Recall: Programming Language Specs

- Syntax of every significant programming language is specified by a formal grammar
  - BNF or some variation there on
- As language engineering has developed, formal methods have improved for defining useful grammars and tools for processing them

Productions

- The rules of a grammar are called *productions*
- Rules contain
  - Nonterminal symbols: grammar variables (program, statement, id, etc.)
  - Terminal symbols: concrete syntax that appears in programs: a, b, c, 0, 1, if, (, ...
- Meaning of nonterminal :<sequence of terminals and nonterminals>

  In a derivation, an instance of nonterminal can be replaced by the sequence of terminals and nonterminals on the right of the production
- Often, there are two or more productions for a single nonterminal – can use either at different times

Grammar for D, a little language

```plaintext
program ::= function-def | program function-def
function-def ::= int id () { statements }
                | int id ( parameters ) { statements }
                | int id ( ) { declarations statements }
parameters ::= parameter | parameters , parameter
declarations ::= declaration | declarations declaration
declaration ::= int id
statements ::= statement | statements statement
statement ::= id = exp ; | return exp ; | { statements }
             | if ( bool-exp ) statement | if ( bool-exp ) statement else statement
             | while ( bool-exp ) statement
bool-exp ::= rel-exp | ! ( rel-exp )
rel-exp ::= exp == exp | exp > exp
exp ::= term | exp + term | exp - term
term ::= factor | term * factor
factor ::= id | int ( exp ) | id ( ) | id ( exps )
exps ::= exp | exps , exp
```
Grammar for Java, a big language

  » Entire document
    500+ pages
    Grammar productions with explanatory text
  » Chapter 18, Syntax
    8 pages of grammar productions, presented in "BNF-style"

Parsing

- Parsing: reconstruct the derivation (syntactic structure) of a program
- In principle, a single recognizer could work directly from the concrete, character-by-character grammar
  » In practice this is never done

Parsing & Scanning

- In real compilers the recognizer is split into two phases
  » Scanner: translate input characters to tokens
    Also, report lexical errors like illegal characters and illegal symbols
  » Parser: read token stream and reconstruct the derivation

Parsing

- The syntax of most programming languages can be specified by a context-free grammar (CFG)
- Parsing
  » Given a grammar $G$ and a sentence $w$ in $L(G)$, traverse the derivation (parse tree) for $w$ in some standard order and do something useful at each node
  » The tree might not be produced explicitly, but the control flow of a parser corresponds to a traversal
For practical reasons we want the parser to be deterministic (no backtracking), and we want to examine the source program from left to right.

» parse the program in linear time in the order it appears in the source file

**Common Orderings**

- Top-down
  » Start with the root
  » Traverse the parse tree depth-first, left-to-right (leftmost derivation)
  » LL(k)
- Bottom-up
  » Start at leaves and build up to the root
    Effectively a rightmost derivation in reverse
  » LR(k) and subsets (LALR(k), SLR(k), etc.)

**“Something Useful”**

- At each point (node) in the traversal, perform some semantic action
  » Construct nodes of full parse tree (rare)
  » Construct abstract syntax tree (common)
  » Construct linear, lower-level representation (more common in later parts of a modern compiler)
  » Generate target code on the fly (1-pass compiler; not common in production compilers – can’t generate very good code in one pass)
Context-Free Grammars

- Formally, a grammar $G$ is a tuple $<N, \Sigma, P, S>$ where
  - $N$ a finite set of non-terminal symbols
  - $\Sigma$ a finite set of terminal symbols
  - $P$ a finite set of productions
    - A subset of $N \times (N \cup \Sigma)^*$
  - $S$ the start symbol, a distinguished element of $N$
    - If not specified otherwise, this is usually assumed to be the non-terminal on the left of the first production

Standard Notations

- $a, b, c$ elements of $\Sigma$ (terminals)
- $w, x, y, z$ elements of $\Sigma^*$ (strings of terminals)
- $A, B, C$ elements of $N$ (non-terminals)
- $X, Y, Z$ elements of $N \cup \Sigma$ (grammar symbols)
- $\alpha, \beta, \gamma$ elements of $(N \cup \Sigma)^*$ (strings of symbols)

- $A \rightarrow \alpha$ or $A ::= \alpha$ if $<A, \alpha>$ in $P$
  - "non-terminal $A$ can take the form $\alpha$"

Derivation Relations

- $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ iff $A ::= \beta$ in $P$
  - "=>$" is read "derives"
- $A \Rightarrow^* w$ if there is a chain of productions starting with $A$ that generates $w$
  - transitive closure

Derivation Relations

- $w A \gamma \Rightarrow_{lm} w \beta \gamma$ iff $A ::= \beta$ in $P$
  - derives leftmost
- $\alpha A w \Rightarrow_{rm} \alpha \beta w$ iff $A ::= \beta$ in $P$
  - derives rightmost
- We will only be interested in leftmost and rightmost derivations – not random orderings
Languages

- For A in N, \( L(A) = \{ w \mid A \Rightarrow^* w \} \)
- If \( S \) is the start symbol of grammar \( G \), define \( L(G) = L(S) \)
  » The language derived by \( G \) is the language derived by the start symbol \( S \)

Reduced Grammars

- Grammar \( G \) is reduced iff for every production \( A ::= \alpha \) in \( G \) there is a derivation \( S \Rightarrow^* x A z \Rightarrow x \alpha z \Rightarrow^* xyz \)
  » i.e., no production is useless
- Convention: we will use only reduced grammars

Ambiguity

- Grammar \( G \) is unambiguous iff every \( w \) in \( L(G) \) has a unique leftmost (or rightmost) derivation
  » Fact: unique leftmost or unique rightmost implies the other
- A grammar without this property is ambiguous
  » Note that other grammars that generate the same language may be unambiguous
- We need unambiguous grammars for parsing

Ambiguous Grammar for Expressions

\[
\begin{align*}
expr & ::= expr + expr \mid expr - expr \\
& \quad \mid expr \ast expr \mid expr / expr \mid int \\
int & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

- Show that this is ambiguous
  » How? Show two different leftmost or rightmost derivations for the same string
  » Equivalently: show two different parse trees for the same string
Example Derivation

Give a leftmost derivation of 2+3*4 and show the parse tree

```
expr ::= expr + expr | expr - expr | expr * expr | expr / expr | int
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Another Derivation

Give a different leftmost derivation of 2+3*4 and show the parse tree

```
expr ::= expr + expr | expr - expr | expr * expr | expr / expr | int
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Another Example

Give two different derivations of 5+6+7

```
expr ::= expr + expr | expr - expr | expr * expr | expr / expr | int
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

What’s going on here?

- The grammar has no notion of precedence or associativity
- Solution
  - Create a non-terminal for each level of precedence
  - Isolate the corresponding part of the grammar
  - Force the parser to recognize higher precedence subexpressions first
Classic Expression Grammar

\[ expr ::= expr + term \mid expr - term \mid term \]
\[ term ::= term * factor \mid term / factor \mid factor \]
\[ factor ::= int \mid ( expr ) \]
\[ int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \]

Derive 2 + 3 * 4

Derive 5 + 6 + 7

Derive 5 + (6 + 7)
Another Classic Example

• Grammar for conditional statements

\[ \text{ifStmt} ::= \text{if} \ (\text{cond}) \ \text{stmt} \]
\[ \text{ifStmt} ::= \text{if} \ (\text{cond}) \ \text{stmt} \ \text{else} \ \text{stmt} \]

» Exercise: show that this is ambiguous
   How?

One Derivation

\[ \text{ifStmt} ::= \text{if} \ (\text{cond}) \ \text{stmt} \]
\[ \text{ifStmt} ::= \text{if} \ (\text{cond}) \ \text{stmt} \ \text{else} \ \text{stmt} \]

Solving \textbf{if} Ambiguity

• Fix the grammar to separate if statements with else clause and if statements with no else
   » Done in Java reference grammar
   » Adds lots of non-terminals
• Use some ad-hoc rule in parser
   » “else matches closest unpaired if”