is empty (or if the selection sets of the right-hand sides are not disjoint) then the grammar is not a deterministic dotted grammar. By the preceding, the selection sets for our example grammar become:

left hand side	selection set	righthandside
$\cdot S$	$\{a\}$	$\cdot aS$
$a \cdot S$	$\{a\}$	$a \cdot aSBC$
$a \cdot S$	$\{b\}$	$a \cdot BC$
$BC \cdot B$	$\{a,b,c\}$	$\cdot BBC$
$a \cdot B$	$\{b\}$	$a \cdot b$
$b \cdot B$	$\{b\}$	$b \cdot b$
$c \cdot C$	$\{c\}$	$c \cdot c$
$C.CB$	$\{a,b,c\}$	$CBC\cdot$
$BC \cdot C$	$\{a,b,c\}$	$\cdot BCC$
$b \cdot CB$	$\{a,b,c\}$	$bBC\cdot$
$b \cdot C$	$\{c\}$	$b \cdot c$

6.3 Selecting a production rule for rewriting

In a previous paragraph the two main problems for a parsers were identified as:

1. deciding at which point in the current sentential form rewriting should take place
2. selecting a "proper" production rule with which to rewrite the current sentential form

We now want to be a bit more specific on the nature of the second problem.

We divide the problem of selecting a production rule into:

1. the selection of a left-hand side which is a substring of the current sentential form
2. choosing a right-hand side with which to rewrite the current sentential form

6.3.1 Selecting a left-hand side

Let $G = (N,T,S,P)$ be a dotted grammar. Let Z be the set of sentential forms of G. Let $L = \{\omega_1 \cdot A\omega_2 | \omega_1 \cdot A\omega_2 \rightarrow \omega_3 \cdot \omega_4 \in P\}$ be the set of left-hand sides of the production rules P.

The left-hand side selection strategy *LSS* for a deterministic dotted grammar is defined as a one to one function from Z to L.

A definition of a left-hand side selection strategy is formulated using the notion of *candidate left-hand sides* for a sentential form $z \in Z$.

For the set C of candidate left-hand sides for a sentential form ω the following conditions must hold:

- The left-hand side of all production rules $c \in C$ are substrings of ω.
- All $c \in C$ have a left-hand side of the same length and no x element of C has a left-hand side of greater length than the left-hand side of c.

From the candidate set for a sentential form the value of $LSS(\omega)$ is deduced as follows:

1. If C is the empty set
   This represents a parser error. $LSS(\omega) = undefined$

2. If C contains but one element c
   $LSS(\omega) = c$.

3. If C contains more than one element
   Define a disambiguation rule (like the order in which the rules are specified) which selects one element c of C as the left-hand side for the current ω. $LSS(\omega) = c$. We will use the order of the production rules as the disambiguation rule.

4. For completeness we define that $LSS(\cdot) = undefined$.

6.3.2 Selecting a right-hand side

The selection set for a right-hand side $\omega_c . \cdot B\omega_d$ of a production rule $\omega_a \cdot A\omega_b \rightarrow \omega_c . \cdot B\omega_d$ of a dotted grammar G is defined as:

- SelectionSet($\omega_c . \cdot B\omega_d$) = $\{B\}$ if $B \in T$.
- SelectionSet($\omega_c . \cdot B\omega_d$) = SelectionSet(LSS($\omega_c . \cdot B\omega_d$)) if $B \in N$and LSS($\omega_c . \cdot B\omega_d$) is defined.
- If LSS($\omega_c \cdot B\omega_d$) is undefined and $\omega_c \cdot B\omega_d \neq .$ then we consider the following: If the left-hand side $\omega_a \cdot A\omega_b$ has more than one right-hand side for which LSS($\omega_c \cdot B\omega_d$) is undefined then SelectionSet($\omega_c \cdot B\omega_d$) is undefined. When $\omega_a \cdot A\omega_b \rightarrow \omega_c \cdot B\omega_d$ is the only production rule for which LSS($\omega_c \cdot B\omega_d$) is undefined we call $\omega_a \cdot A\omega_b \rightarrow \omega_c \cdot B\omega_d$ the *default production rule* and we define SelectionSet($\omega_c \cdot B\omega_d$)as all $t \in T$ such that t is not an element of SelectionSet(r_i) $(1 \leq i \leq k)$and the set of right-hand sides of $\omega_a \cdot A\omega_b$ is defined as: $\{r_1, r_2, ..., r_k\}$.

Chapter 7

LL(1) grammars and LR(1) grammars as subsets of deterministic dotted grammars

7.1 Introduction

Now we'll show that LL(1) and LR(1) grammars are special case deterministic dotted grammars. Since all the properties of these grammar classes are contained within the parse table for these grammar classes, we only have to show that deterministic dotted grammars can mimic all derivations allowed by LL(1) and LR(1) parse tables respectively.

7.2 LL(1) grammars as deterministic dotted grammars

This section will show that LL(1) grammars, are deterministic dotted grammars. An LL(1) grammar G = (N, T, S, P) is a context free grammar for which an LL(1) parse table can be constructed. An LL(1) parse table M is a function from N $\times$T to P. A sketch of the LL(1) parsing algorithm follow (taken from ref 16).

Input: A string ω and a $LL(1)$ parsing table M for a grammar G.

Output: true if $\omega \in L(G)$ and false otherwise .

Method: Initially the contents of the parser stack is $\$S$ with S on the top of the stack. \$ is a distinguished end of input marker. The input string is of the form $\omega\$$. The LL(1) parsing algorithm operates as follows :

set ip to point to the first symbol of $\omega\$$.
repeat
Let x be the top stack symbol and a the symbol pointed to by ip;
    if x is a terminal or \$ then
        if $x = a$ then
            pop x from the stack and advance ip

```
        else
            return false;
    else
        if M(x,a) = x → y1,y2,...,yk    then
        begin
            pop x   from the stack
            push yk,yk−1,...,y1 so that y1          is the symbol at the top of the stack
        end
        else
            return false;
until x = $
```

A deterministic dotted grammar which allows the same derivations as LL(1) grammars is easily constructed as follows:

When $A \rightarrow \beta$ is a production rule for an LL(1) grammar, then $\cdot A \rightarrow \cdot\beta$ is the corresponding production rule for an equivalent deterministic dotted grammar. if $M(A,a) = X \rightarrow \beta$ then $a \in SelectionSet(\beta)$ for the left-hand side $\cdot A$. By inspection it is clear that dotted grammar defined is this manner will generate the same derivation sequences as the corresponding LL(1) grammar.

7.3 LR(1) grammars as deterministic dotted grammars

In this section it will be sketched how deterministic dotted grammars can be constructed to generate the same derivations as *LR* grammars. A context free grammar $G = (N,T,S,P)$ is an *LR* grammar when it is possible to construct an *LR* parsing table for G. An *LR* parsing table is actually two functions called the action and goto functions respectively. The action function maps a special nonterminal called a state symbol s_i and a terminal symbol a_j to one of the following for values.

action(s_i, a_j) =

1. shift s_i, where s_i is a state
2. reduce by a production rule $A \rightarrow \beta$
3. accept
4. error

The goto function maps a state and a nonterminal to a state symbol. Sentential forms of LR parsers take the form: $s_o, X_1, s_1, X_2, s_2, ..., X_i, s_i$ where s_x represents a state and $X_y$represents a grammar symbol. The LR parsing algorithm may be summarized by the following algorithm. (taken from ref. 16)

Input: A string ω and action and goto functions for a *LR* grammar G

Output: true if $\omega \in L(G)$ and false otherwise.

Method: Initially the contents of the parser stack is the initial state s_0. The input string is of the form $\omega\$$. \$ is a distinguished end of input marker. The $LR(1)$ parsing algorithm operates as follows.

set ip to point to the first symbol of $\omega\$$.
repeat
Let s be the state on top of the stack and a the symbol pointed to by ip
if action(s,a) = shift s' then
begin
push a
push s'
advance ip
end else if $action(s,a) = reduce A \rightarrow \beta$ then
begin
pop $2 * |\beta|$ symbolsof the stack
s is the state symbol uncovered after the previous step
push A push $goto(s',A)$
end
else if $action(s,a) = accept$
return $true$;
else
return $false$;
until $true = false$;

A deterministic dotted grammar G = $(N \cup N' \cup E \cup E', T, S, P')$ can be designed to mimic the derivations of a *LR* grammars G' = $G' = (N, T, s_0, P)$. N' represents the set of state symbols used in LR parsers, E and E' are nonterminals used for erasing (or popping) symbols and P' represents the productions of G used to mimic derivations of *LR* grammars. $s_0$is the start symbol of G' is corresponds with the initial state of the *LR* parser.

- If $action(s,a) = shift(s')$ then add the following production rule to P' :$\cdot s \rightarrow .as'$ where $a \in SelectionSet(\cdot as)$
- If $action(s,a)$ is reduce $A \rightarrow \beta$ then add the following set production rules to P': $\cdot s \rightarrow s.E^{\{|\beta|\}}E's'A$ and $\cdot s'A \rightarrow A.s''$
  where $a \in SelectionSet(s.E^{\{|\beta|\}}E's'\mathrm{A})$; $s'' = goto(s'A)$. The symbols E and E' are special nonterminals called the eraser symbols. The production rules for the symbol E have the form:
  $\cdot EE' \rightarrow \cdot$
  together with the following set of production rules:
  $n_1 s_1 \cdot E \rightarrow \cdot$
  $n_1 s_2 \cdot E \rightarrow \cdot$
  ...
  $n_1 s_i \cdot E \rightarrow \cdot$
  $n_2 s_1 \cdot E \rightarrow \cdot$
  $n_2 s_2 \cdot E \rightarrow \cdot$
  ...
  $n_2 s_i \cdot E \rightarrow \cdot$
  ...
  $n_j s_1 \cdot E \rightarrow \cdot$
  $n_j s_2 \cdot E \rightarrow \cdot$
  ...
  $n_j s_i \cdot E \rightarrow \cdot$ where $n_x \in N and s_y \in N'$.

- If $action(s,a) = accept$ then add the following rule to P':
  $\cdot s \rightarrow \cdot$.
- If $action(s,a) = error$ then add no production rules to P'.

Chapter 8

The TINY parser generator

8.1 Introduction

The TINY parser generator (Tiny Is Not Yacc) is a straightforward implementation of the idea of a parser generator for dotted grammars. (This implementation does not include recording grammars in the current version.) The TINY is implemented in C++ and generates table driven parsers in C++. In this section this program will be introduced by means of two grammars.

8.2 Language definition files

The general structure of a language definition file follows:

```
grammar <grammar name>
{
terminals
// A list of terminal symbols
nonterminals
// a list of nonterminal symbols
start <nonterminal>
error <nonterminal>
type <C++ type name>
principles
// a set of dotted production rules
}
```

Each grammar has a grammar name denoted by <grammar name> in the above. The TINY generates a C++ class with this name which implements the grammars. Each grammar designates one nonterminal as its start symbol. The nonterminal designated as the exception symbol is made to be the current symbol (the symbol to the right of the dot) when any syntax error occurs. Using this symbol one can define how the parser acts on unexpected input. The TINY refers to production rules a principles.

C++ comments are also viewed as comments in interaction definition files.

A syntax error can be detected in one of the following situations.

1. A terminal symbol is the symbol to the right of the dot in a sentential form while this symbol is not the lookahead symbol.

2. There exists no left-hand side which is a substring of the current sentential form.

3. There exists a left-hand side which is a substring of the current sentential form however none of the right-hand sides of this left-hand side is applicable.

8.3 An Example

This example introduces a grammar which when submitted to the TINY will generate a C++ program which will accept the language $a^n b^n c^n (n > 0)$. Parsers generated by the TINY consume an input symbol whenever a terminal symbol is to the right of the dot.

```
// A dotted grammar for AnBnCn
grammar AnBnCn
{
terminals
          a b c
nonterminals
        S B C SyntaxError
exception SyntaxError
start S
type int
principles
        .S : .a S
            ;
        a.S :   a. a S B C
            |   a. B C
            ;
        a.B : a .b      ;
        b.B : b .b      ;
        B C.B : .B B C  ;
        C .C B : C B C. ;
        B C.C : .B C C  ;
        b.C B : b B C.  ;
        c.C : c .c      ;
        b.C : b .c      ;
        .SyntaxError : . {% cout << "Syntax error" << endl; %};
}
source
{%
void AnBnCn::getToken(TINY_Symbol<int> &s)
{
        char c;
        cin >> c;
        s.attr = 0;
        if (c == 'a' || c == 'A')
                s.id = AnBnCn::a;
```

```
        else if (c == 'b' || c == 'B')
                s.id = AnBnCn::b;
        else if (c == 'c' || c == 'C')
                s.id = AnBnCn::c;
        else
                s.id = 0; // an error
}
%}
```

If the above example is placed in the file "anbncn.t" the following command will yield the C++ source and header files which correspond with this example.

```
tiny anbncn.t
```

By adding a main program defined as follows a complete working parser is obtained.

```
#include <iostream.h>
#include "anbncn.t.h"

void main()
{
        AnBnCn parser;
        parser.parse(0);
}
```

8.4 Semantic actions

The control program generator allows production rules to be annotated by semantic actions.

Three types of semantic rules are distinguished in the TINY.

1. Left-hand side selection rules
2. Right-hand side selection rules
3. post rewriting rules

A production rule A.B.C : E.F ; may be annotated by semantic actions as in the following:

```
A.BC {% expr1 %} : {% expr2 %} E.F{% statements %} ;
```

expr1: the left-hand side selection rule; This expression determines when the parser will rewrite with a the production rules. When the left-hand side selection rule is omitted the default left-hand side selection strategy is used.

expr2: the right-hand side selection rule. This expression determines when a right-hand side is selected. When this expression is omitted the right-hand side selection strategy (base on selection sets) is used.

statements: the post rewriting rules. These rules allow sematic actions to be performed after rewriting has taken place.

In the TINY these semantic actions are blocks of C++ code.

Chapter 9

Simulating Turing machines with deterministic dotted grammars

9.1 Introduction

In this chapter we'll show how to simulate arbitrary Turing machines with deterministic dotted grammars. We'll see that the dot does not restrict the languages generated by dotted grammars.

9.2 Recursively enumerable languages

The purpose of this section is to show that the dot in production rules of dotted grammars does not restrict the languages generated by dotted grammars. To show that the set of recursively enumerable languages are generated by dotted grammars this section shows how to simulate Turing machines using these grammars.

Turing programs are represented as a finite non empty set of Turing instructions of the form: $(Q_o, S_o, Q_n, Q_n, S_n, d)$.

- Q_o represents the current Turing machine state
- S_o represents the symbol at the current tape position
- Q_n represents the next machine state
- S_{n} represents the next symbol for the current tape location
- $d \in \{L, R, O\}$ represents the direction of movement for the head of the Turing machine.

For the Turing machine M the following holds.

1. M has i states $Q_1, ..., Q_i$
2. M has j alphabet symbols $S_1, ..., S_j$

3. The blank symbol is represented by B
4. The initial string on the tape is ω

For G the following holds:

1. $T = \{B, s_1, s_2, ..., s_j\}$ where $s_1, ..., s_j$ correspond to the symbols $S_1, ..., S_j$ of M.
2. $\$N = \{s, q_1, q_2, ..., q_1\}$ where $q_1, ..., q_i$ correspond to the states $Q_1, ..., Q_i$ of M.
3. $S = s$
4. The productions rules of G are generated by the following procedure:

   - Add the production rule $\cdot s \rightarrow Bx \cdot Q_1 yB$, where Q_1 is the start state of M and x is a tape symbol and y is one or more Turing symbols such that $\omega = xy$.
     In the following s_x represents the current tape symbol. The new tape symbol is represented by s_y.
     The current and next Turing machine state are represented by q_x and q_y respectively. The symbol to the right of the current tape symbol is represented by t.
   - For each Turing machine instruction (Q_x, S_x, Q_y, S_y, R) which moves the head to the right we add a production rule of the form: $s_x \cdot q_x t \rightarrow s_y t \cdot q_y$ where Q_y is not a final state of M and t is not the B. For a Turing machine instruction (Q_x, B, Q_y, S_y, R) we add the production of the form: $B \cdot q_x \rightarrow s_y B \cdot q_y B$ instead of the production rule specified above.
   - For each Turing machine instruction (Q_x, S_x, Q_y, S_y, L) which moves the head to the left we add a production rule of the form: $s_x \cdot q_x \rightarrow \cdot q_y s_y$ where Q_y is not a final state of M. For a Turing machine instruction (Q_x, B, Q_y, S_y, L) we add the production of the form: $B \cdot q_x \rightarrow B \cdot q_y s_y$ instead of the production rule specified above.
   - For each Turing machine instruction $Q_x, S_x, Q_y, S_y 0)$ which doesn't move the head we add a production rule of the form : $s_x \cdot q_x \rightarrow s_y \cdot q_y$where q_y is not a final state of M.
   - For each Turing machine instruction (Q_x, S_x, Q_y, S_y, d) where Q_y is a final state of M we add a production of the form: $s_x \cdot q_x \rightarrow s_y \cdot$.
   - No more production rules are added to G.

The existence of an equivalent dotted grammar for any Turing machine program provides the necesary evidence to support that dotted grammars generate the set of recursively enumerable languages.

By inspecting the production rules of the dotted grammars allows us to note that the production rules used to simulate Turing machines have the form:

1. $\cdot s \rightarrow Bx \cdot Q_1 yB$
2. $s_x \cdot q_x t \rightarrow s_y t \cdot q_y$
3. $B \cdot q_x \rightarrow s_y \cdot B q_y B$
4. $s_x \cdot q_x \rightarrow \cdot q_y s_y$

5. $B \cdot q_x \rightarrow B \cdot q_y s_y$

6. $s_x \cdot q_x \rightarrow s_y \cdot$These rules are only used on transitions into final states.

7. $s_x \cdot q_x \rightarrow s_y \cdot q_y$

The meaning of the symbols used in the rules above is given in the section where the Turing machines are simulated by dotted grammars. Dotted grammars having production rules of the above forms are said to be in the Turing normal form. At this point we take time to note that no length decreasing production rules (with the sole exception of transitions into a final state) are used in the mapping of Turing instructions to the instructions of dotted grammars.

We also note that the dotted production rules which simulate Turing machines are suffixed and prefixed by a string of blanks. No length reducing production rules (outside of transitions into a final state) are needed because blank symbols are used to indicate which parts of sentential forms are not a part of the sentence accepted.

By removing the dot from the production rules of a dotted grammar G in the Turing normal form, a phrase structure grammar G' is obtained which also generates $L(G)$. This becomes evident when we consider that sentential forms of grammars in the Turing normal form only contain one nonterminal. This single nonterminal must be rewritten to obtain the following sentential form.
Since this single nonterminal represents the state of the Turing machines (which always appears to the right of the dot) phrase structure grammars and dotted grammars in the Turing normal form generate exactly the same derivations.

Chapter 10

Multi-dot Grammars

10.1 Introduction

Dotted grammars as introduced in the previous chapters, use a dot in sentential forms of these grammars to identify the location at which rewriting must take place. In this chapter we will allow for more than one dot in sentential form of dotted grammars. The grammars thus obtained we call multi-dot grammars (MDG).

10.2 Definition

A multi-dot grammar is a 6 tuple N,T,R,C,S,P where:

- T is a finite set of terminal symbols
- N is a finite set of nonterminal symbols
- R is a finite set of receptor symbols
- C is a set of compound symbols of the form :
  $[\omega_1 \cdot \omega_2]$
  where ω_1, ω_2 are strings over $(N \cup T \cup R \cup C)*$.
- $S \in N$ is a distinguished nonterminal called the start symbol.
- P is a set of rewriting rules p called production rules of the form:
  $\omega_1 \cdot A\omega_2 \rightarrow \omega_3 \cdot \omega_4$
  where $A \in (N \cup R \cup C)$ while $\omega_1, \omega_2, \omega_3, \omega_4 \in N \cup T \cup R \cup C \cup \{\cdot, [,]\}*$.

An element of the set $N \cup T \cup R \cup C$ will be denoted by Σ. Thus Σ represents the set of all grammar symbols.

A string $AB \cdot CDE$ is said to be reducible by a multi-dot grammar $G = (N,T,R,C,S,P)$ if and only if there exists at least one $p \in P$ of the form: $B \cdot CD \rightarrow F \cdot G$ where A,B,C,D,R,F,G are strings over$(\Sigma \cup \{\cdot, [,], \})*$ and$C \in (N \cup E \cup C)$.

10.3 Rewriting with multi-dot grammars

10.3.1 BNF definition of sentential forms

A sentential form of a multi-dot grammar$G = (N,T,R,C,S,P)$ is defined in this section. Terminal symbols are written in capital letters or between single quotes. The empty string is denoted byε.

sententialForm: symbolList '.' symbolList
symbolList: symbolList symbol
symbolList:ε
symbol:TERMINAL
symbol:NONTERMINAL
symbol: RECEPTOR
symbol:[sententialForm]

10.3.2 The reduce function for multi-dot grammars

Rewriting for a multi-dot grammar$G = (N,T,R,C,S,P)$ will be defined by a function called REDUCE. Let F represent the set of sentential forms of G then the $REDUCE$ function is a function from F to F defined as follows:

1. $REDUCE(ab \cdot cde) = abc \cdot de$ if $c \in T$.

2. $REDUCE(ab \cdot cde) = af \cdot ge$ where$b \cdot cd \rightarrow f \cdot g \in P$ and $c \in (N \cup \mathrm{C})$.
   $REDUCE(ab \cdot cde) = ab \cdot cde$ where $b \cdot cd \rightarrow f \cdot g \notin P$and $c \in (N \cup C)$.

3. $REDUCE(ab \cdot cde) = \lambda < c \cdot cd > [ab \cdot cde](f \cdot g)$where $b \cdot cd \rightarrow f \cdot g \in P$ and $c \cup R$. $REDUCE(ab \cdot cde) = ab \cdot cde$ where $b \cdot cd \rightarrow f \cdot g \notin P$ and $c \in R$. The λ expression $\lambda < b \cdot cd > [ab \cdot cde](f \cdot g)$ is understood as the substitution of $f \cdot g$ for $b \cdot cd$ at all possible locations in the sentential form $ab \cdot cde$.

4. In all other cases $REDUCE(ab \cdot cde) = a'b' \cdot c'd'e)$ where the following correspondence exists between $ab \cdot cde$ and $a'b' \cdot c'd'e'$.
   Let $abcde = x_1,x_2,...,x_k$ where $|abcde| = k$, and each $x_i(1 \leq i \leq k)$. Then $a'b' \cdot c'd'e = x'_1,x'_2,...,x'_k$ where the following holds for x'_i:

   (a) $x'_i = x_i$ if $x_i \in (N \cup T \cup R)$.
   (b) $x'_i = REDUCE(\omega_1 \cdot \omega_2)$ if $x_i = [\omega_1 \cdot \omega_2]$.

Reducing sentential forms as defined under point 2 corresponds to rewriting as defined in Chomsky's phrase structure grammars. Thus rewriting takes place at one location in the sentential form. Reducing sentential forms as defined under point 3 corresponds to rewriting as defined in Church's λ calculus. Thus a string s_1 is replaced by a string s_2 at all location s_1 occurs in the sentential form. Reducing sentential forms as defined under point 4 corresponds to rewriting as defined in Lindenmayer's L-systems. Thus in one rewriting step all possible production rules are applied. A sentential form f is said to be in normal form for a multi-dot grammar G if and only if $REDUCE(f) = f$. A normal form is derived from the start symbol of a multi-dot grammar as a finite sequence $\omega_1, \omega_2, ..., \omega_n$ defined as:

$\omega_1 = .S$

$\omega_i = B_i C_i . D_i E_i F_i$

$\omega_{i+1} = REDUCE(\omega_i)\ \omega_i \neq \omega_{i+1}$

$\omega_n = \omega_{i+1}$ when $REDUCE(\omega_i) = \omega_i = \omega_{i+1}$.

Sentences and languages (as defined for phrase structure grammars) may be viewed as special cases of normal forms as defined for multi-dot grammars.

Chapter 11

Conclusion

This paper introduced the dotted grammar formalisms. Also recording grammars were introduced in this paper. It was shown that a class of deterministic dotted grammars allowed table driven parsers to defined for grammars which are a super set of LL(1) and LR(1) grammars. Deterministic dotted grammars are shown to be able to simulate deterministic Turing machines which implies that deterministic dotted grammars generate the set of recursively enumerable languages.

Bibliography

[1] R. Book [1973] "On the structure of context sensitive grammars". International Journal of Computer Sciences Vol 2, No. 2. pages 129 - 138.

[2] R. Book [1973] edited by A. Aho "Currents in the theory of computing" chapter 1. Prentice-Hall, Inc.

[3] S. Ginsburg, S. A. Greibach [1966] "Mappings which preserve context sensitive languages" Information and Control 9, 563-582 .

[4] D. Grune, C. Jacobs [1990] "Parsing Techniques". Ellis Horwood Limited.

[5] D. Harel [1989] "Algoritmiek" Academic Service.

[6] S.-Y Kuroda [1964] "Classes of Languages and Linear-Bounded Automata", Information and Control 7, 207-223.

[7] J. Loeckx [1970] "Parsing for general phrase structure grammars". Information and control 16, 443-464.

[8] G. Matthews [1964] "A note on asymmetry in phrase structure grammars", Information and control 7, 360-365.

[9] T. A. Sudkamp [1991] "Languages and Machines" Addison-Wesley.

[10] D. Ullman , J. Hopcroft [1969] "Formal languages and their relation to automata". Addison-Wesley.

[11] R. Uzgalis, J. Cleaveland [1977] "Grammars for Programming Languages" Elsevier Computer Science Library.

[12] V.J. Rayward-Smith [1988] "Inleiding in de theorie van formele talen" Academic Service.

[13] Robert Sedgewick [1988] "Algorithms" Addison-Wesley.

[14] S. Dik, J. Kooij, [1991], "Algemene taal wetenschap", Uitgeverij het spectrum

[15] F. McCabe, [1992], "Logic and Objects", Prentice Hall International (UK) Ltd

[16] A. Aho, R.Sethi, J. Ullman, [1986], "COMPILERS Principles, Techniques and tools", Addison-Wesley.

[17] P. Brown, [1981], "Writing Interactive Compilers and Interpreters", John Wiley & Sons

[18] V.J. Rayward-Smith, [1985], "Inleiding in de berekenbaarheidstheorie", Academic Service

[19] J. van Eijck, E. Thijsse, [1989], Logica voor alfa's en informatici,Academic Service

[20] C.A. Koster and various co-authors, [1992], "1992/1992", University of Nijmegen Dept. Informatics (The Netherlands).

[21] Kent Beck, [1994], "Patterns and Software development", Dr. Dobbs Journal

[22] M.T. Rosetta [1994] "ROSETTA Compositional Translation", Kluwer Academic Publishers

[23] Samuel Guttenplan [1986] "The languages of logic", Blackwell Publishers

[24] James Allen [1995] "Natural Language Understanding" The Benjamin/Cummings Publishing Company, Inc.

[25] Arie Sturm en Fred Weerman [1983] "Generatieve Syntaxis", Martinus Nijhoff/Leiden

[26] H. van Riemsdijk and Edwin Williams [1986] "Introduction to the theory of grammar", The Massachusetts Institute of Technology.

[27] J. Glenn Brookshear [1989] "Formal Languages, Automata, and Complexity", The Benjamin/Cummings Publishing Company, Inc.

[28] R. Sommerhalder/S.C. van Westrhenen [1987] "Introduction to the theory of Computability of Programs, machines, effectiveness and feasibility", Technische Universiteit Delft

[29] R. Plasmeijer, M van Eekelen [1993] "Functional Programming and Parallel Graph Rewriting", Addison-Wesley Publishing Company.