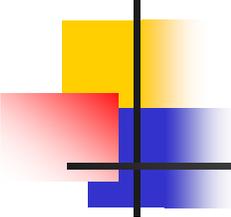


CSE 401 – Compilers

Parsing & Context-Free Grammars

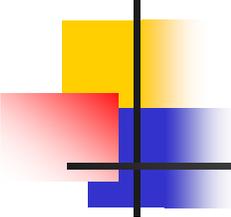
Hal Perkins

Winter 2009



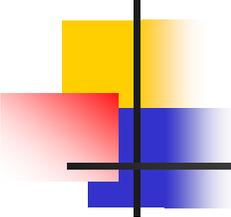
Agenda for Today

- Parsing overview
- Context free grammars
- Ambiguous grammars
- Reading: Cooper & Torczon 3.1-3.2



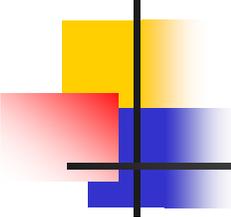
Syntactic Analysis / Parsing

- Goal: Convert token stream to **abstract syntax tree**
- Abstract syntax tree (AST):
 - Captures the structural features of the program
 - Primary data structure for remainder of compilation
- Three Part Plan
 - Study how context-free grammars specify syntax
 - Study algorithms for parsing / building ASTs
 - Study the miniJava Implementation



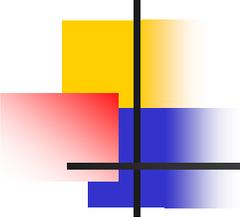
Context-free Grammars

- The syntax of most programming languages can be specified by a *context-free grammar* (CFG)
- Compromise between
 - REs, can't nest or specify recursive structure
 - General grammars, too powerful, undecidable
- Context-free grammars are a sweet spot
 - Powerful enough to describe nesting, recursion
 - Easy to parse; but also allow restrictions for speed
- Not perfect
 - Cannot capture semantics, as in, "variable must be declared," requiring later semantic pass
 - Can be ambiguous
- EBNF, Extended Backus Naur Form, is popular notation



Derivations and Parse Trees

- **Derivation**: a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals
- **Parsing**: inverse of derivation
 - Given a sequence of terminals ($a_1 \dots a_n$ tokens) want to recover the nonterminals representing structure
- Can represent derivation as a parse tree, that is, the concrete syntax tree



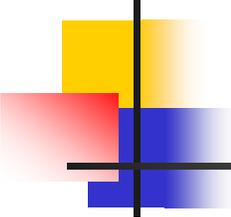
Example Derivation

G

```
program ::= statement | program statement  
statement ::= assignStmt | ifStmt  
assignStmt ::= id = expr ;  
ifStmt ::= if ( expr ) stmt  
expr ::= id | int | expr + expr  
Id ::= a | b | c | i | j | k | n | x | y | z  
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

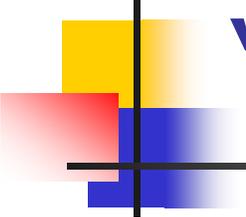
program

w → a = 1 ; if (a + 1) b = 2 ;



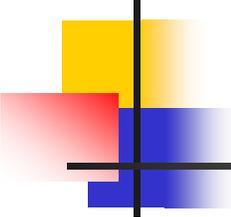
Parsing

- 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



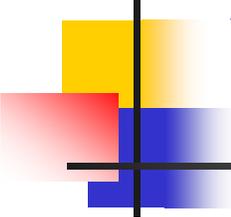
“Standard Order”

- For practical reasons we want the parser to be *deterministic* (no backtracking), and we want to examine the source program from *left to right*.
 - (i.e., parse the program in linear time in the order it appears in the source file)



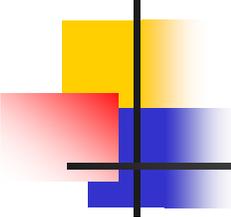
Common Orderings

- 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.)



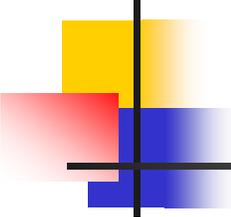
“Something Useful”

- 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 – but great if you need a quick ‘n dirty working compiler)



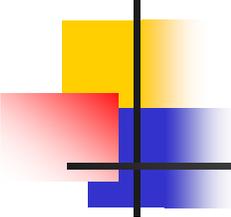
Context-Free Grammars

- Formally, a grammar G is a tuple $\langle N, \Sigma, P, S \rangle$ where
 - N a finite set of non-terminal symbols
 - Σ 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



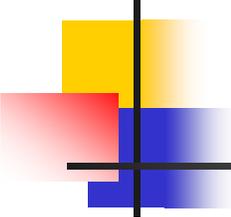
Standard Notations

- a, b, c elements of Σ
- w, x, y, z elements of Σ^*
- A, B, C elements of N
- X, Y, Z elements of $N \cup \Sigma$
- α, β, γ elements of $(N \cup \Sigma)^*$
- $A \rightarrow \alpha$ or $A ::= \alpha$ if $\langle A, \alpha \rangle$ in P



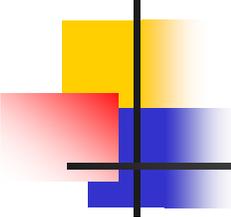
Derivation Relations (1)

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



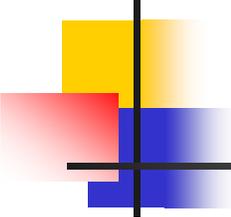
Derivation Relations (2)

- $w A \gamma \Rightarrow_{lm} w \beta \gamma$ iff $A ::= \beta$ in \mathcal{P}
 - derives **leftmost**
- $\alpha A w \Rightarrow_{rm} \alpha \beta w$ iff $A ::= \beta$ in \mathcal{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 $L(G) = L(S)$

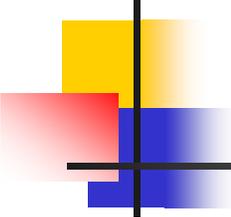


Reduced Grammars

- Grammar G is *reduced* iff for every production $A ::= \alpha$ in G there is a derivation

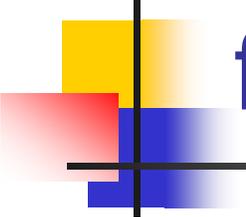
$$S \Rightarrow^* x A z \Rightarrow x \alpha z \Rightarrow^* xyz$$

- 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

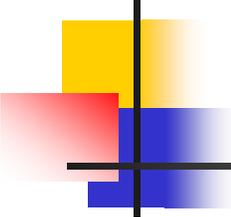


Example: Ambiguous Grammar for Arithmetic Expressions

$$\begin{aligned} \text{expr} ::= & \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \\ & \mid \text{expr} * \text{expr} \mid \text{expr} / \text{expr} \mid \text{int} \\ \text{int} ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

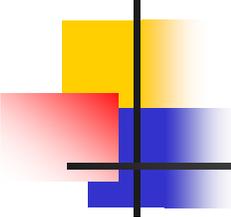
- Exercise: show that this is ambiguous
 - How? Show two different leftmost or rightmost derivations for the same string
 - Equivalently: show two different parse trees for the same string

$expr ::= expr + expr \mid expr - expr$
 $\mid expr * expr \mid expr / expr \mid int$
 $int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



Example (cont)

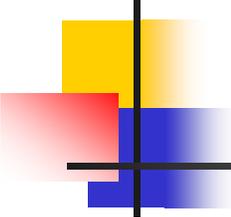
- Give a leftmost derivation of $2+3*4$ and show the parse tree

$$\begin{aligned} \text{expr} ::= & \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \\ & \mid \text{expr} * \text{expr} \mid \text{expr} / \text{expr} \mid \text{int} \\ \text{int} ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$


Example (cont)

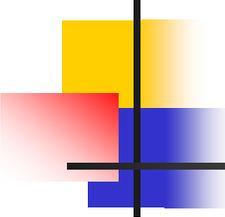
- Give a different leftmost derivation of $2+3*4$ and show the parse tree

$expr ::= expr + expr \mid expr - expr$
 $\mid expr * expr \mid expr / expr \mid int$
 $int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



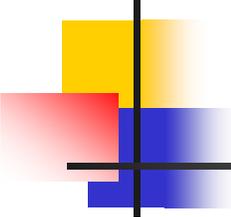
Another example

- Give two different derivations of $5+6+7$



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
 - Use left- or right-recursion for left- or right-associative operators (non-associative operators are not recursive)



Classic Expression Grammar

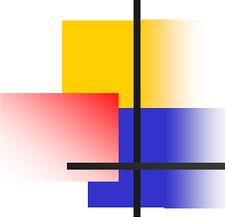
$expr ::= expr + term \mid expr - term \mid term$

$term ::= term * factor \mid term / factor \mid factor$

$factor ::= int \mid (expr)$

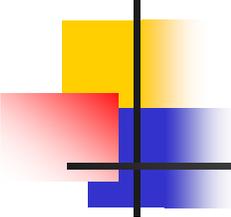
$int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$

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



Check: Derive $2 + 3 * 4$

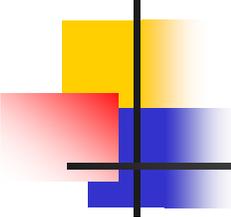
$expr ::= expr + term \mid expr - term \mid term$
 $term ::= term * factor \mid term / factor \mid factor$
 $factor ::= int \mid (expr)$
 $int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$



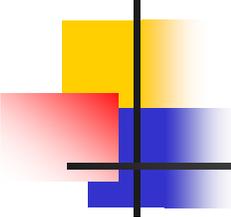
Check: Derive $5 + 6 + 7$

- Note interaction between left- vs right-recursive rules and resulting associativity

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



Check: Derive $5 + (6 + 7)$



Another Classic Example

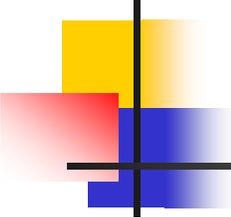
- Grammar for conditional statements

ifStmt ::= if (cond) stmt

| if (cond) stmt else stmt

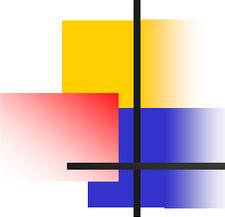
- Exercise: show that this is ambiguous
 - How?

$ifStmt ::= if (cond) stmt$
 $| if (cond) stmt else stmt$



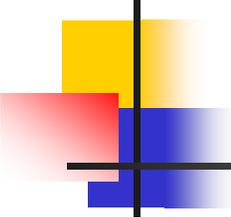
One Derivation

$if (cond) if (cond) stmt else stmt$

$$\begin{aligned} \text{ifStmt} ::= & \text{if (} \textit{cond} \text{) } \textit{stmt} \\ & | \text{if (} \textit{cond} \text{) } \textit{stmt} \text{ else } \textit{stmt} \end{aligned}$$


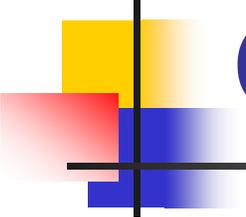
Another Derivation

$\text{if (} \textit{cond} \text{) } \text{if (} \textit{cond} \text{) } \textit{stmt} \text{ else } \textit{stmt}$



Solving “if” Ambiguity

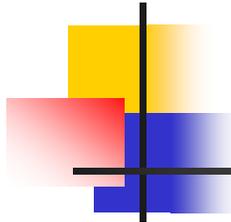
- Fix the grammar to separate if statements with else clause and if statements with no else
 - Done in Java reference grammar
 - Adds lots of non-terminals
- Change the language
 - But it'd better be ok to do this
- Use some ad-hoc rule in parser
 - “else matches closest unpaired if”



Resolving Ambiguity with Grammar (1)

```
Stmt      ::= MatchedStmt | UnmatchedStmt
MatchedStmt ::= ... |
              if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
                  if ( Expr ) MatchedStmt else UnmatchedStmt
```

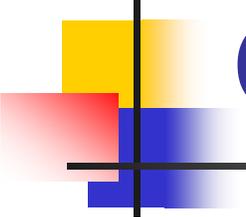
- formal, no additional rules beyond syntax
- sometimes obscures original grammar



Check

```
Stmt      ::= MatchedStmt | UnmatchedStmt
MatchedStmt ::= ... |
             if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
                  if ( Expr ) MatchedStmt else UnmatchedStmt
```

if (*cond*) if (*cond*) *stmt* else *stmt*

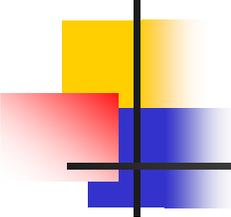


Resolving Ambiguity with Grammar (2)

- If you can (re-)design the language, avoid the problem entirely

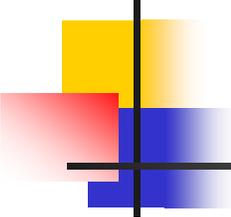
```
Stmt ::= ... |  
       if Expr then Stmt end |  
       if Expr then Stmt else Stmt end
```

- formal, clear, elegant
- allows sequence of Stmts in then and else branches, no { , } needed
- extra end required for every if
 - (But maybe this is a good idea anyway?)



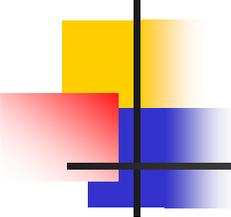
Parser Tools and Operators

- Most parser tools can cope with ambiguous grammars
 - Makes life simpler if used with discipline
- Typically one can specify operator precedence & associativity
 - Allows simpler, ambiguous grammar with fewer nonterminals as basis for generated parser, without creating problems



Parser Tools and Ambiguous Grammars

- Possible rules for resolving other problems
 - Earlier productions in the grammar preferred to later ones
 - Longest match used if there is a choice
- Parser tools normally allow for this
 - But be sure that what the tool does is really what you want



Coming Attractions

- Next topic: LR parsing
 - Continue reading ch. 3