CS 345

Lexical and Syntactic Analysis

Vitaly Shmatikov

Reading Assignment

- Mitchell, Chapters 4.1
- C Reference Manual, Chapters 2 and 7

Syntax

- Syntax of a programming language is a precise description of all grammatically correct programs
  - Precise formal syntax was first used in ALGOL 60
- Lexical syntax
  - Basic symbols (names, values, operators, etc.)
- Concrete syntax
  - Rules for writing expressions, statements, programs
- Abstract syntax
  - Internal representation of expressions and statements, capturing their “meaning” (i.e., semantics)

Grammars

- A meta-language is a language used to define other languages
- A grammar is a meta-language used to define the syntax of a language. It consists of:
  - Finite set of terminal symbols
  - Finite set of non-terminal symbols
  - Finite set of production rules
  - Start symbol

Backus-Naur Form (BNF)

  - Language = (possibly infinite) set of all sequences of symbols that can be derived by applying production rules starting from the start symbol

Example: Decimal Numbers

- Grammar for unsigned decimal integers
  - Terminal symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Non-terminal symbols: Digit, Integer
  - Production rules:
    - Integer → Digit | Integer Digit
    - Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - Start symbol: Integer

Shorthand for
Integer → Digit
Integer → Integer Digit

- Can derive any unsigned integer using this grammar
  - Language = set of all unsigned decimal integers

Derivation of 352 as an Integer

Production rules:

Integer → Digit | Integer Digit

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

Integer ⇒ Integer Digit
⇒ Integer 2
⇒ Integer Digit 2
⇒ Integer 5 2
⇒ Digit 5 2
⇒ 3 5 2

Rightmost derivation

At each step, the rightmost non-terminal is replaced

Leftmost Derivation

Production rules:

Integer → Digit | Integer Digit

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

Integer ⇒ Integer Digit

⇒ Integer Digit Digit

⇒ Digit Digit Digit

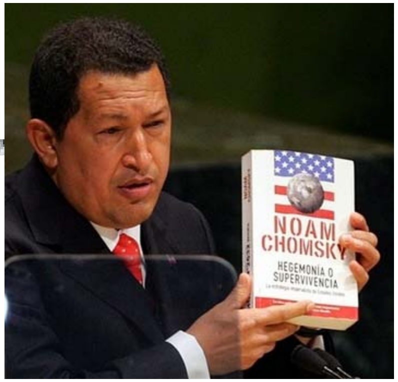
⇒ 3 Digit Digit

⇒ 3 5 Digit

⇒ 3 5 2

At each step, the leftmost non-terminal is replaced

Chomsky Hierarchy



- Regular grammars
  - Regular expressions, finite-state automata
  - Used to define lexical structure of the language
- Context-free grammars
  - Non-deterministic pushdown automata
  - Used to define concrete syntax of the language
- Context-sensitive grammars
- Unrestricted grammars
  - Recursively enumerable languages, Turing machines

Regular Grammars

- Left regular grammar
  - All production rules have the form
    $A \rightarrow \omega$ or $A \rightarrow B\omega$
    - Here A, B are non-terminal symbols, ω is a terminal symbol
- Right regular grammar
  - $A \rightarrow \omega$ or $A \rightarrow \omega B$
- Example: grammar of decimal integers
- Not a regular language: $\{a^n\ b^n \mid n \geq 1\}$ (why?)
- What about this: "any sequence of integers where (is eventually followed by)"?

Lexical Analysis

- Source code = long string of ASCII characters
- Lexical analyzer splits it into tokens
  - Token = sequence of characters (symbolic name) representing a single terminal symbol
- Identifiers: myVariable …
- Literals: 123 5.67 true …
- Keywords: char sizeof …
- Operators: + - * / …
- Punctuation: ; , } { …
- Discards whitespace and comments

Regular Expressions

x	character x
\x	escaped character, e.g., \n
{ name }	reference to a name
M | N	M or N
M N	M followed by N
M*	0 or more occurrences of M
M+	1 or more occurrences of M
[x_1 … x_n]	One of x_1 … x_n

- Example: [aeiou] – vowels, [0-9] - digits

Examples of Tokens in C

- Lexical analyzer usually represents each token by a unique integer code
  - "+" { return(PLUS); } // PLUS = 401
  - "-" { return(MINUS); } // MINUS = 402
  - "*" { return(MULT); } // MULT = 403
  - "/" { return(DIV); } // DIV = 404
- Some tokens require regular expressions
  - [a-zA-Z_][a-zA-Z0-9_]* { return (ID); } // identifier
  - [1-9][0-9]* { return(DECIMALINT); }
  - 0[0-7]* { return(OCTALINT); }
  - (0x|0X)[0-9a-fA-F]+ { return(HEXINT); }

Reserved Keywords in C

- auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, wchar_t, while
- C++ added a bunch: bool, catch, class, dynamic_cast, inline, private, protected, public, static_cast, template, this, virtual and others
- Each keyword is mapped to its own token

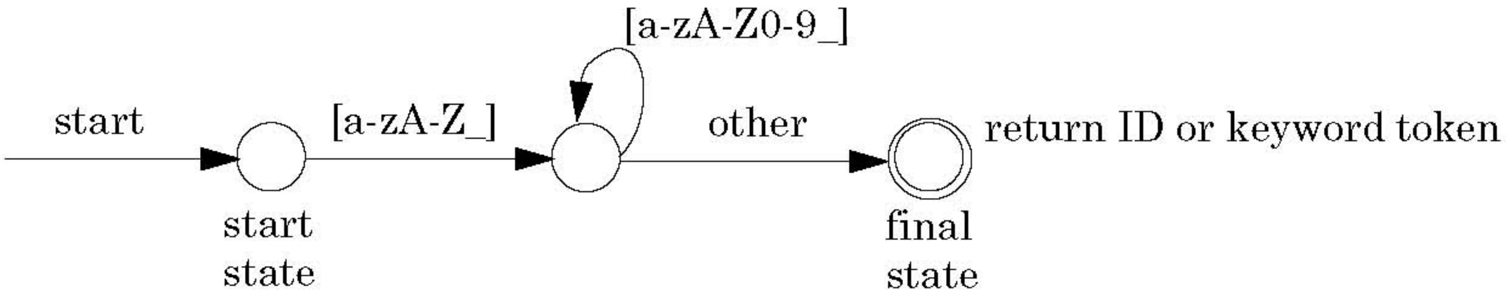
Automatic Scanner Generation

- Lexer or scanner recognizes and separates lexical tokens
  - Parser usually calls lexer when it's ready to process the next symbol (lexer remembers where it left off)
- Scanner code usually generated automatically
  - Input: lexical definition (e.g., regular expressions)
  - Output: code implementing the scanner
    - Typically, this is a deterministic finite automaton (DFA)
  - Examples: Lex, Flex (C and C++), JLex (Java)

Finite State Automata

- Set of states
  - Usually represented as graph nodes
- Input alphabet + unique "end of program" symbol
- State transition function
  - Usually represented as directed graph edges (arcs)
  - Automaton is deterministic if, for each state and each input symbol, there is at most one outgoing arc from the state labeled with the input symbol
- Unique start state
- One or more final (accepting) states

DFA for C Identifiers



Traversing a DFA

- Configuration = state + remaining input
- Move = traversing the arc exiting the state that corresponds to the leftmost input symbol, thereby consuming it
- If no such arc, then…
  - If no input and state is final, then accept
  - Otherwise, error
- Input is accepted if, starting with the start state, the automaton consumes all the input and halts in a final state

Context-Free Grammars

- Used to describe concrete syntax
  - Typically using BNF notation
- Production rules have the form A → ω
  - A is a non-terminal symbol, ω is a string of terminal and non-terminal symbols
- Parse tree = graphical representation of derivation
  - Each internal node = LHS of a production rule
    - Internal node must be a non-terminal symbol (why?)
  - Children nodes = RHS of this production rule
  - Each leaf node = terminal symbol (token) or "empty"

Syntactic Correctness

- Lexical analyzer produces a stream of tokens
- Parser (syntactic analyzer) verifies that this token stream is syntactically correct by constructing a valid parse tree for the entire program
  - Unique parse tree for each language construct
  - Program = collection of parse trees rooted at the top by a special start symbol
- Parser can be built automatically from the BNF description of the language's CFG
  - Example tools: yacc, Bison

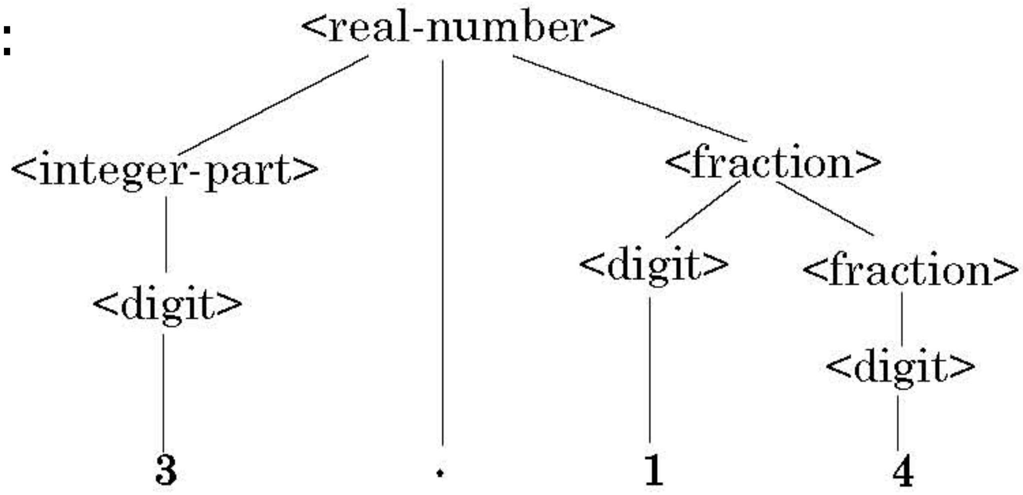
CFG For Floating Point Numbers

```
<real-number> ::= <integer-part> `.' <fraction-part>
<integer-part> ::= <digit> | <integer-part> <digit>
<fraction> ::= <digit> | <digit> <fraction>
<digit> ::= `0' | `1' | `2' | `3' | `4' | `5' | `6' | `7' | `8' | `9'
```

::= stands for production rule; <…> are non-terminals;

| represents alternatives for the right-hand side of a production rule

Sample parse tree:



CFG For Balanced Parentheses

```
<balanced> ::= ( <balanced> ) | <empty>
```

Could we write this grammar using regular expressions or DFA? Why?

Sample derivation: <balanced> ⇒ (<balanced>)
⇒ ((<balanced>))
⇒ ((<empty>))
⇒ (( ))

CFG For Decimal Numbers (Redux)

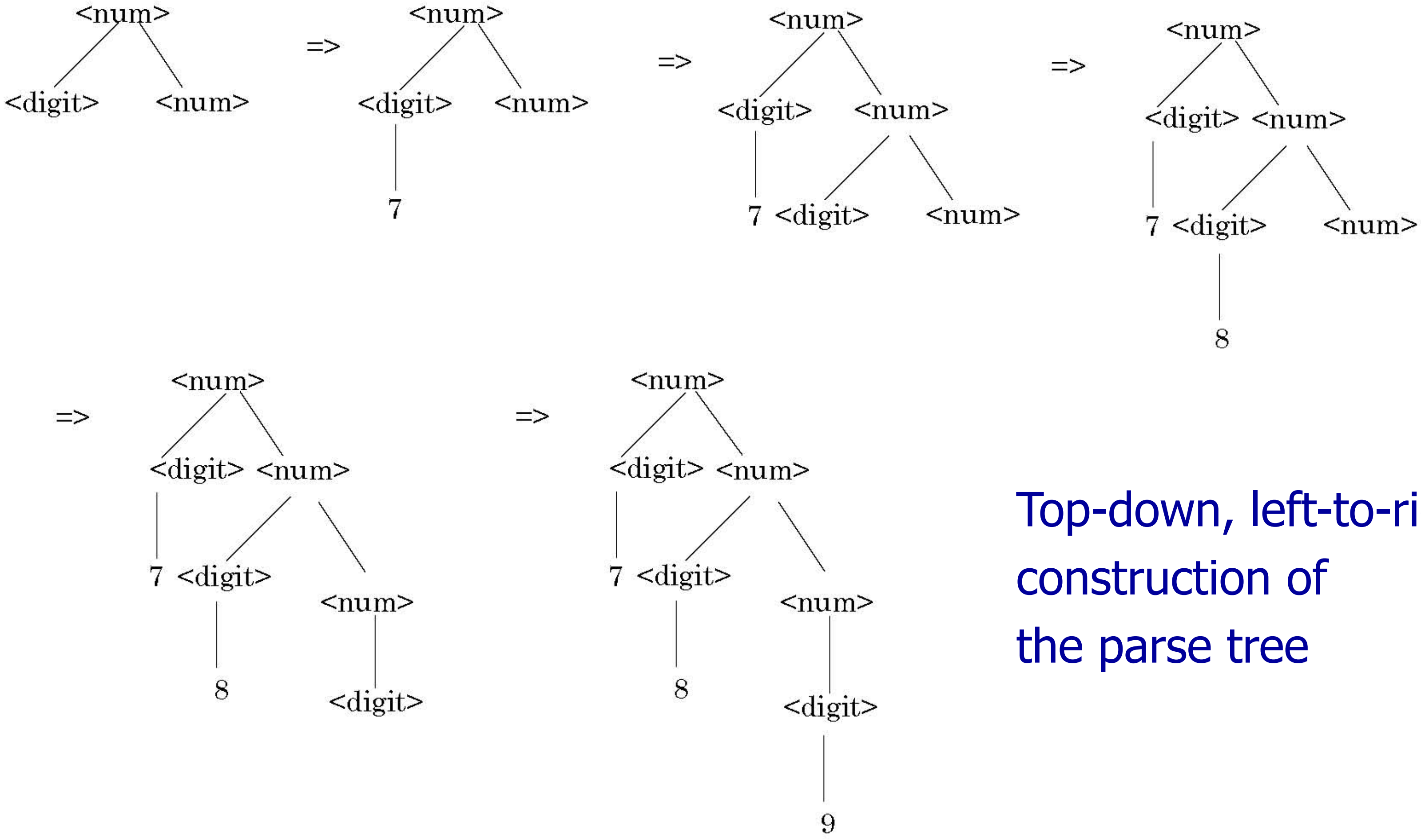
```
<num> ::= <digit> | <digit> <num>
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

This grammar is **right-recursive**

Sample
top-down leftmost
derivation:

<num> ⇒ <digit> <num>
⇒ 7 <num>
⇒ 7 <digit> <num>
⇒ 7 8 <num>
⇒ 7 8 <digit>
⇒ 7 8 9

Recursive Descent Parsing



Top-down, left-to-right construction of the parse tree

Shift-Reduce Parsing

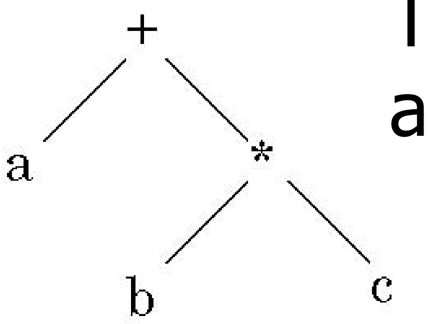
- Idea: build the parse tree bottom-up
  - Lexer supplies a token, parser find production rule with matching right-hand side (i.e., run rules in reverse)
  - If start symbol is reached, parsing is successful

789 ⇒ 7 8 <digit>
reduce ⇒ 7 8 <num>
shift ⇒ 7 <digit> <num>
reduce ⇒ 7 <num>
shift ⇒ <digit> <num>
reduce ⇒ <num>

Production rules:
Num → Digit | Digit Num
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Concrete vs. Abstract Syntax

- Different languages have different concrete syntax for representing expressions, but expressions with common meaning have the same abstract syntax
  - C: a+b*c   Forth: bc*a+ (reverse Polish notation)



This expression tree represents the abstract “meaning” of expression

- Assumes certain operator precedence (why?)
- Not the same as parse tree (why?)
- Does the value depend on traversal order?

Expression Notation

Inorder traversal (3+4)*5=35 3+(4*5)=23

When constructing expression trees, we want inorder traversal to produce correct arithmetic result based on operator precedence and associativity

Postorder traversal 3 4 + 5 * =35 3 4 5 * + =23

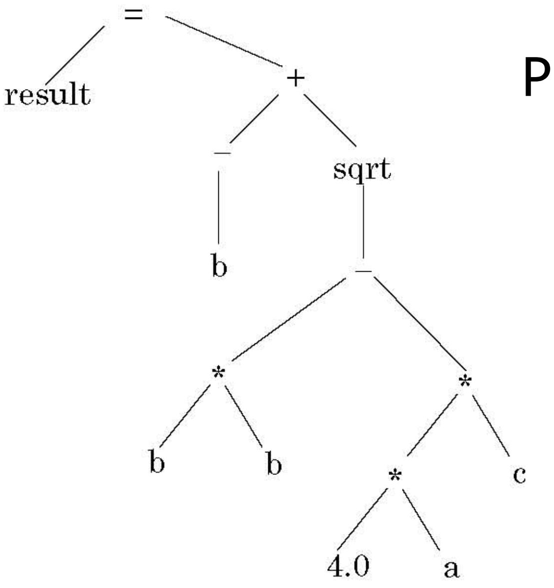
Easily evaluated using operand stack (example: Forth)

- Leaf node: push operand value on the stack
- Non-leaf binary or unary operator: pop two (resp. one) values from stack, apply operator, push result back on the stack
- End of evaluation: print top of the stack

Mixed Expression Notation

```
result = -b + sqrt(b*b - 4.0 * a * c);
```

unary prefix operators



Prefix:

: result + $-_1$ b sqrt $-_2$ * b b * * 4.0 a c

Need to indicate arity to distinguish between unary and binary minus

Postfix, Prefix, Mixfix in Java and C

- Increment and decrement: `x++`, `--y`
  - `x = ++x + x++`   legal syntax, undefined semantics!
- Ternary conditional
  - `(conditional-expr) ? (then-expr) : (else-expr);`
  - Example:
    `int min(int a, int b) { return (a<b) ? a : b; }`
  - This is an <u>expression</u>, NOT an if-then-else command
  - What is the type of this expression?

Expression Compilation Example

```
float position, initial, rate;
position = initial + rate * 60;
```

⇩ lexical analyzer

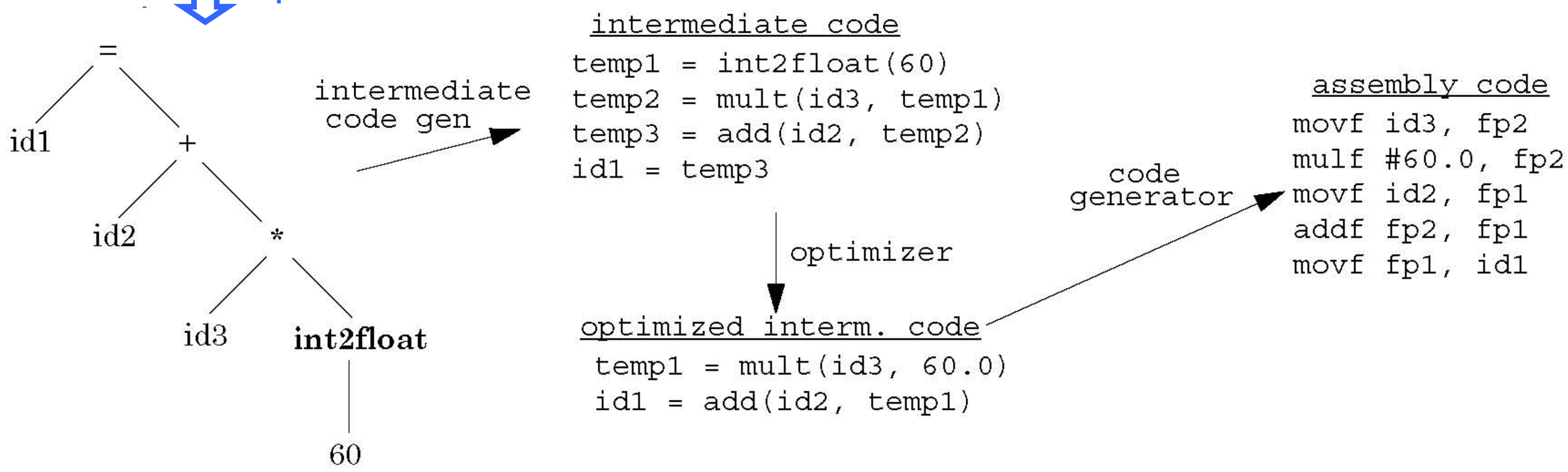
```
[ID, "position"] [ASSIGN, '='] [ID, "initial"] [PLUS, '+'] [ID, "rate"] [MULT, '*'] [NUM, 60] [SEMICOLON,';']
```

tokenized expression:

```
id1 = id2 + id3 * 60
```

implicit type conversion (why?)

⇩ parser



intermediate code gen

intermediate code

```
temp1 = int2float(60)
temp2 = mult(id3, temp1)
temp3 = add(id2, temp2)
id1 = temp3
```

optimizer

optimized interm. code

```
temp1 = mult(id3, 60.0)
id1 = add(id2, temp1)
```

code generator

assembly code

```
movf id3, fp2
mulf #60.0, fp2
movf id2, fp1
addf fp2, fp1
movf fp1, id1
```

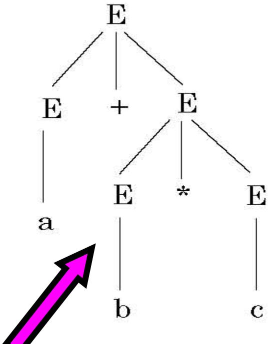
Syntactic Ambiguity

```
<expr> ::= <expr> + <expr> | <expr> * <expr> | a | b | c
```

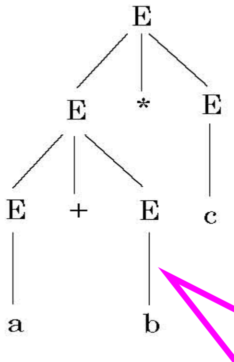
This grammar is **ambiguous**

How to parse a+b*c using this grammar?

Parse Tree from a rightmost derivation
starting from <expr> + <expr>



Parse Tree from a leftmost derivation
starting with <expr> * <expr>



Both parse trees are syntactically valid

Only this tree is semantically correct (operator precedence and associativity are semantic, not syntactic rules)

Problem: this tree is syntactically correct, but semantically incorrect

Removing Ambiguity

Not always possible to remove ambiguity this way!

- Define a distinct non-terminal symbol for each operator precedence level
- Define RHS of production rule to enforce proper associativity
- Extra non-terminal for smallest subexpressions

```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= ( E ) | id | num
```

This Grammar Is Unambiguous

```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= ( E ) | id | num
```

Leftmost:

E => $\underline{\text{E}}$ + T
=> $\underline{\text{T}}$ + T
=> $\underline{\text{F}}$ + T
=> id + $\underline{\text{T}}$
=> id + $\underline{\text{T}}$ * F
=> id + $\underline{\text{F}}$ * F
=> id + id * F
=> id + id * id

Rightmost:

E => E + $\underline{\text{T}}$
=> E + T * $\underline{\text{F}}$
=> E + $\underline{\text{T}}$ * id
=> E + $\underline{\text{F}}$ * id
=> $\underline{\text{E}}$ + id * id
=> $\underline{\text{T}}$ + id * id
=> $\underline{\text{F}}$ + id * id
=> id + id * id

Left- and Right-Recursive Grammars

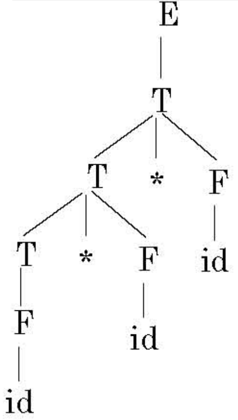
```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= ( E ) | id | num
```

Leftmost non-terminal on the RHS of production is the same as the LHS

```
E ::= T + E | T - E | T
T ::= F * T | F / T | F
F ::= ( E ) | id | num
```

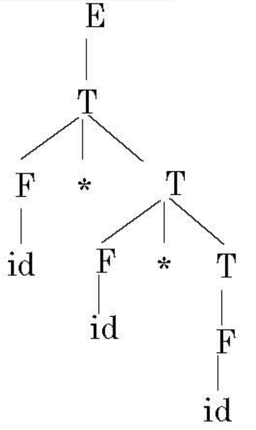
Right-recursive grammar

Left recursive parse tree for id * id * id with left-associativity



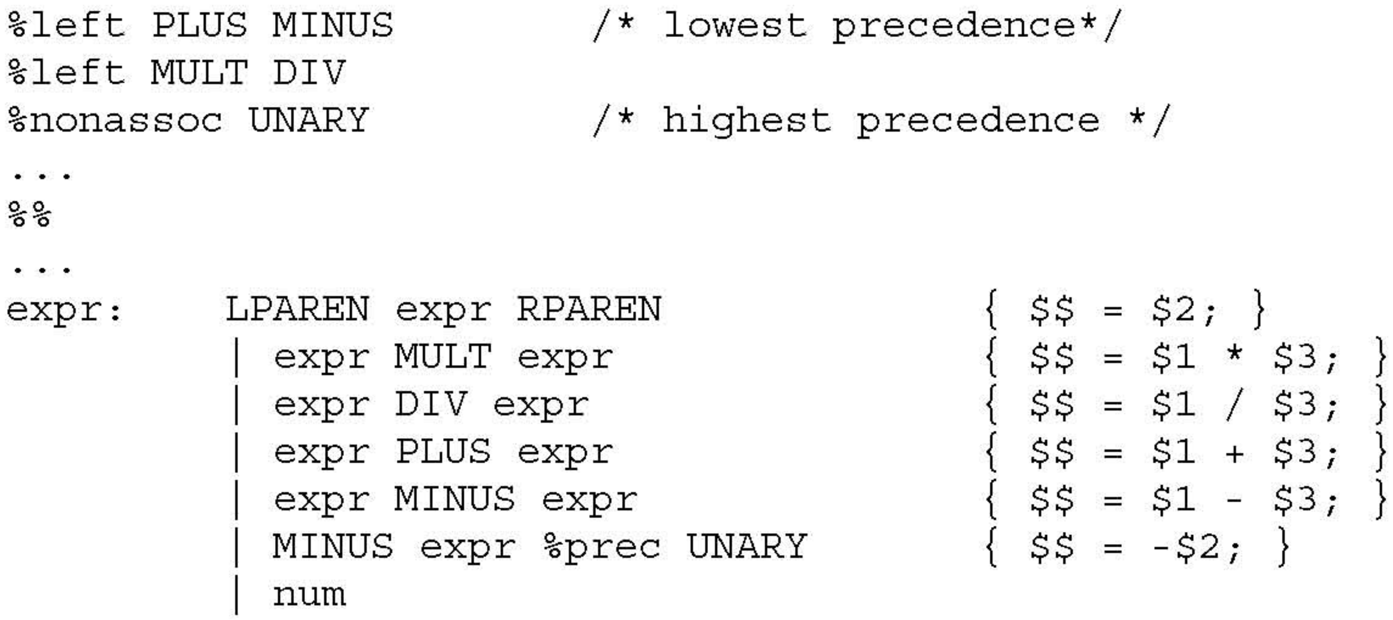
Can you think of any operators that are right-associative?

Right recursive parse tree for id * id * id with right associativity



Yacc Expression Grammar

- Yacc: automatic parser generator
- Explicit specification of operator precedence and associativity (don’t need to rewrite grammar)



```
%left PLUS MINUS          /* lowest precedence*/
%left MULT DIV
%nonassoc UNARY           /* highest precedence */
...
%%
...
expr:    LPAREN expr RPAREN             { $$ = $2; }
         | expr MULT expr               { $$ = $1 * $3; }
         | expr DIV expr                { $$ = $1 / $3; }
         | expr PLUS expr               { $$ = $1 + $3; }
         | expr MINUS expr              { $$ = $1 - $3; }
         | MINUS expr %prec UNARY       { $$ = -$2; }
         | num
```

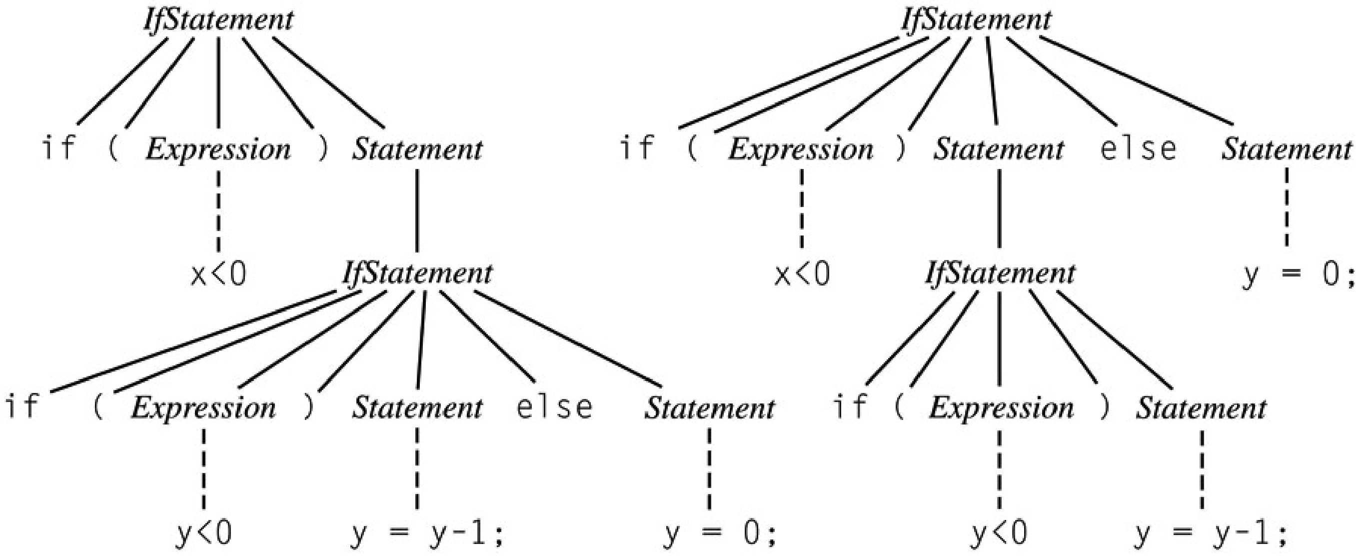
“Dangling Else” Ambiguity

```
stmt ::= if (expr) then stmt | if (expr) then stmt else stmt
```

With which if does this else associate?

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

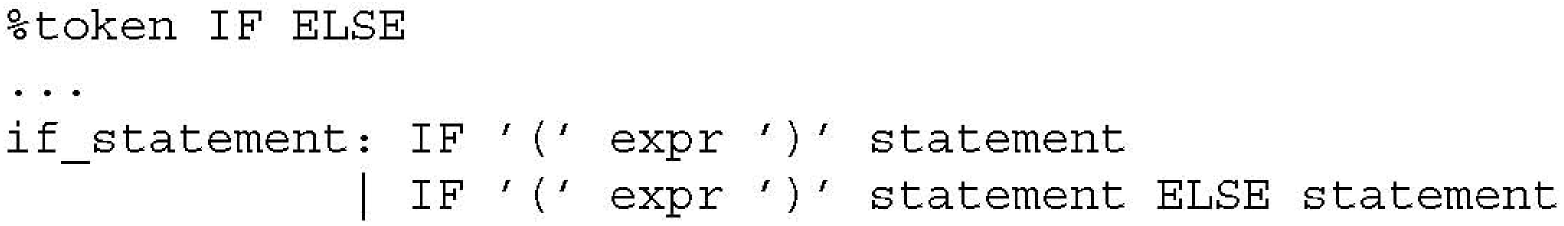
Classic example of a **shift-reduce conflict**



Solving the Dangling Else Ambiguity

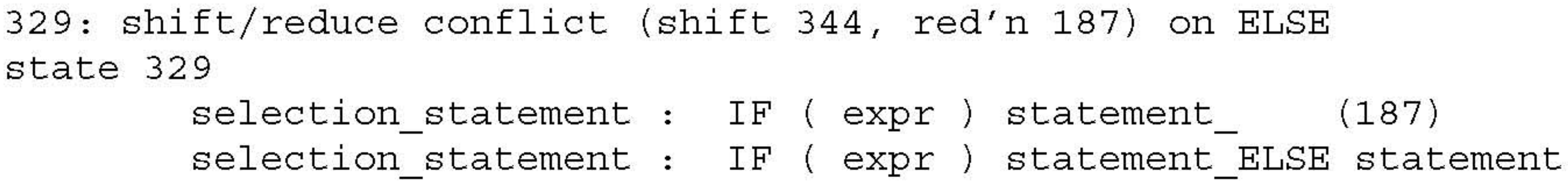
- Algol 60, C, C++: associate each else with closest if; use { … } or begin … end to override
  - Does this prefer "shift" to "reduce" or vice versa?
- Algol 68, Modula, Ada: use an explicit delimiter to end every conditional (e.g., if … endif)
- Java: rewrite the grammar and restrict what can appear inside a nested if statement
  - IfThenStmt → if (Expr) Stmt
  - IfThenElseStmt → if (Expr) StmtNoShortIf else Stmt
    - The category StmtNoShortIf includes all except IfThenStmt

Shift-Reduce Conflicts in Yacc



```
%token IF ELSE
...
if_statement: IF '(' expr ')' statement
            | IF '(' expr ')' statement ELSE statement
```

- ◆ This grammar is ambiguous!
- ◆ By default, Yacc shifts (i.e., pushes the token onto the parser's stack) and generates warning
  - Equivalent to associating "else" with closest "if" (this is correct semantics!)



```
329: shift/reduce conflict (shift 344, red'n 187) on ELSE
state 329
        selection_statement :  IF ( expr ) statement_    (187)
        selection_statement :  IF ( expr ) statement_ELSE statement
```

Avoiding Yacc Warning

```
%token IF ELSE
...
%nonassoc LOWER_THAN_ELSE  /* dummy token */
%nonassoc ELSE
...
%%
...
if_statement: IF ’(’ expr ’)’ statement %prec LOWER_THAN_ELSE
            | IF ’(’ expr ’)’ statement ELSE statement
```

Forces parser to shift ELSE onto the stack because it has higher precedence than dummy LOWER_THAN_ELSE token

More Powerful Grammars

- **Context-sensitive:** production rules have the form $\alpha A \beta \rightarrow \alpha \omega \beta$
  - A is a non-terminal symbol, α,β,ω are strings of terminal and non-terminal symbols
  - Deciding whether a string belongs to a language generated by a context-sensitive grammar is PSPACE-complete
  - Emptiness of a language is undecidable
    - What does this mean?
- **Unrestricted:** equivalent to Turing machine