Grammars and Normal Forms

Read K & S 3.7.

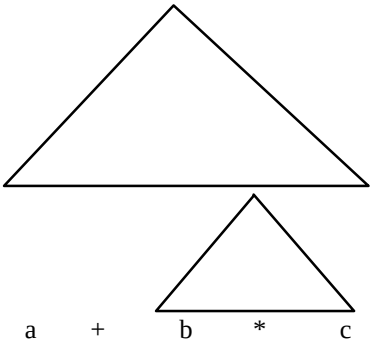
Recognizing Context-Free Languages

Two notions of recognition:

(1) Say yes or no, just like with FSMs

(2) Say yes or no, AND

if yes, describe the structure



Now it's time to worry about extracting structure (and **doing so efficiently**).

Optimizing Context-Free Languages

For regular languages:
Computation = operation of FSMs. So,
Optimization = Operations on FSMs:
Conversion to deterministic FSMs
Minimization of FSMs

For context-free languages:
Computation = operation of parsers. So,
Optimization = **Operations on languages**
Operations on grammars
Parser design

Before We Start: Operations on Grammars

There are lots of ways to transform grammars so that they are more useful for a particular purpose.
the basic idea:

1. Apply transformation 1 to G to get of undesirable property 1. Show that the language generated by G is unchanged.
2. Apply transformation 2 to G to get rid of undesirable property 2. Show that the language generated by G is unchanged AND that undesirable property 1 has not been reintroduced.
3. Continue until the grammar is in the desired form.

Examples:

- Getting rid of ε rules (nullable rules)
- Getting rid of sets of rules with a common initial terminal, e.g.,
  - $A \rightarrow aB$, $A \rightarrow aC$ become $A \rightarrow aD$, $D \rightarrow B \mid C$
- Conversion to normal forms

Normal Forms

If you want to design algorithms, it is often useful to have a limited number of input forms that you have to deal with.

Normal forms are designed to do just that. Various ones have been developed for various purposes.

Examples:

- Clause form for logical expressions to be used in resolution theorem proving
- Disjunctive normal form for database queries so that they can be entered in a query by example grid.
- Various normal forms for grammars to support specific parsing techniques.

Clause Form for Logical Expressions

$\forall x : [Roman(x) \wedge know(x, Marcus)] \rightarrow [hate(x, Caesar) \vee (\forall y : \exists z : hate(y, z) \rightarrow thinkcrazy(x, y))]$

becomes

$\neg Roman(x) \vee \neg know(x, Marcus) \vee hate(x, Caesar) \vee \neg hate(y, z) \vee thinkcrazy(x, z)$

Disjunctive Normal Form for Queries

(category = fruit or category = vegetable)
and
(supplier = A or supplier = B)

becomes

(category = fruit and supplier = A) or
(category = fruit and supplier = B) or
(category = vegetable and supplier = A) or
(category = vegetable and supplier = B)

Category	Supplier	Price
fruit	A	
fruit	B	
vegetable	A	
vegetable	B	

Normal Forms for Grammars

Two of the most common are:

- **Chomsky Normal Form**, in which all rules are of one of the following two forms:
  - $X \rightarrow a$, where $a \in \Sigma$, or
  - $X \rightarrow BC$, where B and C are nonterminals in G

- **Greibach Normal Form**, in which all rules are of the following form:
  - $X \rightarrow a\beta$, where $a \in \Sigma$ and β is a (possibly empty) string of nonterminals

If L is a context-free language that does not contain ε, then if G is a grammar for L, G can be rewritten into both of these normal forms.

What Are Normal Forms Good For?

Examples:

- **Chomsky Normal Form**:
  - $X \rightarrow a$, where $a \in \Sigma$, or
  - $X \rightarrow BC$, where B and C are nonterminals in G

♦ The branching factor is precisely 2. Tree building algorithms can take advantage of that.

- **Greibach Normal Form**
  - $X \rightarrow a\beta$, where $a \in \Sigma$ and β is a (possibly empty) string of nonterminals

♦Precisely one nonterminal is generated for each rule application. This means that we can put a bound on the number of rule applications in any successful derivation.

Conversion to Chomsky Normal Form

Let G be a grammar for the context-free language L where $\varepsilon \notin L$.
We construct G', an equivalent grammar in Chomsky Normal Form by:

0. Initially, let G' = G.
1. Remove from G' all ε productions:
   1.1. If there is a rule $A \rightarrow \alpha B \beta$ and B is nullable, add the rule $A \rightarrow \alpha\beta$ and delete the rule $B \rightarrow \varepsilon$.
   Example:
   $S \rightarrow aA$
   $A \rightarrow B \mid CD$
   $B \rightarrow \varepsilon$
   $B \rightarrow a$
   $C \rightarrow BD$
   $D \rightarrow b$
   $D \rightarrow \varepsilon$

Conversion to Chomsky Normal Form

2. Remove from G' all unit productions (rules of the form $A \rightarrow B$, where B is a nonterminal):
   2.1. Remove from G' all unit productions of the form $A \rightarrow A$.
   2.2. For all nonterminals A, find all nonterminals B such that $A \Rightarrow^* B$, $A \neq B$.
   2.3. Create G" and add to it all rules in G' that are not unit productions.
   2.4. For all A and B satisfying 3.2, add to G"
   $A \rightarrow y1 \mid y2 \mid \ldots$ where $B \rightarrow y1 \mid y2 \mid$ is in G".
   2.5. Set G' to G".

   Example:
   $A \rightarrow a$
   $A \rightarrow B$
   $A \rightarrow EF$
   $B \rightarrow A$
   $B \rightarrow CD$
   $B \rightarrow C$
   $C \rightarrow ab$

At this point, all rules whose right hand sides have length 1 are in Chomsky Normal Form.

Conversion to Chomsky Normal Form

3. Remove from G' all productions P whose right hand sides have length greater than 1 and include a terminal (e.g., $A \rightarrow aB$ or $A \rightarrow BaC$):
   - 3.1. Create a new nonterminal T_a for each terminal a in Σ.
   - 3.2. Modify each production P by substituting T_a for each terminal a.
   - 3.3. Add to G', for each T_a, the rule $T_a \rightarrow a$

Example:

$A \rightarrow aB$
$A \rightarrow BaC$
$A \rightarrow BbC$

$T_a \rightarrow a$
$T_b \rightarrow b$

Conversion to Chomsky Normal Form

4. Remove from G' all productions P whose right hand sides have length greater than 2 (e.g., $A \rightarrow BCDE$)
   - 4.1. For each P of the form $A \rightarrow N_1N_2N_3N_4\ldots N_n$, $n > 2$, create new nonterminals M_2, M_3, … M_{n-1}.
   - 4.2. Replace P with the rule $A \rightarrow N_1M_2$.
   - 4.3. Add the rules $M_2 \rightarrow N_2M_3$, $M_3 \rightarrow N_3M_4$, … $M_{n-1} \rightarrow N_{n-1}N_n$

Example:

$A \rightarrow BCDE$ $(n = 4)$

$A \rightarrow BM_2$
$M_2 \rightarrow C\ M_3$
$M_3 \rightarrow DE$

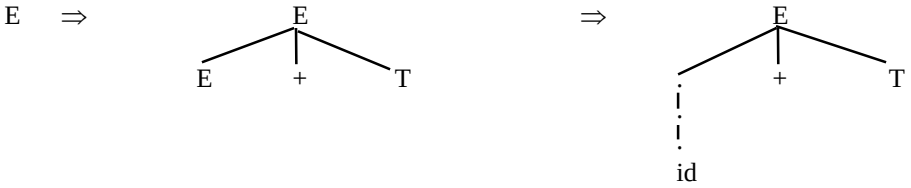
Top Down Parsing

Read K & S 3.8.
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Parsing, Sections 1 and 2.
Do Homework 15.

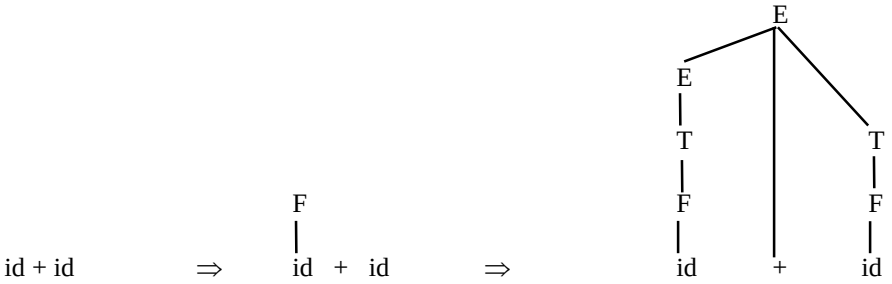
Parsing

Two basic approaches:

Top Down



Bottom Up



A Simple Parsing Example

A simple top-down parser for arithmetic expressions, given the grammar

[1] $E \rightarrow E + T$
[2] $E \rightarrow T$
[3] $T \rightarrow T * F$
[4] $T \rightarrow F$
[5] $F \rightarrow (E)$
[6] $F \rightarrow id$
[7] $F \rightarrow id(E)$

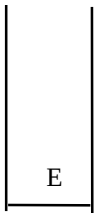
A PDA that does a top down parse:

(0) (1, ε, ε), (2, E)
(1) (2, ε, E), (2, E+T)
(2) (2, ε, E), (2, T)
(3) (2, ε, T), (2, T*F)
(4) (2, ε, T), (2, F)
(5) (2, ε, F), (2, (E))
(6) (2, ε, F), (2, id)
(7) (2, ε, F), (2, id(E))
(8) (2, id, id), (2, ε)
(9) (2, (, ( ), (2, ε)
(10) (2,),)), (2, ε)
(11) (2, +, +), (2, ε)
(12) (2, *, *), (2, ε)

How Does It Work?

Example: id + id * id(id)

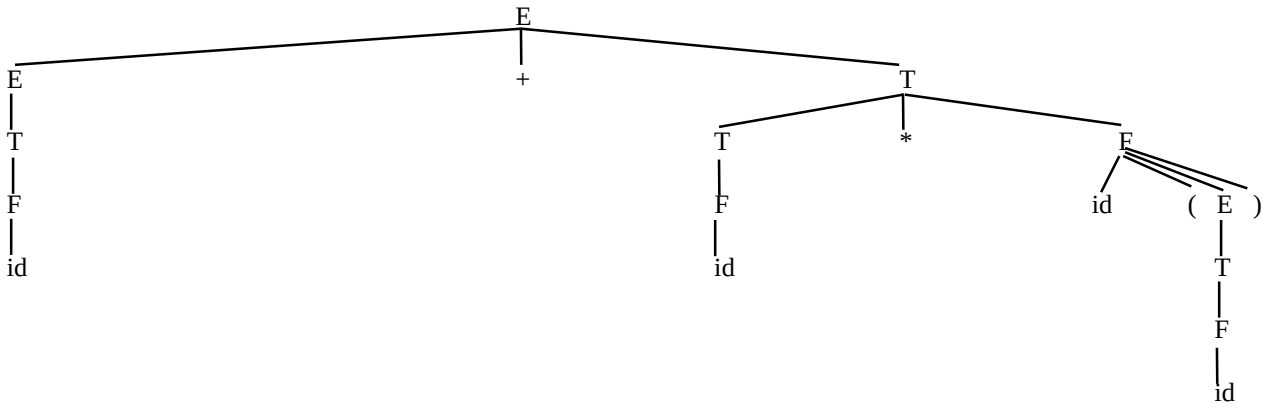
Stack:



What Does It Produce?

The leftmost derivation of the string. Why?

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow id + T \Rightarrow$
$id + T * F \Rightarrow id + F * F \Rightarrow id + id * F \Rightarrow$
$id + id * id(E) \Rightarrow id + id * id(T) \Rightarrow$
$id + id * id(F) \Rightarrow id + id * id(id)$



But the Process Isn't Deterministic

(0) (1, ε, ε), (2, E)
(1) (2, ε, E), (2, E+T)
(2) (2, ε, E), (2, T)
(3) (2, ε, T), (2, T*F)
(4) (2, ε, T), (2, F)
(5) (2, ε, F), (2, (E))
(6) (2, ε, F), (2, id)
(7) (2, ε, F), (2, id(E))
(8) (2, id, id), (2, ε)
(9) (2, (, ( ), (2, ε)
(10) (2,),)), (2, ε)
(11) (2, +, +), (2, ε)
(12) (2, *, *), (2, ε)

nondeterministic

nondeterministic

nondeterministic

Is Nondeterminism A Problem?

Yes.

In the case of regular languages, we could cope with nondeterminism in either of two ways:

- Create an equivalent deterministic recognizer (FSM)
- Simulate the nondeterministic FSM in a number of steps that was still linear in the length of the input string.

For context-free languages, however,

- The best straightforward general algorithm for recognizing a string is $O(n^3)$ and the best (very complicated) algorithm is based on a reduction to matrix multiplication, which may get close to $O(n^2)$.

We'd really like to find a deterministic parsing algorithm that could run in time proportional to the length of the input string.

Is It Possible to Eliminate Nondeterminism?

In this case: Yes

In general: No

Some definitions:

- A PDA **M is deterministic** if it has no two transitions such that for some (state, input, stack sequence) the two transitions could both be taken.

- A language **L is deterministic context-free** if L$ = L(M) for some deterministic PDA M.

Theorem: The class of deterministic context-free languages is a ***proper*** subset of the class of context-free languages.

Proof: Later.

Adding a Terminator to the Language

We define the class of deterministic context-free languages with respect to a terminator ($) because we want that class to be as large as possible.

Theorem: Every deterministic CFL (as just defined) is a context-free language.

Proof:

Without the terminator ($), many seemingly deterministic cfls aren't. Example:

$a^* \cup \{a^n b^n : n > 0\}$

Possible Solutions to the Nondeterminism Problem

1) **Modify the language**
   - Add a terminator $
2) **Change the parsing algorithm**
3) **Modify the grammar**

Modifying the Parsing Algorithm

What if we add the ability to look one character ahead in the input string?

Example: id + id * id(id)
⇑
$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow id + T \Rightarrow$
$id + T * F \Rightarrow id + F * F \Rightarrow id + id * F$

Considering transitions:

(5) (2, ε, F), (2, (E))
(6) (2, ε, F), (2, id)
(7) (2, ε, F), (2, id(E))

If we add to the state an indication of what character is next, we have:

(5) (2, **(**, ε, F), (2, (E))
(6) (2, **id**, ε, F), (2, id)
(7) (2, **id**, ε, F), (2, id(E))

Modifying the Language

So we've solved part of the problem. But what do we do when we come to the end of the input? What will be the state indicator then?

The solution is to modify the language. Instead of building a machine to accept L, we will build a machine to accept L$.

Using Lookahead

		(0) (1, ε, ε), (2, E))
[1]	$E \rightarrow E + T$	(1) (2, ε, E), (2, E+T)
[2]	$E \rightarrow T$	(2) (2, ε, E), (2, T)
[3]	$T \rightarrow T * F$	(3) (2, ε, T), (2, T*F)
[4]	$T \rightarrow F$	(4) (2, ε, T), (2, F)
[5]	$F \rightarrow (E)$	(5) (2, **(**, ε, F), (2, (E))
[6]	$F \rightarrow id$	(6) (2, **id**, ε, F), (2, id)
[7]	$F \rightarrow id(E)$	(7) (2, **id**, ε, F),(2, id(E))
		(8) (2, id, id), (2, ε)
		(9) (2, (, ( ), (2, ε)
		(10) (2,),)), (2, ε)
		(11) (2, +, +), (2, ε)
		(12) (2, *, *), (2, ε)

For now, we'll ignore the issue of when we read the lookahead character and the fact that we only care about it if the top symbol on the stack is F.

Possible Solutions to the Nondeterminism Problem

1) **Modify the language**
   - Add a terminator $
2) **Change the parsing algorithm**
   - Add one character look ahead
3) **Modify the grammar**

Modifying the Grammar

Getting rid of identical first symbols:

[6]	$F \rightarrow id$	(6) (2, id, ε, F),(2, id)
[7]	$F \rightarrow id(E)$	(7) (2, id, ε, F),(2, id(E))

Replace with:

[6']	$F \rightarrow id\ A$	(6') (2, id, ε, F), (2, id A)
[7']	$A \rightarrow \varepsilon$	(7') (2, ¬(, ε, A), (2, ε)
[8']	$A \rightarrow (E)$	(8') (2, (, ε, A), (2, (E))

The general rule for **left factoring**:

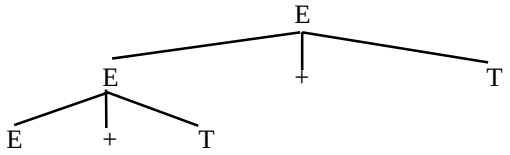
Whenever $A \rightarrow \alpha\beta_1$
$A \rightarrow \alpha\beta_2 \ldots$
$A \rightarrow \alpha\beta_n$
are rules with $\alpha \neq \varepsilon$ and $n \geq 2$, then replace them by the rules:
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1$
$A' \rightarrow \beta_2 \ldots$
$A' \rightarrow \beta_n$

Modifying the Grammar

Getting rid of left recursion:

[1]	$E \rightarrow E + T$	(1) (2, ε, E), (2, E+T)
[2]	$E \rightarrow T$	(2) (2, ε, E), (2, T)

The problem:



Replace with:

[1]	$E \rightarrow T\ E'$	(1) (2, ε, E), (2, T E')
[2]	$E' \rightarrow + T\ E'$	(2) (2, ε, E'), (2, + T E')
[3]	$E' \rightarrow \varepsilon$	(3) (2, ε, E'), (2, ε)

Getting Rid of Left Recursion

The general rule for eliminating **left recursion**:

If G contains the following rules:	Replace them with:
$A \rightarrow A\alpha_1$	$A' \rightarrow \alpha_1 A'$
$A \rightarrow A\alpha_2$ …	$A' \rightarrow \alpha_2 A'$ …
$A \rightarrow A\alpha_3$	$A' \rightarrow \alpha_3 A'$
$A \rightarrow A\alpha_n$	$A' \rightarrow \alpha_n A'$
	$A' \rightarrow \varepsilon$
$A \rightarrow \beta_1$ (where β's do not start with $A\alpha$)	$A \rightarrow \beta_1 A'$
$A \rightarrow \beta_2$	$A \rightarrow \beta_2 A'$
…	…
$A \rightarrow \beta_m$	$A \rightarrow \beta_m A'$

and $n > 0$, then

Possible Solutions to the Nondeterminism Problem

I. **Modify the language**
   A. Add a terminator $
II. **Change the parsing algorithm**
   A. Add one character look ahead
III. **Modify the grammar**
   A. Left factor
   B. Get rid of left recursion

LL(k) Languages

We have just offered heuristic rules for getting rid of some nondeterminism.

We know that not all context-free languages are deterministic, so there are some languages for which these rules won't work.

We define a **grammar** to be **LL(k)** if it is possible to decide what production to apply by looking ahead at most k symbols in the input string.

Specifically, a **grammar** G is **LL(1)** iff, whenever
$A \rightarrow \alpha \mid \beta$ are two rules in G:

1. For no terminal a do α and β derive strings beginning with a.
2. At most one of $\alpha \mid \beta$ can derive ε.
3. If $\beta \Rightarrow^* \varepsilon$, then α does not derive any strings beginning with a terminal in FOLLOW(A), defined to be the set of terminals that can immediately follow A in some sentential form.

We define a **language** to be **LL(k)** if there exists an LL(k) grammar for it.

Implementing an LL(1) Parser

If a language L has an LL(1) grammar, then we can build a deterministic LL(1) parser for it. Such a parser scans the input **L**eft to right and builds a **L**eftmost derivation.

The heart of an LL(1) parser is the parsing table, which tells it which production to apply at each step.
For example, here is the parsing table for our revised grammar of arithmetic expressions without function calls:

V\Σ	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Given input id + id * id, the first few moves of this parser will be:

	E	id + id * id$
E→TE'	TE'	id + id * id$
T→FT'	FT'E'	id + id * id$
F→id	idT'E'	id + id * id$
	T'E'	+ id * id$
T'→ε	E'	+ id * id$

But What If We Need a Language That Isn't LL(1)?

Example:

ST → if C then ST else ST
ST → if C then ST

We can apply left factoring to yield:

ST → if C then ST S'
S' → else ST | ε

Now we've procrastinated the decision. But the language is still ambiguous. What if the input is

if C_1 then if C_2 then ST_1 else ST_2

Which bracketing (rule) should we choose?

A common practice is to choose S' → else ST

We can force this if we create the parsing table by hand.

Possible Solutions to the Nondeterminism Problem

I. **Modify the language**
   A. Add a terminator $
II. **Change the parsing algorithm**
   A. Add one character look ahead
   B. Use a parsing table
   C. Tailor parsing table entries by hand
III. **Modify the grammar**
   A. Left factor
   B. Get rid of left recursion

The Price We Pay

Old Grammar

[1] $E \rightarrow E + T$
[2] $E \rightarrow T$

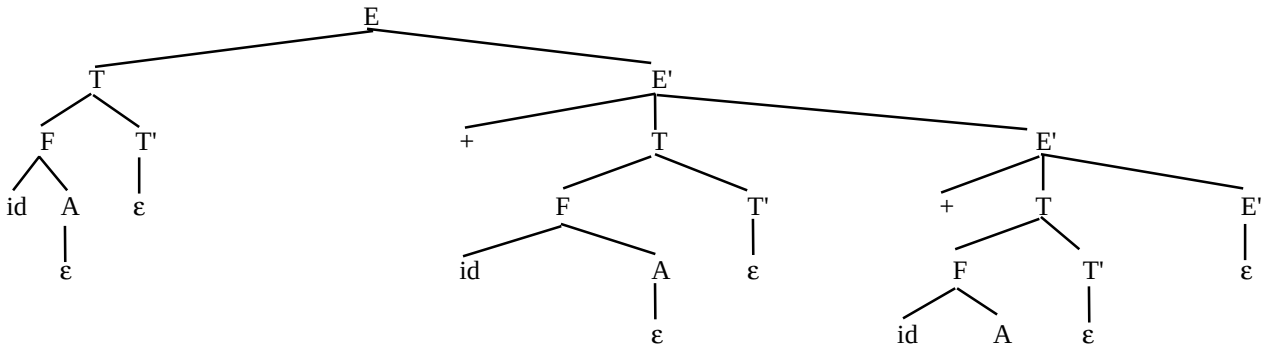
[3] $T \rightarrow T * F$
[4] $T \rightarrow F$

[5] $F \rightarrow (E)$
[6] $F \rightarrow id$
[7] $F \rightarrow id(E)$

New Grammar

$E \rightarrow TE'$
$E' \rightarrow +TE'$
$E' \rightarrow \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'$
$T' \rightarrow \varepsilon$
$F \rightarrow (E)$
$F \rightarrow idA$
$A \rightarrow \varepsilon$
$A \rightarrow (E)$

input = id + id + id



Comparing Regular and Context-Free Languages

Regular Languages

- regular exprs.
  or
- regular grammars
- = DFSAs
- recognize
- minimize FSAs

Context-Free Languages

- context-free grammars

- = NDPDAs
- parse
- find deterministic grammars
- find efficient parsers