Lexical and Syntactic Analysis

Course Plan

- Instructor: Prakash Raghavendra
- Lecturers:
  - Friday: 2 hours
  - Saturday: 2 sessions of 2 hours each
  - Two weeks a month
- Evaluation:
  - One assignment (20 marks)
  - One mid-sem (30 marks)
  - One end-sem (50 marks)

Syllabus

Lecture Series (hours)	Topics
1-4	Introduction and Motivation, Paradigms
5-10	Syntax and Semantics, BNF, Compilation
11-18	Data Types, Constructs, Functions, Activation Records, Names and Bindings
19-28	Functional PLs, Logical PLs, Lambda Calculus, Event driven programming, Concurrency
29-36	Virtual Machines, Managed Languages, JIT, Case study

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)

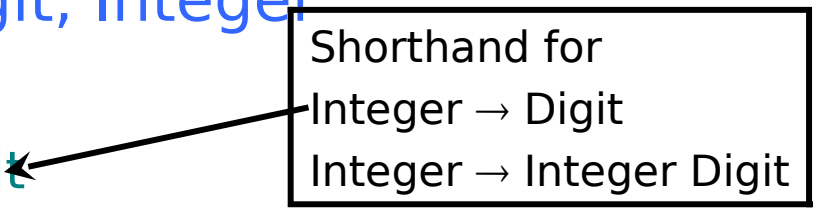
Grammars

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

Backus-Naur Form (BNF)

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
- 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

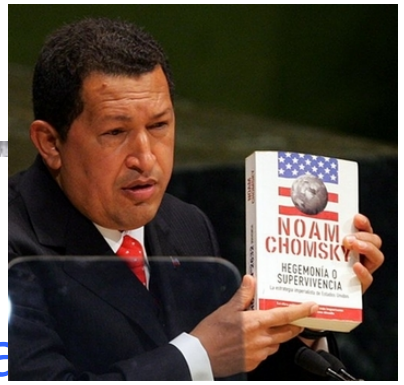
⇒ 3 Digit Digit

⇒ 3 5 Digit

⇒ 3 5 2

At each step, the leftmost non-terminal is replaced

Chomsky Hierarchy



- Regular grammars
  - Regular expressions, finite-state automa
  - 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

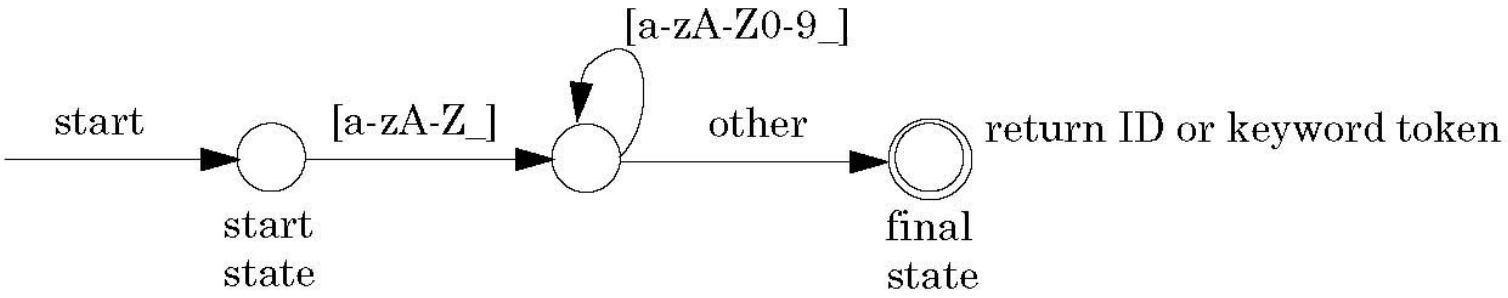
Automatic Scanner Generation

- 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 $A \rightarrow \omega$
  - 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

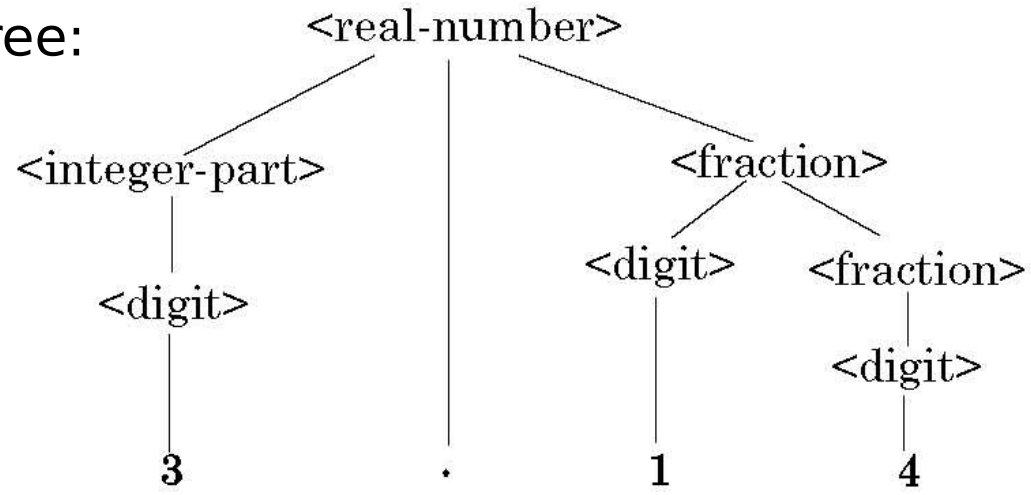
CFG For Floating Point Numbers

```
<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

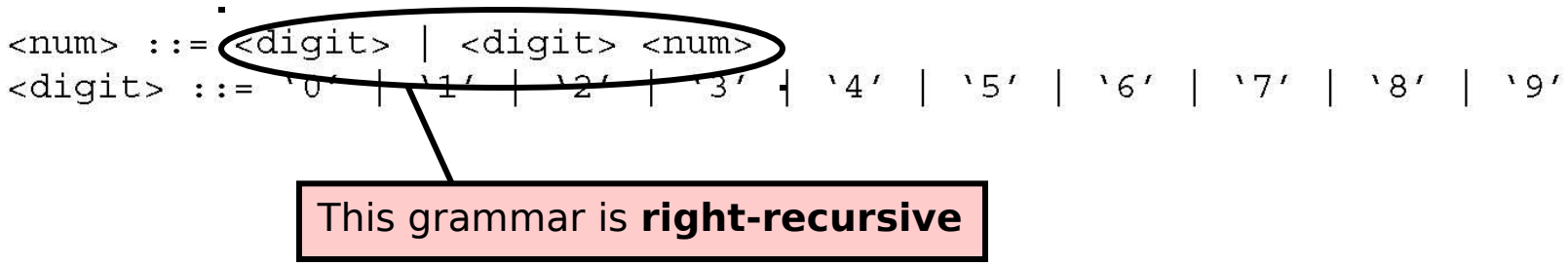
<balanced> ::= (<balanced>) | <empty>

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

Sample derivation:

<balanced> ⇒ (<balanced>)
⇒ ((<balanced>))
⇒ ((<empty>))
⇒ (( ))

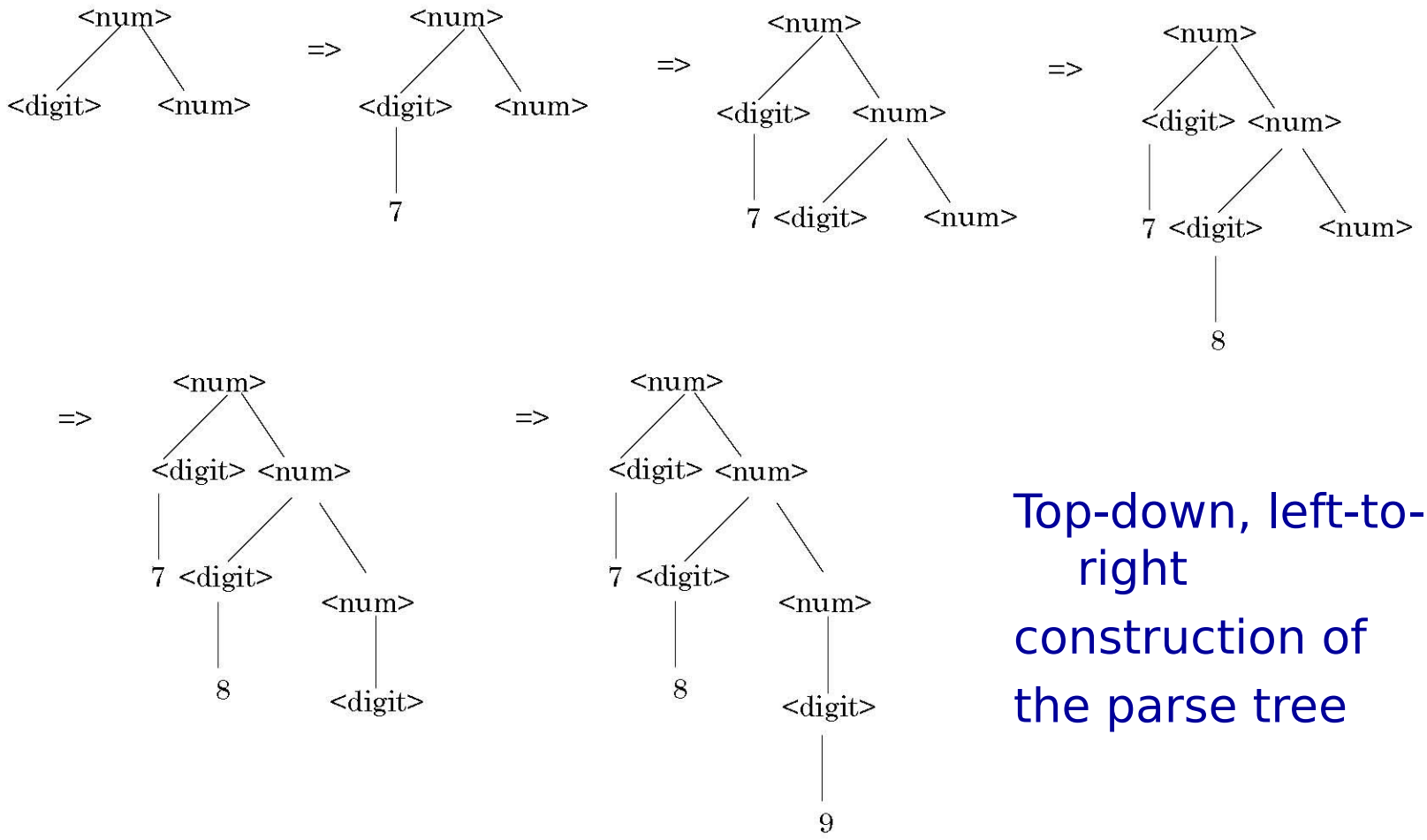
CFG For Decimal Numbers (Redux)



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

Left-Recursive Grammars

- Here is an example of a (directly) left-recursive grammar:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

- This grammar can be re-written as the following non left-recursive grammar:

$$E \rightarrow T\ E' \qquad E' \rightarrow + TE' \mid \epsilon$$
$$T \rightarrow F\ T' \qquad T' \rightarrow * F\ T' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$

Predictive Parser: Details

- The key problem during predictive parsing is that of determining the production to be applied for a non-terminal.
- This is done by using a parsing table.
- A parsing table is a two-dimensional array M[A,a] where A is a non-terminal, and a is a terminal or the symbol $, menaing “end of input string”.
- The other inputs of a predictive parser are:
  - The input buffer, which contains the string to be parsed followed by $.
  - The stack which contains a sequence of grammar symbols with, initially, $S (end of input string and start symbol) in it.

Predictive Parser:

- The predictive parser considers X, the symbol on top of the stack, and a, the current input symbol. It uses, M, the parsing table.
  - If X=a=$ ➔ halt and return success
  - If X=a≠$ ➔ pop X off the stack and advance input pointer to the next symbol
  - If X is a non-terminal ➔ Check M[X,a]
    - If the entry is a production rule, then replace X on the stack by the Right Hand Side of the production
    - If the entry is blank, then halt and return failure

Predictive Parser: An Example

	id	+	*	(	)	$
E	E→ TE’			E→ TE’		
E’		E’ →+ TE’			E’ →ϵ	E’ →ϵ
T	T→ FT’			T→ FT’		
T’		T’ →ϵ	T’ →* FT’		T’ →ϵ	T’ →ϵ
F	F→ id			F→ (E)		

Parsing Table

Stack	Input	Output
$E	id+id*id$	
$E’T	id+id*id$	E → TE’
$E’T’F	id+id*id$	T → FT’
$E’T’id	id+id*id$	F → id
$E’T’	+id*id$	
$E’	+id*id$	T’ → ϵ
$E’T+	+id*id$	E’ → +TE’
$E’T	id*id$	
$E’T’F	id*id$	T → FT’
$E’T’id	id*id$	F → id
$E’T’	*id$	
$E’T’F*	*id$	T’ → *FT’
$E’T’F	id$	
$E’T’id	id$	F → id
$E’T’	$	

Algorithm Trace →

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

LR/Shift-Reduce Parsing

- LR Parsers can recognize any language for which a context free grammar can be written.
- LR Parsing is the most general non-backtracking shift-reduce method known, yet it is as efficient as other shift-reduce approaches
- The class of grammars that can be parsed by an LR parser is a proper superset of that that can be parsed by a predictive parser.
- An LR-parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

LR-Parsing: Drawback/Solution

- The main drawback of LR parsing is that it is too much work to construct an LR parser by hand for a typical programming language grammar.
- Fortunately, specialized tools to construct LR parsers automatically have been designed.
- With such tools, a user can write a context-free grammar and have a parser generator automatically produce a parser for that grammar.
- An example of such a tool is Yacc “Yet Another Compiler-Compiler”

LR Parsing Algorithms: Details I

- An LR parser consists of an input, output, a stack, a driver program and a parsing table that has two parts: action and goto.
- The driver program is the same for all LR Parsers. Only the parsing table changes from one parser to the other.
- The program uses the stack to store a string of the form $s_0X_1s_1X_2...X_ms_m$, where s_m is the top of the stack. The S_k's are state symbols while the X_i's are grammar symbols. Together state and grammar symbols determine a shift-reduce parsing decision.

LR Parsing Algorithms: Details II

- The parsing table consists of two parts: a parsing *action* function and a *goto* function.
- The LR parsing program determines sm, the state on top of the stack and a_i, the current input. It then consults action[s_m, a_i] which can take one of four values:
  - Shift
  - Reduce
  - Accept
  - Error

LR Parsing Algorithms: Details III

- If action[s_m, a_i] = Shift s, where s is a state, then the parser pushes a_i and s on the stack.
- If action[s_m, a_i] = Reduce A → β, then a_i and s_m are replaced by A, and, if s was the state appearing below a_i in the stack, then goto[s, A] is consulted and the state it stores is pushed onto the stack.
- If action[s_m, a_i] = Accept, parsing is completed
- If action[s_m, a_i] = Error, then the parser discovered an error.

LR Parsing Example: The Grammar

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

LR-Parser Example: The Parsing Table

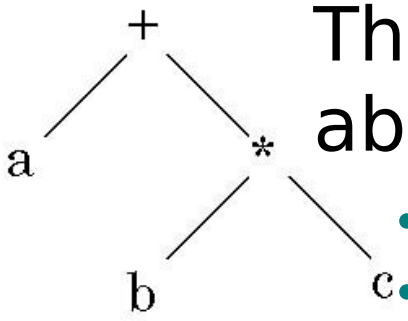
State	Action						Goto		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	s5			s4			1	2	3
1		s6				Acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		R1	r1			
10		r3	r3		r3	r3			

LR-Parser Example: Parsing Trace

Stack	Input	Action
(1) 0	id * id + id $	Shift
(2) 0 id 5	* id + id $	Reduce by F → id
(3) 0 F 3	* id + id $	Reduce by T → F
(4) 0 T 2	* id + id $	Shift
(5) 0 T 2 * 7	id + id $	Shift
(6) 0 T 2 * 7 id 5	+ id $	Reduce by F → id
(7) 0 T 2 * 7 F 10	+ id $	Reduce by T → T * F
(8) 0 T 2	+ id $	Reduce by E →T
(9) 0 E 1	+ id $	Shift
(10) 0 E 1 + 6	id $	Shift
(11) 0 E 1 + 6 id 5	$	Reduce by F → id
(12) 0 E 1 + 6 F 3	$	Reduce by T → F
(13) 0 E 1 + 6 T 9	$	E → E + T

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 Compilation Example

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

lexical analyzer

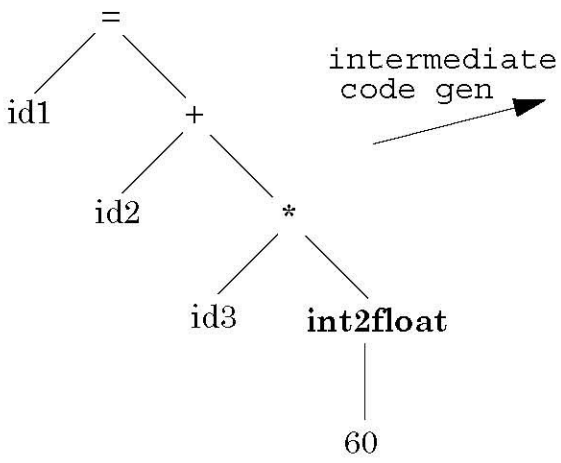
```
[ID, "position"] [ASSIGN, '='] [ID, "initial"] [PLUS, '+'] [ID, "rate"] [MULT, '*'] [NUM, 60] [SEMICOLON,';']
```

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
```

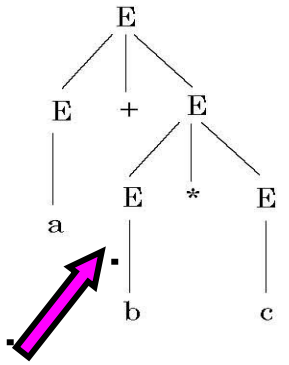
Syntactic Ambiguity

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

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

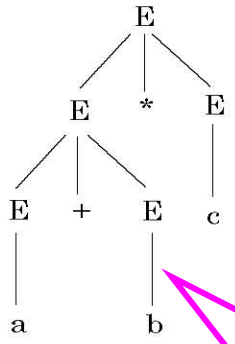
This grammar is **ambiguous**

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



Both parse trees are syntactically valid

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



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 => E + T
  => T + T
  => F + T
  => id + T
  => id + T * F
  => id + F * F
  => id + id * F
  => id + id * id
```

Rightmost:

```
E => E + T
  => E + T * F
  => E + T * id
  => E + F * id
  => E + id * id
  => T + id * id
  => F + id * id
  => id + id * id
```

Left- and Right-Recursive Grammars

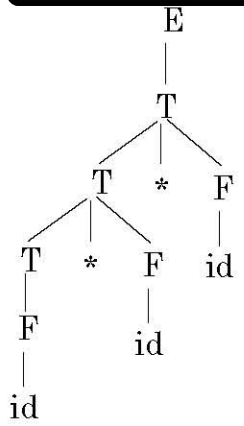
```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= ( E ) | id | num
```

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

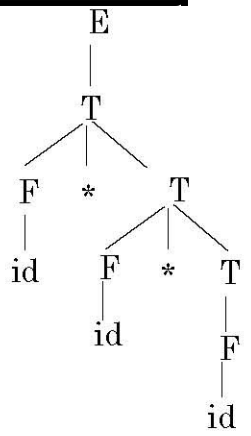
```
E ::= T + E | T - E | T
T ::= F * T | F / T | F
F ::= ( E ) | id | num
```

Right-recursive grammar

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



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



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

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
```

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

Assignment

- Implement a interpreter which accepts and evaluates the following CFG:

<expr> := [<op> <expr> <expr>] |
<symbol> | <value>
<symbol> := [a-zA-Z]+
<value> := [0-9]+
<op> := '+' | '*' | '==' | '<'

Assignment...2

<prog> := [= <symbol> <expr>] |
[; <prog> <prog>] |
[if <expr> <prog> <prog>] |
[while <expr> <prog>] |
[return <expr>]

Assignment....examples

- [* [+ 1 3] [+ 2 3]]

Should return (1 + 3) * (2 + 3) = 20

- [; [= x 5] [; if [< x 3] [= x [+ x 2]] [= x [* x 2]]] [return x]]

Should return 10

Assignment...3

- [; [= y [+ x 1]] [return y]]

What should happen?

- Implement either using the tools (yacc/lex) or by yourself
- Show the working during evaluation
- 4 in a group (~25 groups)
- Submit by Feb 28, 2018