Concepts of Programming Languages

Lecture 4 - Grammars

Patrick Donnelly

Montana State University

Spring 2014

Administrivia

Assignments:

Programming #1 : due 02.10

Reading:

Chapter 3

> *A language that is simple to parse for the compiler is also simple to parse for the human programmer.*
>
> N. Wirth (1974)

Terminology

Definition

A ***sentence*** is a string of characters over some alphabet.

Definition

A ***language*** is a set of sentences.

Definition

A ***lexeme*** is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`).

Definition

A ***token*** is a category of lexemes (e.g., identifier).

Grammars

Definition

A ***metalanguage*** is a language used to define other languages.

Definition

A ***grammar*** is a metalanguage used to define the syntax of a language.

This course is interested in using grammars to define the syntax of a programming language.

Context-Free Grammars

Developed by Noam Chomsky in the mid-1950s

Language generators, meant to describe the syntax of natural languages

Define a class of languages called context-free languages

Backus-Naur Form (BNF)

Definition

Backus Normal Form (1959) is a stylized version of a context-free grammar (cf. Chomsky hierarchy)

First used to define syntax of Algol 60

Now used to define syntax of most major languages

BNF Grammar

Definition

Nonterminals act like syntactic variables for representing classes of syntactic structures.

Definition

Terminals are lexemes or tokens.

BNF Grammar

Definition

Nonterminals act like syntactic variables for representing classes of syntactic structures.

Definition

Terminals are lexemes or tokens.

Definition

A ***production rule*** has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

BNF Grammar

Definition

Nonterminals act like syntactic variables for representing classes of syntactic structures.

Definition

Terminals are lexemes or tokens.

Definition

A ***production rule*** has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

Definition

A ***start symbol*** is a special element of the nonterminals of a grammar.

BNF Grammar

Set of	*productions*:	P
	terminal symbols:	T
	nonterminal symbols:	N
	start symbols:	$S \in N$

A *production* has the form:

$$A \rightarrow \omega$$

where $A \in N$ and $\omega \in (N \cup T)^*$

Example: Binary Digits

Consider the grammar:

```
binaryDigit → 0
binaryDigit → 1
```

or equivalently:

```
binaryDigit → 0 | 1
```

where | is a metacharacter that separates alternatives.

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

We can derive any unsigned integer, like `352`, from this grammar.

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Derivation of 352 as an Integer is a 6-step process.

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Use a grammar rule to enable each step:

Integer → Integer Digit

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Replace a nonterminal by a right-hand side of one of its rules:

$$\begin{array}{llll} \text{Integer} & \rightarrow & \text{Integer} & \text{Digit} \\ & \rightarrow & \text{Integer} & 2 \end{array}$$

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Each step follows from the one before it:

$$\begin{array}{llll} \text{Integer} & \rightarrow & \text{Integer} & & \text{Digit} \\ & \rightarrow & \text{Integer} & & 2 \\ & \rightarrow & \text{Integer} & \text{Digit} & 2 \end{array}$$

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

$$\begin{array}{llll} \text{Integer} & \rightarrow & \text{Integer} & & \text{Digit} \\ & \rightarrow & \text{Integer} & & 2 \\ & \rightarrow & \text{Integer} & \text{Digit} & 2 \\ & \rightarrow & \text{Integer} & 5 & 2 \end{array}$$

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

$$
\begin{array}{llllll}
\text{Integer} & \rightarrow & \text{Integer} & & & \text{Digit} \\
 & \rightarrow & \text{Integer} & & & 2 \\
 & \rightarrow & \text{Integer} & \text{Digit} & & 2 \\
 & \rightarrow & \text{Integer} & 5 & & 2 \\
 & \rightarrow & \text{Digit} & 5 & & 2
\end{array}
$$

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

You finish when there are only terminal symbols remaining.

$$\begin{array}{llll} \text{Integer} & \rightarrow & \text{Integer} & & \text{Digit} \\ & \rightarrow & \text{Integer} & & 2 \\ & \rightarrow & \text{Integer} & \text{Digit} & 2 \\ & \rightarrow & \text{Integer} & 5 & 2 \\ & \rightarrow & \text{Digit} & 5 & 2 \\ & \rightarrow & 3 & 5 & 2 \end{array}$$

Derivations

Consider the grammar:

```
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This method was a *rightmost derivation.*

$$\begin{array}{lllll}
\text{Integer} & \rightarrow & \text{Integer} & & \text{Digit} \\
 & \rightarrow & \text{Integer} & & 2 \\
 & \rightarrow & \text{Integer} & \text{Digit} & 2 \\
 & \rightarrow & \text{Integer} & 5 & 2 \\
 & \rightarrow & \text{Digit} & 5 & 2 \\
 & \rightarrow & 3 & 5 & 2
\end{array}$$

Derivations

An alternate derivation of `352`.

$$\begin{array}{llll} \text{Integer} & \rightarrow & \text{Integer} & \text{Digit} & \\ & \rightarrow & \text{Integer} & \text{Digit} & \text{Digit} \\ & \rightarrow & \text{Digit} & \text{Digit} & \text{Digit} \\ & \rightarrow & 3 & \text{Digit} & \text{Digit} \\ & \rightarrow & 3 & 5 & \text{Digit} \\ & \rightarrow & 3 & 5 & 2 \end{array}$$

This is called a *leftmost derivation*, since at each step the leftmost nonterminal is replaced.

Notation for Derivations

$$\texttt{Integer} \rightarrow^{*} \texttt{352}$$

Means that `352` can be derived in a finite number of steps using the grammar for *Integer*.

Notation for Derivations

$$\texttt{Integer} \rightarrow^{*} \texttt{352}$$

Means that `352` can be derived in a finite number of steps using the grammar for *Integer*.

$$\texttt{352} \rightarrow \texttt{L(G)}$$

Means that `352` is a member of the language defined by grammar *G*.

Notation for Derivations

$$\texttt{Integer} \rightarrow^{*} \texttt{352}$$

Means that `352` can be derived in a finite number of steps using the grammar for *Integer.*

$$\texttt{352} \rightarrow \texttt{L(G)}$$

Means that `352` is a member of the language defined by grammar *G*.

$$\texttt{L(G)} = \{\ \omega \in T^{*} \mid \texttt{Integer} \rightarrow^{*} \omega\ \}$$

Means that the language defined by grammar *G* is the set of all symbol strings ω that can be derived as an *Integer*.

Parse Trees

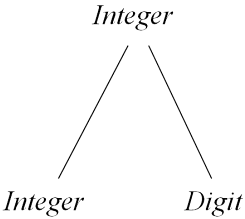
Definition

A ***parse tree*** is a graphical representation of a derivation.

- Each internal node of the tree corresponds to a step in the derivation.
- Each child of a node represents a right-hand side of a production.
- Each leaf node represents a symbol of the derived string, reading from left to right.

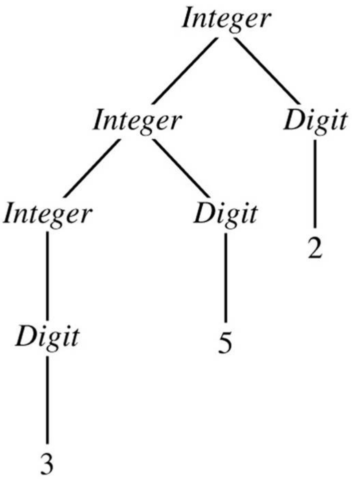
Parse Trees

The step `Integer → Integer Digit` appears in the parse tree as:



Parse Trees

Parse Tree for 352 as an Integer:



Arithmetic Expression Grammar

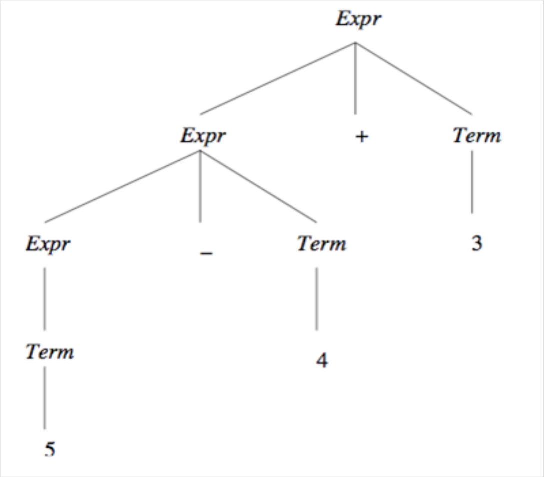
The following grammar defines the language of arithmetic expressions with:

- 1-digit integers,
- addition, and
- subtraction.

```
Expr → Expr + Term | Expr - Term | Term
Term → 0 | ...| 9 | ( Expr )
```

Parse Trees

Parse of the String 5-4+3:



Associativity and Precedence

A grammar can be used to define associativity and precedence among the operators in an expression.

Associativity and Precedence

A grammar can be used to define associativity and precedence among the operators in an expression.

Example

+ and - are left-associative operators in mathematics;
* and / have higher precedence than + and - .

Associativity and Precedence

A grammar can be used to define associativity and precedence among the operators in an expression.

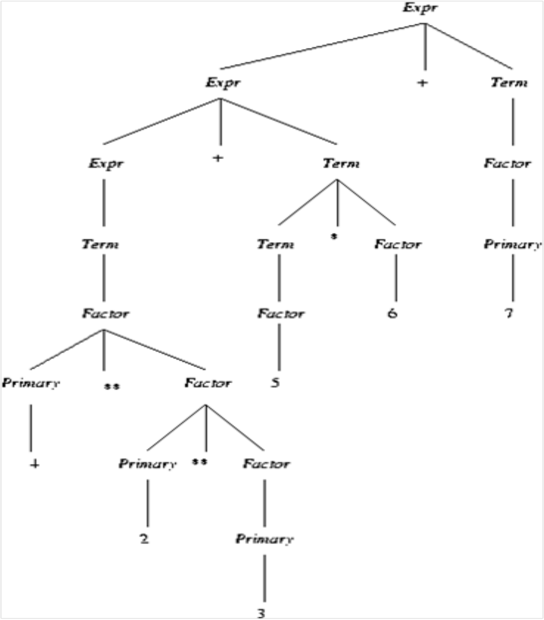
Example

+ and - are left-associative operators in mathematics;
* and / have higher precedence than + and - .

Consider the more interesting grammar G_1:

```
Expr → Expr + Term | Expr - Term | Term
Term → Term * Factor | Term / Factor |
        Term % Factor | Factor
Factor → Primary ** Factor | Primary
Primary → 0 | ...| 9 | ( Expr )
```

Parse of
4**2**3+5*6+7
for Grammar G_1



Associativity and Precedence for G_1

Precedence	Associativity	Operators
3	right	**
2	left	* / %
1	left	+ -

These relationships are shown by the structure of the parse tree: highest precedence at the bottom, and left-associativity on the left at each level.

Derivations

Definition

Every string of symbols in a derivation is a ***sentential form***.

Definition

A ***sentence*** is a sentential form that has only terminal symbols.

Definition

A ***leftmost derivation*** is one in which the leftmost nonterminal in each sentential form is the one that is expanded.

Ambiguous Grammars

Definition

A grammar is ***ambiguous*** if one of its strings has two or more diffferent parse trees.

e.g., Grammar G_1 *above is unambiguous.*

Ambiguous Grammars

Definition

A grammar is ***ambiguous*** if one of its strings has two or more diffferent parse trees.

e.g., Grammar G_1 *above is unambiguous.*

C, C++, and Java have a large number of

- operators and
- precedence levels

Instead of using a large grammar, we can:

- Write a smaller ambiguous grammar, and
- Give separate precedence and associativity

An Ambiguous Expression

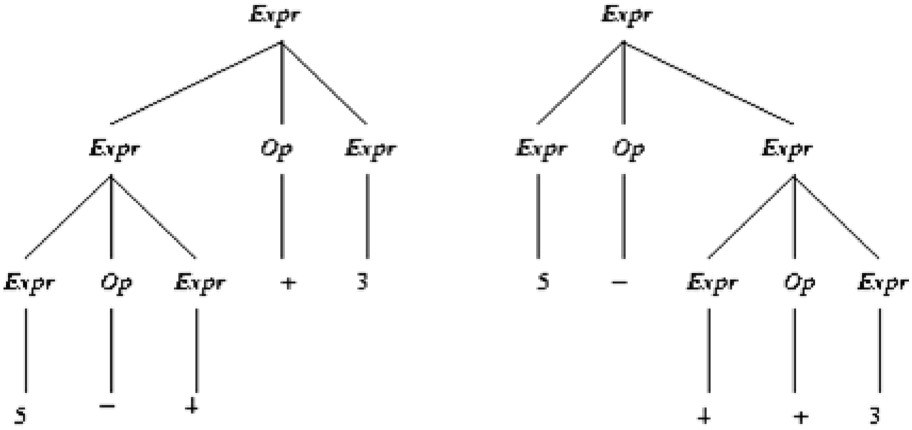
Consider the grammar G_2:

```
    Expr → Expr Op Expr | ( Expr ) | Integer
                + | − | * | / | % | **
```

Notes:

- G_2 is equivalent to G_1 (i.e., its language is the same)
- G_2 has fewer productions and nonterminals than G_1.
- However, G_2 is ambiguous.

Ambiguous Parse of 5-4+3 using grammar G_2



(a)

(b)

Ambiguous Expression Grammar

Ambiguous Example:

```
<expr> → <expr> <op> <expr> | const
<op> → / | -
```

Ambiguous Expression Grammar

Ambiguous Example:

```
<expr> → <expr> <op> <expr> | const
<op> → / | -
```

Unambiguous Example:

```
<expr> → <expr> - <term> | <term>
<term> → <term> / const| const
```

The Dangling Else

```
IfStatement -> if ( Expression ) Statement |
     if ( Expression ) Statement else Statement

Statement -> Assignment | IfStatement | Block

Block ->  Statements

Statements -> Statements Statement | Statement
```

Example

With which ‘if’ does the ‘else’ associate?

```
if (x < 0)
    if (y < 0)
        y = y - 1;
else
    y = 0;
```

Example

With which ‘if’ does the ‘else’ associate?   The first?

```
if (x < 0)
    if (y < 0)
        y = y - 1;
else
    y = 0;
```

Example

With which ‘if’ does the ‘else’ associate? The first or the second?

```
if (x < 0)
    if (y < 0)
        y = y - 1;
    else
        y = 0;
```

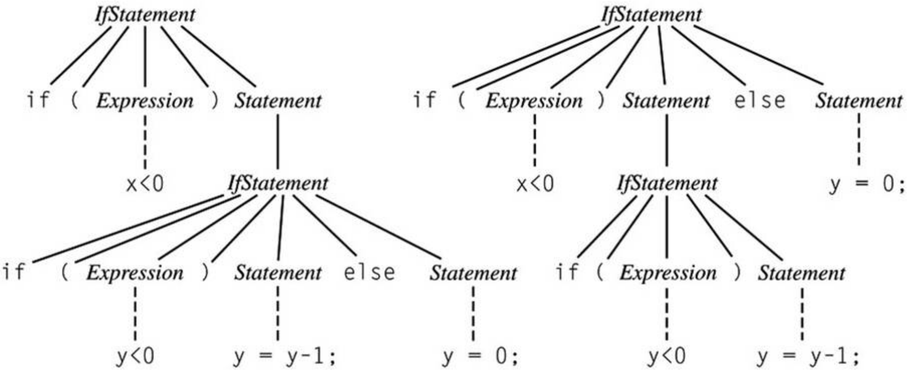
Example

With which ‘if’ does the ‘else’ associate? The first or the second?

```
if (x < 0)
    if (y < 0)
        y = y - 1;
    else
        y = 0;
```

Answer: either one!

The *Dangling Else* Ambiguity



Solving the *Dangling Else* Ambiguity

1. Algol 60, C, C++:
   associate each else with closest `if`;
   use `{}` or `begin...end` to override.

Solving the *Dangling Else* Ambiguity

1. Algol 60, C, C++:
   associate each else with closest `if`;
   use `{}` or `begin...end` to override.

2. Algol 68, Modula, Ada:
   use explicit delimiter to end every conditional (e.g., `if...fi`)

Solving the *Dangling Else* Ambiguity

1. Algol 60, C, C++:
   associate each else with closest `if`;
   use `{}` or `begin...end` to override.

2. Algol 68, Modula, Ada:
   use explicit delimiter to end every conditional (e.g., `if...fi`)

3. Java: rewrite the grammar to limit what can appear in a conditional:

```
IfThenStatement → if ( Expression ) Statement
IfThenElseStatement → if ( Expression )
                          StatementNoShortIf
                      else Statement
```

The category `StatementNoShortIf` includes all except `IfThenStatement`.

Extended BNF (EBNF)

BNF:

- recursion for iteration
- nonterminals for grouping

EBNF: additional metacharacters

- { } for a series of zero or more
- ( ) for a list, must pick one
- [ ] for an optional list; pick none or one

EBNF Examples

Expression is a list of one or more ***Terms*** separated by operators + and -

```
Expression → Term { ( + | - ) Term }
IfStatement → if ( Expression ) Statement
                 [ else Statement ]
```

EBNF Examples

Expression is a list of one or more ***Terms*** separated by operators + and -

Expression → Term { (+ | −) Term }
IfStatement → **if** (Expression) Statement
[**else** Statement]

C-style EBNF lists alternatives vertically and uses `opt` to signify optional parts.

IfStatement:
if (Expression) Statement $\texttt{ElsePart}_{opt}$
ElsePart:
else Statement

EBNF to BNF

We can always rewrite an EBNF grammar as a BNF grammar.

For example:

$$\texttt{A} \rightarrow \texttt{x} \ \{ \ \texttt{y} \ \} \ \texttt{z}$$

can be rewritten:

$$\begin{aligned} \texttt{A} &\rightarrow \texttt{x} \ \texttt{A}' \ \texttt{z} \\ \texttt{A}' &\rightarrow \epsilon \mid \texttt{y} \ \texttt{A}' \end{aligned}$$

While EBNF is no more powerful than BNF, its rules are often simpler and clearer.

EBNF to BNF Example

BNF:

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

EBNF to BNF Example

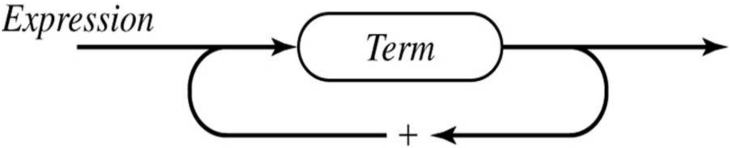
BNF:

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

EBNF:

```
<expr> → <term> (+ | -) <term>
<term> → <factor> (* | /) <factor>
```

Syntax Diagram for Expressions with Addition



Chomsky Hierarchy

1. Regular grammar
2. Context-free grammar (BNF)
3. Context-sensitive grammar
4. Unrestricted grammar

Regular Grammar

Simplest; least powerful

Equivalent to:

- Regular expression
- Finite-state automaton

Right regular grammar: $\omega \in T^*, B \in N$

- $A \rightarrow \omega B$
- $A \rightarrow \omega$

```
Integer → 0 Integer | 1 Integer | ...| 9 Integer |
          0 | 1 | ...| 9
```

Regular Grammars

Left regular grammar: equivalent

Used in construction of tokenizers

Less powerful than context-free grammars

Not a regular language, such as:

$$\{a^n b^n | n \geq 1\}$$

Therefore, cannot balance: ( ), { }, `begin end`

Context-Free Grammars

BNF a stylized form of CFG

Equivalent to a pushdown automaton

For a wide class of unambiguous CFGs, there are table-driven, linear time parsers

Context-Sensitive Grammars

Production:

$$\alpha \to \beta \qquad |\alpha| \geq |\beta|$$

$$\alpha, \beta \in (N \cup T)^*$$

Lefthand side can be composed of strings of terminals and nonterminals

Undecidable Properties of CSGs

Given a string ω and grammar $G : \omega \in L(G)$

$L(G)$ is non-empty

Definition

Undecidable means that you cannot write a computer program that is guaranteed to halt to decide the question for all $\omega \in L(G)$.

Unrestricted Grammar

Equivalent to:

- Turing machine
- von Neumann machine
- C++, Java

That is, can compute any computable function.