

A Quick Introduction to Generalized Deterministic Parsing: Take 1

Maurice Gittens <maurice at gittens dot nl>

6th April 2004

Abstract

This paper presents a first take at introducing Generalized Deterministic Parsing based on a slight modification of Noam Chomsky's phrase structure grammars called dotted grammars. Using the toy language $a^n b^n c^n$ ($n \geq 1$) and the Tiny parser generator a quick introduction to generalized parsing technique will be presented. It is assumed that the reader is familiar with grammars and parsers.

The copyright of this document belongs to its author. Making complete and unmodified copies of this document is allowed.

Status: preliminary draft

Changes

- April 9 2004; Here's an attempt to fulfill a promise... a first almost complete version; Still Todo: proofreading, readability enhancements, general cleaning up...
- March 28 2004; First cut

Parsing context sensitive phrase structure grammars by example

Consider the following phrase structure grammar which generates a set of strings which can be partitioned into substrings of a's, b's and c' of equal length, denoted $a^n b^n c^n$ ($n \geq 1$).

$S \rightarrow aSBC$

$S \rightarrow aBC$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$$cC \rightarrow cc$$

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	$S \rightarrow aSBC$
$aSBC$	$S \rightarrow aSBC$
$aaSBCBC$	$S \rightarrow aSBC$
$aaaSBCBCBC$	$S \rightarrow aBC$
$aaaaBCBCBCBC$	$aB \rightarrow ab$
$aaaabCBCBCBC$	$CB \rightarrow BC$
$aaaabBCCBCBC$	$CB \rightarrow BC$
$aaaabBCBCCBC$	$CB \rightarrow BC$
$aaaabBCBCBCC$	$CB \rightarrow BC$
$aaaabBCBBCCC$	$CB \rightarrow BC$
$aaaabBBBCCC$	$CB \rightarrow BC$
$aaaabBBBCCC$	$bB \rightarrow bb$
$aaaabbBBCCC$	$bB \rightarrow bb$
$aaaabbbBCCC$	$bB \rightarrow bb$
$aaaabbbbCCC$	$bC \rightarrow bc$
$aaaabbbbcCCC$	$cC \rightarrow cc$
$aaaabbbbccCC$	$cC \rightarrow cc$
$aaaabbbbccC$	$cC \rightarrow cc$
$aaaabbbcccc$	

Given a context sensitive grammar such as the presented in the example above it is generally not trivial to deduce which production rule is appropriately applied at specific points in the generation of a the resulting string. Attempting to understand how to choose a production rule appropriately prompted the recognition of the two key principles of generalized deterministic parsing:

- The first priority is: deciding where to rewrite
- Only after having determined where rewriting should take place does the selection of a “proper” production rule with which rewriting should take place become relevant.

So how can one have a parser (or sentence generator) know where it should rewrite? It does not take long to figure out that we might as well tell the parser where rewriting should take place. So we keep it simple and encode the location at which rewriting should take place into the production rules of grammars and try again.

Grammars for Generalized Deterministic Parsing

A new grammar for the language $a^n b^n c^n$ ($n > 0$) becomes:

$$\cdot S \rightarrow a \cdot SBC$$

$\cdot S \rightarrow a \cdot BC$
 $\cdot CB \rightarrow BC \cdot$
 $BC \cdot C \rightarrow \cdot BCC$
 $BC \cdot B \rightarrow \cdot BBC$
 $a \cdot B \rightarrow ab \cdot$
 $b \cdot B \rightarrow bb \cdot$
 $b \cdot C \rightarrow cc \cdot$
 $c \cdot C \rightarrow cc \cdot$

Using this grammar the sentence *aaaabbbbcccc* is generated.

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

Notice that it is relatively easy to derive the string *aaaabbbbcccc* using the dotted grammar from the example above. Because we now know where rewriting should take place parsing with certain dotted grammars becomes similar in complexity to parsing with LL(k) and LR(k) parsers.

A parse table for our example dotted grammars

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

A parse table for the language $a^n b^n c^n (n > 0)$ based on the dotted grammar introduced about is presented here.

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

The columns in this table are identified terminal symbols while the rows are identified by left hand sides of production rules. Each row/column pair defines a string to be substituted into the current sentential form when a the left hand side of a production rule is a substring of the current sentential form and the current look ahead symbol corresponds to the column identifier.

Recording grammars and their parse tables

Generalizing dotted grammars even more brings us to the idea of recording grammars. The idea is that the parser should be able to record segments it must recognize in the future based on the input it has already seen. So that which is to be done in the future is determined by which has happened in the past. At designated points recording grammars allows “executing” or “playing back” actions that were previously recorded. 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.

Consider a recording grammar 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.

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

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

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

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

The production rules for recording grammars are extended to include a substring to be recorded whenever a production rule is applied. Additionally there exists a special nonterminal symbol denoted by $\#$ called the playback symbol which designates that symbols recorded actions should be playback. Using this grammar the sentence $aaabbbccc$ is produced.

recording sentential form	production rule
(,·S,·, aaabbbccc)	·S → ·aS → ·B#
(,·aS,·B#,aaabbbccc)	accept a
(a,a·S,·B#,aaabbbccc)	a·S → ·aS → ·B
(a,·aS,·BB#,aaabbbccc)	accept a
(aa,a·S,·BB#,abbccc)	a·S → ·aS → ·B
(aa,·aS,·BBB#,abbccc)	accept a
(aaa,a·S,·BBB#,bbccc)	a·S → ·# → ·
(aaa,·#,·BBB#,bbccc)	playback
(aaa,·BBB#,·,bbccc)	·B → ·b → ·c
(aaa,·bBB#,·c,bbccc)	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·,)	-

Naturally we can also define parse tables for recording grammars. So 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	-

Relative to deterministic dotted grammars, recording grammars may guide the acceptance of the language intended with less production rules and smaller parse tables.

How do we construct parsing tables for deterministic dotted grammars?

The TINY parser generator (available at: <http://www.gittens.nl>) currently generates parse tables for deterministic dotted grammars based on my reading of the principle of procrastination:

Postpone until later all things which you might never have to do.

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. 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.

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:

$\cdot S \rightarrow \cdot aS$
 $a \cdot S \rightarrow a \cdot aSBC$
 $a \cdot S \rightarrow a \cdot BC$
 $BC \cdot B \rightarrow \cdot BBC$
 $C \cdot CB \rightarrow \cdot CBC$
 $BC \cdot C \rightarrow \cdot BCC$
 $b \cdot CB \rightarrow b \cdot BC$
 $a \cdot B \rightarrow a \cdot b$
 $b \cdot B \rightarrow b \cdot b$
 $b \cdot C \rightarrow b \cdot c$
 $c \cdot C \rightarrow c \cdot c$

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	righthand side
$\cdot S$	$\{a\}$	$\cdot aS$
$a \cdot S$	$\{a\}$	$a \cdot aSBC$
$a \cdot B$	$\{b\}$	$a \cdot b$
$b \cdot B$	$\{b\}$	$b \cdot b$
$c \cdot C$	$\{c\}$	$c \cdot c$
$b \cdot C$	$\{c\}$	$b \cdot c$

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: $a \cdot B \rightarrow a \cdot b$ is a substring of the right-hand

side of the production rule $a \cdot S \rightarrow a \cdot BC$. From this it follows that whenever the string $a \cdot BC$ is substituted into a sentential form, it is possible that the rule $a \cdot B \rightarrow a \cdot b$ 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 on its relative position in the grammar definition. The connected right-hand sides for our example grammar are:

production rule	connected left hand side
$a \cdot S \rightarrow a \cdot aSBC$	$a \cdot S$
$a \cdot S \rightarrow a \cdot BC$	$a \cdot B$

Before it is verified if a left-hand side is a substring of a right-hand side 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 $a \cdot S \rightarrow a \cdot aSBC$ connects to its own left-hand side in our example grammar.

Computing subsequent terminal sets

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

```
void computeFirstSets()
{
    do
    {
        didSomething = false;
        foreach nt in nonterminals
        {
            foreach rule in nt.getLhsRules()
            {
                foreach alternative in rule.RightHandSideAlternatives()
                {
                    if (alternative.isConnected())
                    {
```


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

Compute the selection sets for the right-hand sides

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	righthand side
$\cdot S$	$\{a\}$	$\cdot aS$
$a \cdot S$	$\{a\}$	$a \cdot aSBC$
$a \cdot S$	$\{b\}$	$a \cdot BC$
$BC \cdot B$	$\{a, b, c\}$	$\cdot BBC$
$a \cdot B$	$\{b\}$	$a \cdot b$
$b \cdot B$	$\{b\}$	$b \cdot b$
$c \cdot C$	$\{c\}$	$c \cdot c$
$C \cdot CB$	$\{a, b, c\}$	$CBC \cdot$
$BC \cdot C$	$\{a, b, c\}$	$\cdot BCC$
$b \cdot CB$	$\{a, b, c\}$	$bBC \cdot$
$b \cdot C$	$\{c\}$	$b \cdot c$

The TINY parser generator

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.

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.

Our example 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);
}

```

Semantic actions

The TINY parser 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, if present, determines when the parser will rewrite with a production rule by return the boolean value true. 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, if present, determines when a right-hand side is selected by return the boolean value true. 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 semantic actions to be performed after rewriting has taken place.

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

Concluding remarks

This article provided a quick example driven overview of generalized deterministic parsing. For more complete information the reader is referred to the article: “Generating efficient table driven parsers for non context free languages” available at: <http://www.gittens.nl>.

References

- [1] Maurice Gittens, Generating efficient table driven parsers for non context free languages. <http://www.gittens.nl>