Parsers, Context-free Grammars & Top-down Parsing

MICHAEL WOLLOWSKI

Purpose of Parsers

A parser takes the input from the lexical analyzer,

Determines whether the input is a valid sentence in the source language and

Produces a parse tree for further processing

Limitations of Regular Expressions

REs are not sufficient to capture the structure of a language.

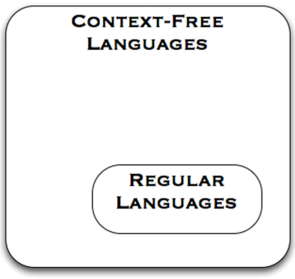
Consider blocks in Java:

```
{
  stmt
  stmt
  {
    stmt
  }
}
```

Why can this structure not be specified by a regular expressions?

Context-Free Grammars

Context-free grammars are more expressive than regular expressions



Context-Free Grammars

A *context-free grammar G* is a quadruple (*T*, *NT, S, P*) where:

- *T* is the set of terminal symbols, or words, in the language L(G). Terminal symbols correspond to syntactic categories returned by the scanner.
- *NT* is the set of nonterminal symbols that appear in the productions of G. Non-terminals are variables to provide abstraction and structure in the productions.
- *S* is a nonterminal designated as the *start symbol* of the grammar. *S* represents the set of sentences in L(G).
- *P* is the set of productions or rewrite rules in G. Each rule in *P* has the form $NT \rightarrow (T \cup NT)^+$. It replaces a single nonterminal with a string of one or more grammar symbols.

Context-Free Grammars

Here is a sample grammar:

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Derivations

A derivation is the process of showing that a sentence is a member of the language defined by a grammar.

It begins with the start symbol, *Expr*

In the example to the right, we derive **(a + b) * c**

Rule	Sentential Form
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name

Rightmost Derivation of (a + b) × c

Leftmost vs Rightmost Derivation

In a rightmost derivation, we attempt to use a rule which replaces the rightmost nonterminal.

In a leftmost derivation, we attempt to rewrite the leftmost nonterminal.

Example:

Rule	Sentential Form
	Expr
2	*Expr Op* name
1	(*Expr*) *Op* name
2	(*Expr Op* name) *Op* name
3	(name *Op* name) *Op* name
4	(name + name) *Op* name
6	(name + name) × name

Leftmost Derivation of (a + b) x c

Leftmost vs Rightmost Derivation

While the derivations are different, the parse trees are the same.

Rule	Sentential Form
	Expr
2	*Expr Op* name
1	(*Expr*) *Op* name
2	(*Expr Op* name) *Op* name
3	(name *Op* name) *Op* name
4	(name + name) *Op* name
6	(name + name) × name

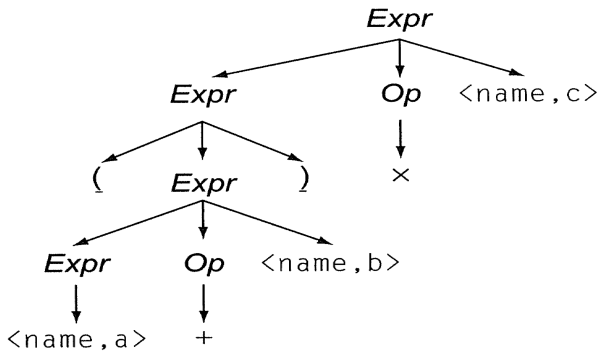
Leftmost Derivation of (a + b) x c

Rule	Sentential Form
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name

Rightmost Derivation of (a + b) × c

Derivations

The corresponding parse tree looks like this:



Ambiguity

A grammar *G* is ambiguous, if some sentence in L(G) has more than one rightmost (or leftmost) derivation.

Consider the following grammar:

Statement → `if` *Expr* `then` *Statement* `else` *Statement* |

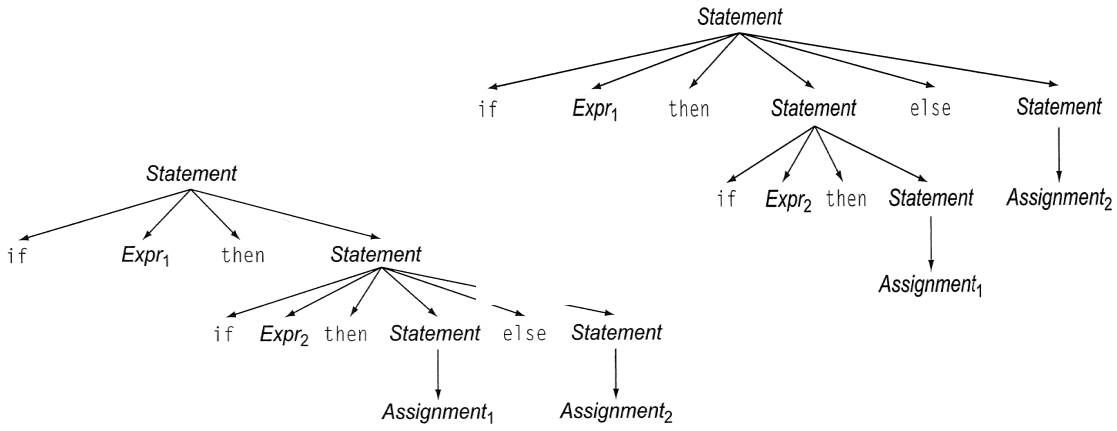
`if` *Expr* `then` *Statement* |

Assignment

The following code fragment has two distinct derivations:

`if` *Expr1* `then if` *Expr2* `then` *Assignment1* `else` *Assignment2*

Ambiguity Example



Removing Ambiguity

The general rule is to match each **else** with the closest previously unmatched **then**.

Rewrite the grammar so that a statement appearing between a **then** and **else** must be matched.

In other words, it must not end with an unmatched **then** followed by any statement

Removing Ambiguity

The original grammar:

Statement → if *Expr* then *Statement* else *Statement* |
if *Expr* then *Statement* |
Other

Ambiguity removed:

Statement → Matched_statement | Unmatched_statement

Matched_statement → if *Expr* then *Matched_Statement* else *Matched_Statement* |
Other

Unmatched_statement → if *Expr* then *Statement* |
if *Expr* then *Matched_Statement* else *Unmatched_Statement*|
Other

Precedence

Consider the following grammar:

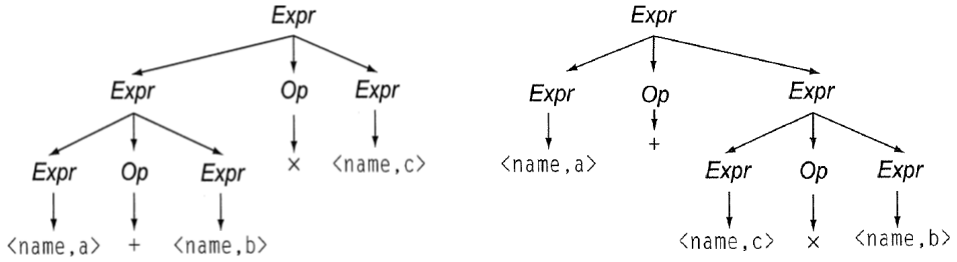
1	*Expr*	→	(*Expr*)
2		|	*Expr Op Expr*
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Derive the expression: a + b * c

Precedence

We can obtain two different parse trees.

One obeys the precedence of the arithmetic operators.



16

Precedence

We eliminate the ambiguity that leads to potentially incorrect parse trees by modifying the grammar.

In essence, we move operators with higher precedence further down in the grammar.

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	num
9		|	name

Precedence In-class Assignment

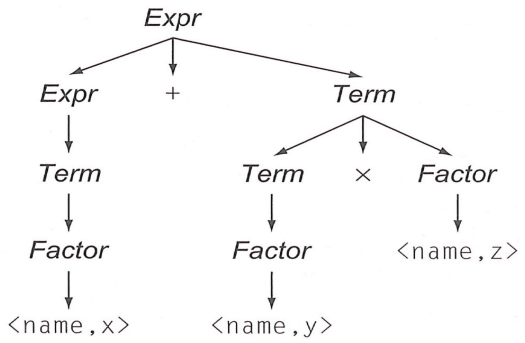
Using the revised grammar, derive a + b * c, again

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	num
9		|	name

18

Precedence

The resulting parse tree:



19

Top-Down Parsers

A *top-down* parser, as the name suggests, begins with the start symbol and systematically applies the rules of the CFG, until there are no more non-terminal symbols left.

By contrast, a *bottom-up* parser, begins with the tokens returned by the lexical analyzer and using the rules of the CFG, replaces them with non-terminals until the start symbol is reached.

Graphically speaking, a top-down parser builds the parse tree from the root, while a bottom-up parser begins with the leaves, combining partial parse trees.

20

Oracles

Given an unmodified CFG, the challenge of a parser is to decide which rule to pick so as to avoid back-tracking.

If the parser were to be all knowing, it would always select the right rule, as shown:

Rule	Sentential Form	Input
	Expr	↑ name + name x name
1	*Expr* + *Term*	↑ name + name x name
3	*Term* + *Term*	↑ name + name x name
6	*Factor* + *Term*	↑ name + name x name
9	name + *Term*	↑ name + name x name
→	name + *Term*	name ↑ + name x name
→	name + *Term*	name + ↑ name x name
4	name + *Term* x *Factor*	name + ↑ name x name
6	name + *Factor* x *Factor*	name + ↑ name x name
9	name + name x *Factor*	name + ↑ name x name
→	name + name x *Factor*	name + name ↑ x name
→	name + name x *Factor*	name + name x ↑ name
9	name + name x name	name + name x ↑ name
→	name + name x name	name + name x name ↑

Classes of Context-free Grammars

There are no oracles!

However, we can modify the grammar so as to make oracles unnecessary.

Classes of Context-free Grammars

LR(1) grammars include a large subset of the unambiguous CFGs.

LR(1) grammars can be parsed, bottom-up, in a linear scan from left to right, looking at most one word ahead of the current input symbol. The “R” stands for a right-most derivation.

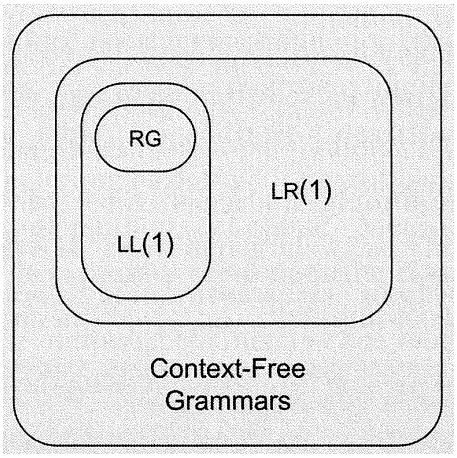
The widespread availability of tools that derive parsers from LR(1) grammars has made LR(1) parsers “everyone’s” favorite grammar.

LL(1) grammars are an important subset of the LR(1) grammars.

LL(1) grammars can be parsed, top-down, in a linear scan from left to right, looking at most one word ahead of the current input symbol. The “L” stands for a left-most derivation.

Many programming languages can be written in an LL(1) grammar.

Classes of Context-free Grammars



Top-Down Parsing: Eliminating Left-Recursion

We will focus on Top-down parsing for now.

Consider the following excerpt from our grammar:

$$\begin{aligned} Expr &\rightarrow Expr + Term \\ &\mid Expr - Term \\ &\mid Term \end{aligned}$$

Suppose we wish to derive: `a + b * c`

The grammar is recursive.

Computers being as systematic as they are, our compiler may decide to always use the first rule, leading to:

$$Expr \rightarrow Expr + Expr + Expr + \ldots$$

Top-Down Parsing: Eliminating Left-Recursion

We are going to re-write the productions that are left-recursive to be right-recursive.

This requires the introduction of additional non-terminals.

The general pattern of this transformation is as follows:

$A \rightarrow A\alpha \mid \beta$

is transformed to:

$A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \varepsilon$

Top-Down Parsing: Eliminating Left-Recursion

Result of eliminating left-recursion from the productions for ***Expr*** and ***Term***

Notice the right-recursive nature of the grammar

Original			Transformed		
Expr	→	*Expr* + *Term*	*Expr*	→	*Term Expr′*
	|	*Expr* - *Term*	*Expr′*	→	+ *Term* Expr′
	|	*Term*		|	- *Term Expr′*
				|	ϵ
Term	→	*Term* x *Factor*	*Term*	→	*Factor Term′*
	|	*Term* ÷ *Factor*	*Term′*	→	x *Factor Term′*
	|	*Factor*		|	÷ *Factor Term′*
				|	ϵ

Top-Down Parsing: Eliminating Left-Recursion

Visually, this works out to be like so (courtesy of the dragon book)

A -> Aα | β

A -> βR, R-> αR|ε

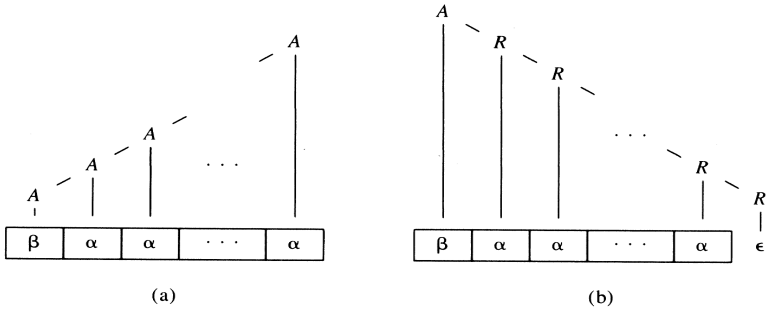


Fig. 2.18. Left- and right-recursive ways of generating a string.

Eliminating Left-Recursion: In-class Assignment

Using the revised grammar, evaluate: a + b * c

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name