CS3063 Theory of Computing

Semester 4 (21 Intake), Jan – May 2024

Lecture 9

Context-Free Languages: Session 4

Sanath Jayasena

Announcements

- Assignment 2 will be out within next few days
  - **Due end of April**

- Assignment 1
  - Marking being completed

Previous Lecture

- Context-free Languages – Session 3
  - Acceptance by PDA
  - Non-determinism in PDA
  - PDA for a CFG
    - Top-Down Approach
    - Bottom-up Approach

Today's Outline:

Lecture 9

Context-free Languages (CFLs) - 4

- **Conclusion of CFLs/PDAs**
- **Parsing in Compilers**
  - **Overview of compilation**
  - **Top-down parsing**
  - **Bottom-up parsing**

[References for parsing in compilers]

- pp. 280-290 of textbook (overview only)
- Any book on compilers

PART 1

Outline:

Lecture 9

Context-free Languages (CFLs) - 4

- Conclusion of CFLs/PDAs
- Parsing in Compilers
  - Overview of compilation
  - Top-down parsing
  - Bottom-up parsing

Pumping Lemma for CFLs

- Recall pumping lemma for regular lang.
  - If an FA accepts strings of the form $x=uvw$ and if the substring v causes a loop in the FA, then the FA accepts strings of the form uv^im
  - Can check whether a language is regular

- Similar idea with CFLs
  - Can check if a given language is a CFL

Pumping Lemma for CFLs ...contd

- Suppose a derivation in a CFG, G, is:

$$S \Rightarrow^* vAz \Rightarrow^* vwAyz \Rightarrow^* vwxyz$$

where v, w, x, y, z are terminals

  - Both x and wAy are derived from A
  - We can write

$$S \Rightarrow^* vAz \Rightarrow^* vwAyz \Rightarrow^* vw^2Ay^2z \Rightarrow^* \ldots$$

- Replacing A by x, get many strings in L(G)

Pumping Lemma for CFLs ...contd

- The Pumping Lemma for CFLs
  - Let ***L*** be a CFL. Then there is an integer ***n*** so that for any ***u*** in ***L*** satisfying $|u| \geq n$, there are strings ***v***, ***w***, ***x***, ***y*** and ***z*** satisfying

    $u = vwxyz$

    $|wy| > 0$

    $|wxy| \leq n$

    For any $m \geq 0$, vw^mxy^mz is in ***L***.

Pumping Lemma for CFLs ...contd

- Read more on pp. 297-303 in book
  - Discussion using derivation trees and CNF
- To show a language is not context-free
  - First assume it is context-free
  - Derive a contradiction (use pumping lemma)
- Examples
  - $\{ a^i b^i c^i \mid i \geq 1\}$, $\{ \{a,b,c\}^* \mid$ # a's, b's, c's equal$\}$
  - $\{ ss \mid s$ is in $\{a, b\}^*\}$, $\{ scs \mid s$ is in $\{a, b\}^*\}$

Pumping Lemma for CFLs ...contd

- Examples ...contd
  - Set of legal C/Java programs
  - Much of the syntax of many high-level languages can be described by CFGs
  - But some language rules depend on context
  - E.g., a variable must be declared before use

    main( ) {int aa…a; aa…a; aa…a;}
  - Checking this is equivalent to checking whether a string has the form *xcx*

Pumping Lemma for CFLs ...contd

- In the pumping lemma, little info about the location of strings ***w***, ***y*** in string ***u*** provided
- ***Ogden's Lemma***
  - A generalization of the pumping lemma
  - Can designate "**distinguished**" positions of ***u***
  - Can guarantee pumped up portions include at least some of these distinguished positions
  - Sometimes more convenient to use; can also use when the pumping lemma fails

Conclusion of CFLs

- Recall that regular languages
  - Closed under union, Kleene-*, concatenation
  - Also closed under intersection, complement
- But, CFLs are not closed under intersection and complementation
  - There are CFLs, L1 and L2 such that L1∩L2 is not a CFL
  - There is a CFL, L such that L' is not a CFL
  - (See textbook for more details, examples)

PART 2

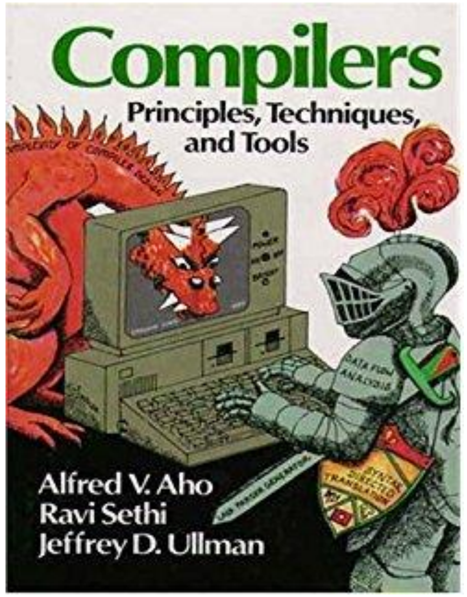
Outline:

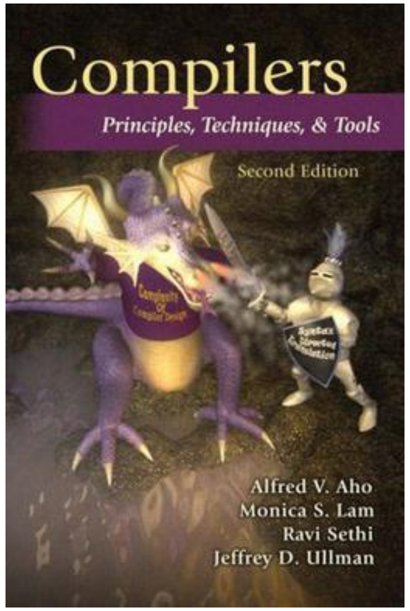
Lecture 9

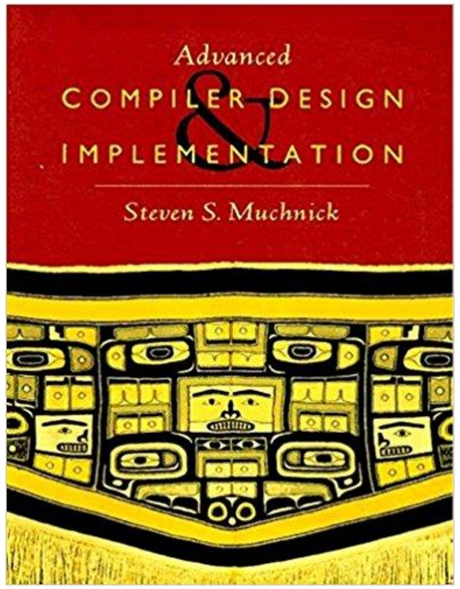
Context-free Languages (CFLs) - 4

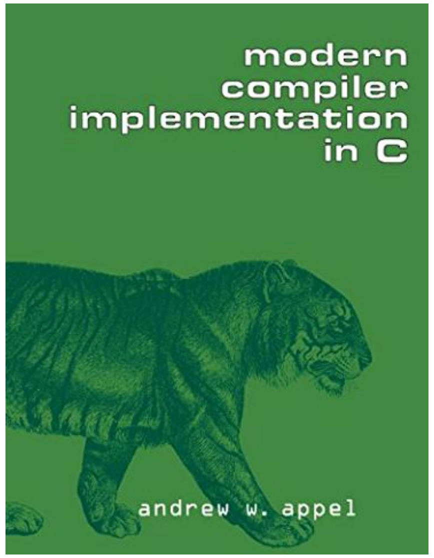
- Conclusion of CFLs/PDAs
- **Parsing in Compilers**
  - **Overview of compilation**
  - Top-down parsing
  - Bottom-up parsing

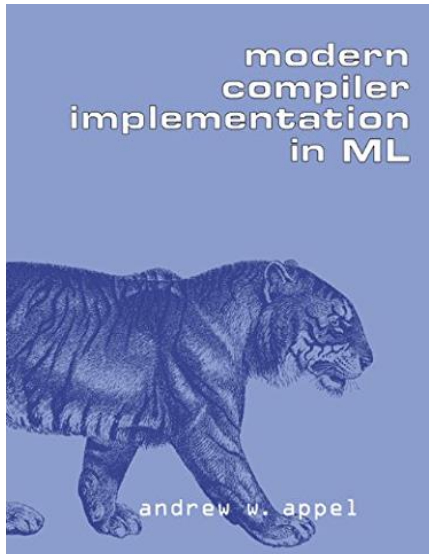
Some Books on "*Compilers*"

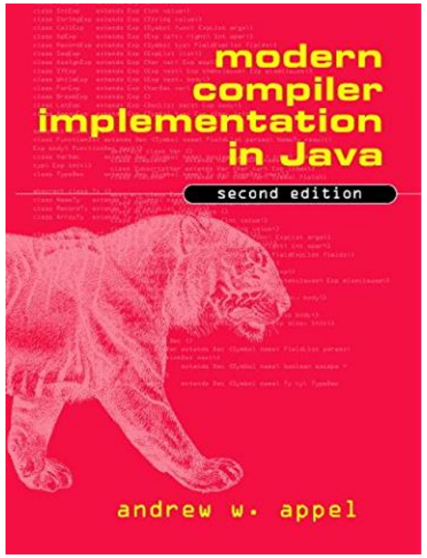


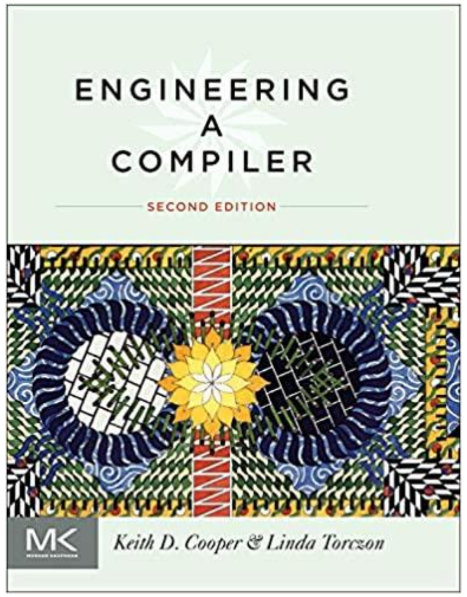


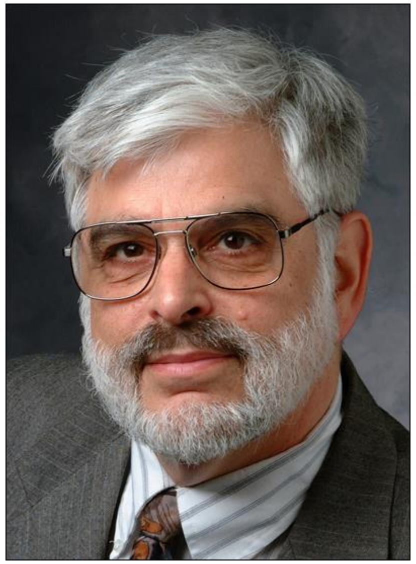












ACM Turing Award Honors Innovators Who Shaped the Foundations of Programming Language Compilers and Algorithms

Columbia's Aho and Stanford's Ullman Developed Tools and Fundamental Textbooks Used by Millions of Software Programmers around the World

ACM named Alfred Vaino Aho and Jeffrey David Ullman recipients of the 2020 ACM A.M. Turing Award for fundamental algorithms and theory underlying programming language implementation and for synthesizing these results and those of others in their highly influential books, which educated generations of computer scientists. Aho is the Lawrence Gussman Professor Emeritus of Computer Science at Columbia University. Ullman is the Stanford W. Ascherman Professor Emeritus of Computer Science at Stanford University.

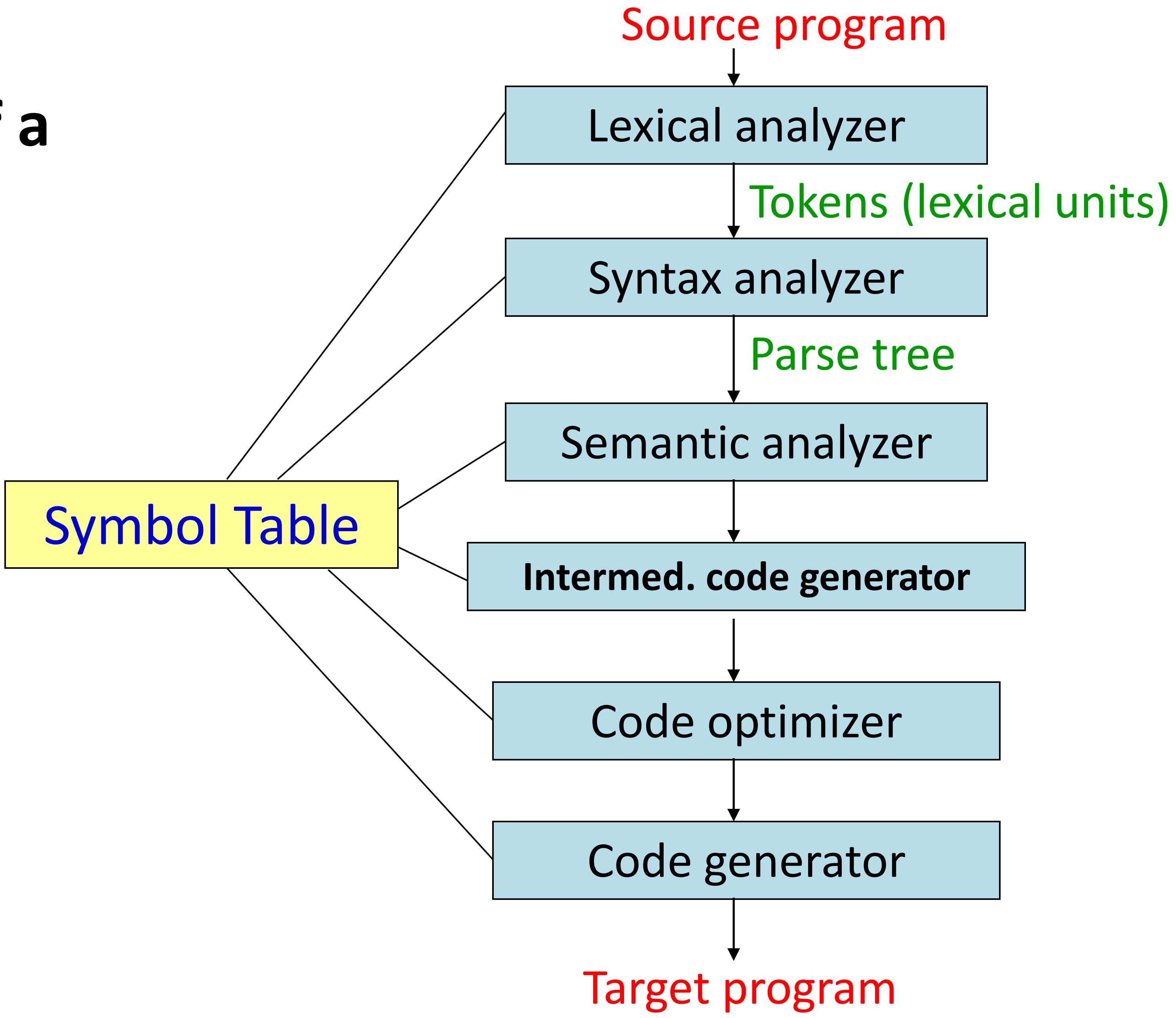
Source: https://awards.acm.org/about/2020-turing

Compilation Overview

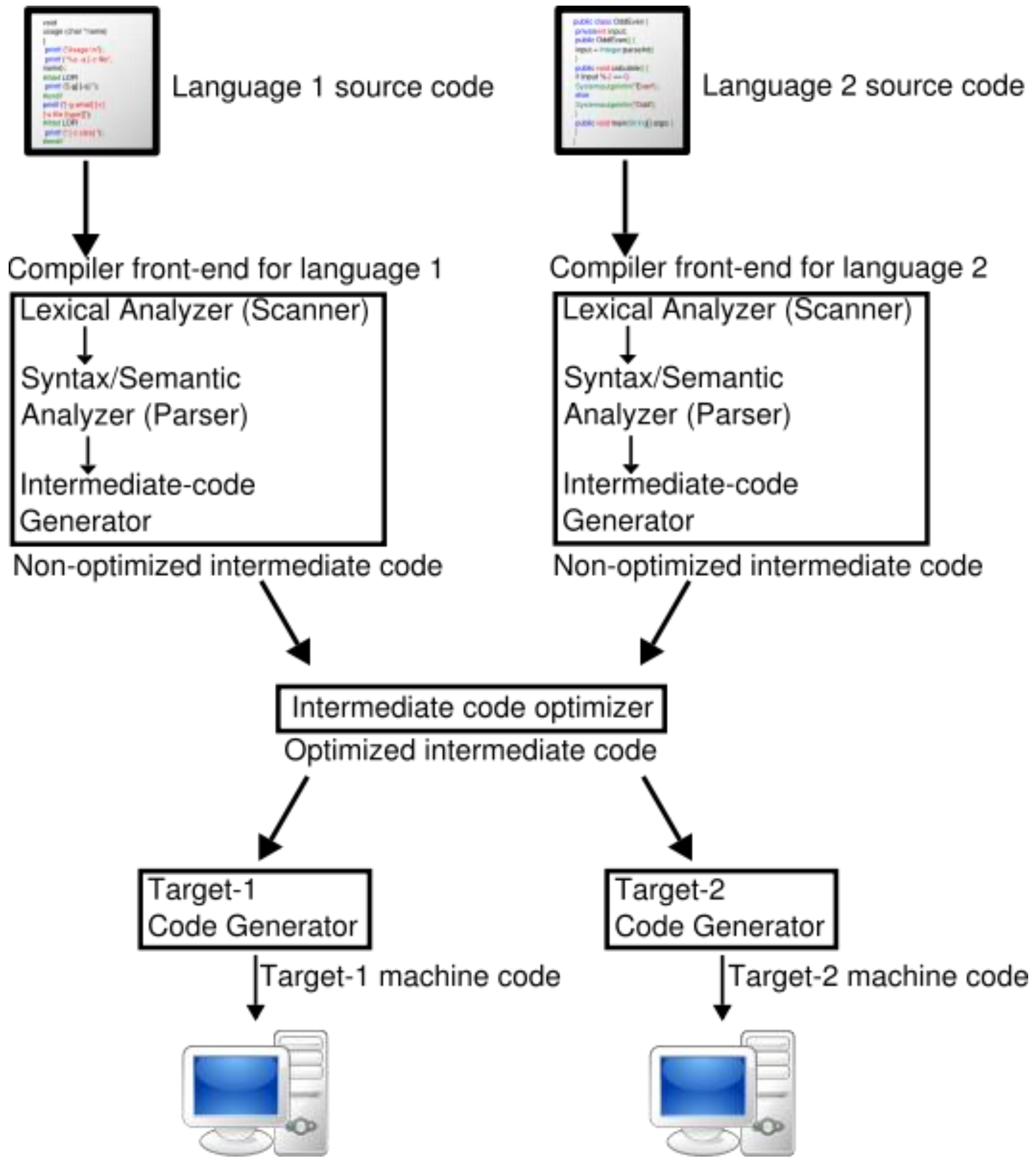
- ***Compiling*** **: language translation**
  - Translate a program (source) written in one language into a program in another language (target)

- Compilation consists of multiple ***phases***
  - May involve multiple ***passes*** through the source also

Phases of a Compiler



The operation of a typical multi-language, multi-target compiler


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Non-optimized intermediate code
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

Ref: Wikipedia at http://en.wikipedia.org/wiki/Compiler

Lexical Analysis

- Also called *scanning*, *linear analysis* or *lex*
- Reads stream of characters in the source program left-to-right
- Discards white space and comments
- If language is case-insensitive, makes all characters (except string const.’s) uniform

Lexical Analysis ...contd

- Breaks the source into individual *lexical units* or *lexemes* collectively called “tokens”
  – Sequence of characters with a collective meaning in the grammar of a language
- Tokens are classified into *types*
  – E.g., identifiers, keywords, numeric constants, string constants, operators
- Non-tokens
  – Comments, preprocessor directives and macros (in C/C++), blanks, tabs and new lines

Lexical Analysis ...contd

Token type	Example lexemes	
ID	sum, rate, calc	
NUM	60, 0 , 23, 082	
REAL	66.1, .5, 10. , 1e-9	
IF	if	(keyword)
EQ	=	
PLUS	+	
SCOLON	;	
LPAREN	(	

Lexical Analysis ...contd

- Consider:

```
pos = init + rate * 60 ;
```

- What is the stream of tokens generated?

  ID(pos) EQ ID(init) PLUS ID(rate) MULT NUM(60) SCOLON

Tokens as Regular Expressions

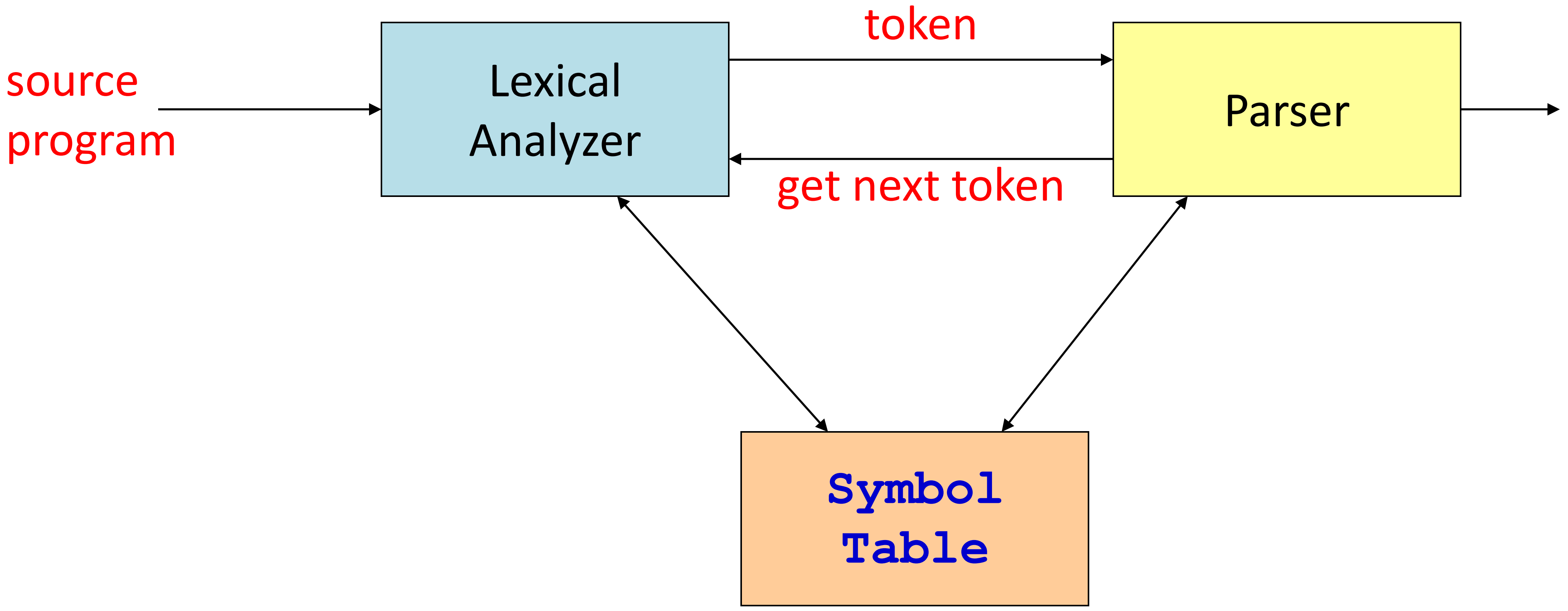
- Example definitions for tokens

IF	→ if
digit	→ [0-9]
ID	→ [a-zA-Z] ([a-zA-Z] | digit)*
opt_frac	→ "." digit^+ | Λ *or* ("."digit^+)?
opt_exp	→ (E (+ | - | Λ) digit^+) | Λ
NUM	→ digit^+ opt_frac opt_exp

Tokens as Regular Expressions

- Consider the example definitions above
  - Does "**if67**" match as the single token ID or two tokens IF and NUM?
  - Does the string "**if 367**" begins with token ID or the token for keyword IF?
- How to resolve conflicts like these?

Role of Lexical Analyzer



Lexical Analyzer Generation

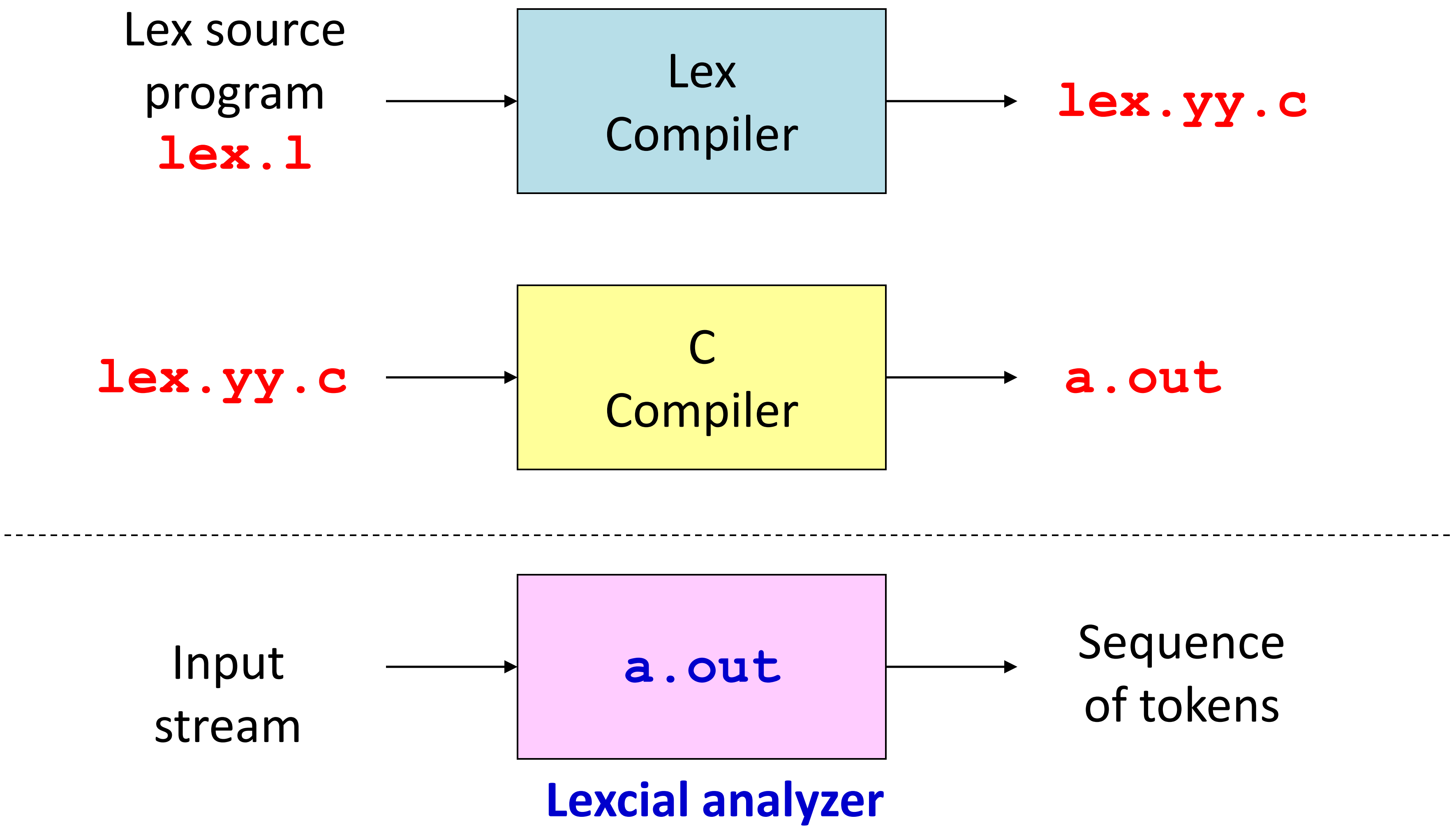
- Lexical analyzers can be generated automatically using a software tool called "*lexical analyzer generator*"
  - E.g., lex, flex, JLex, JavaCC
  - Basically: regular expressions $\rightarrow$FA
- We provide as input the lexical specifications for the tokens to be matched

Example: *lex*

- Tool is also called "*lex compiler*"
- Works with the "*yacc*" program (parser generator) in C environment

- Input is given in "Lex language"
  - Specify patterns using regular expressions
  - Specify actions for the lexical analyzer
    - E.g., make entries to symbol table, return token to parser

How *lex* is Used

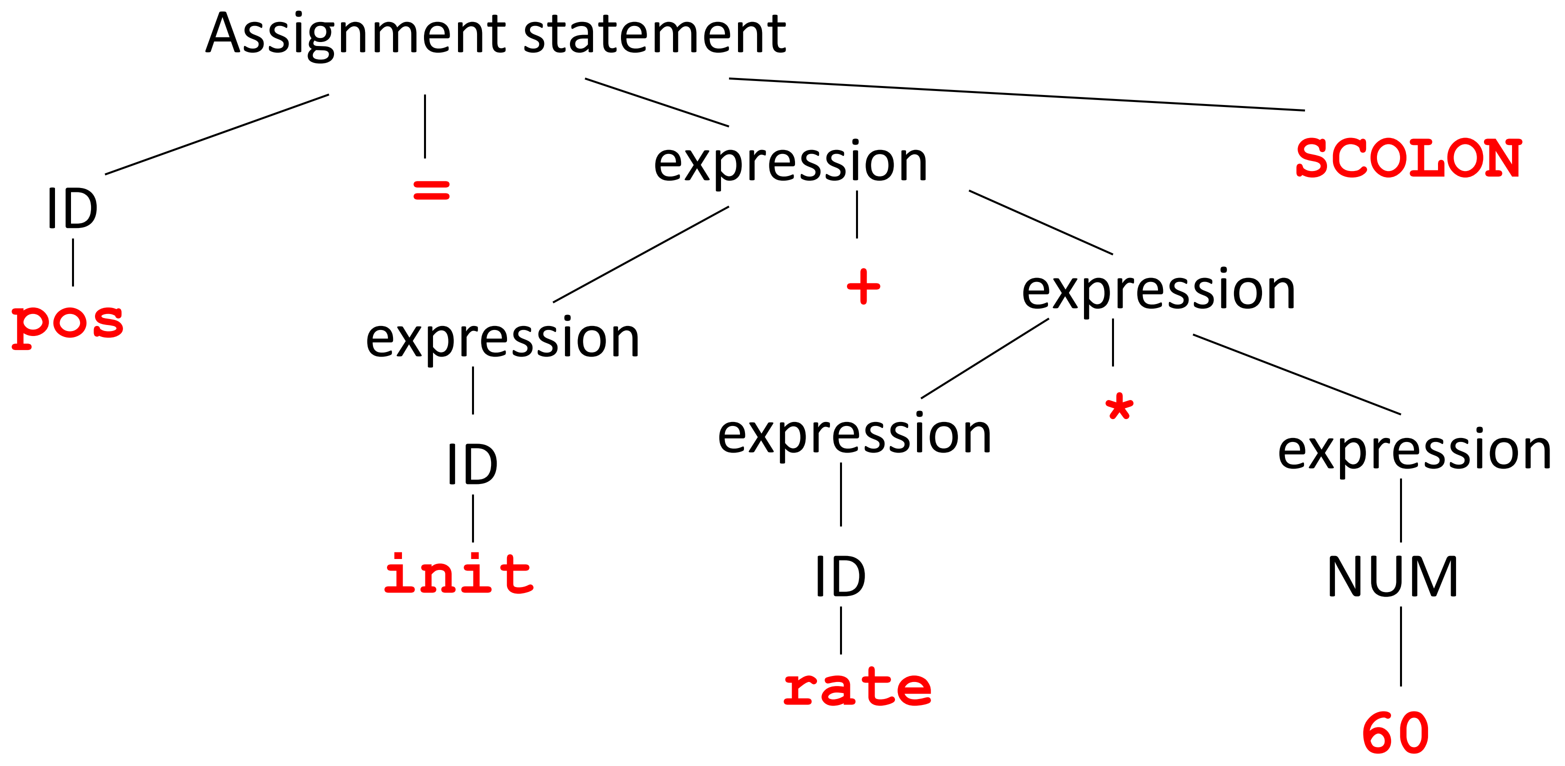


Parsing (Syntax Analysis)

- Also called hierarchical analysis
- Groups tokens hierarchically into grammatical phrases called *parse trees*
- Determines the syntactic structure of the program and of individual statements
- Detects errors in statements that violate grammar rules

Parsing ...contd

- E.g., `pos = init + rate * 60 ;`



Parsing ...contd

- Hierarchical structure of a program usually expressed by recursive rules
- Example: definition of an expression
  1. Any identifier (ID) is an expression
  2. Any number (NUM) is an expression
  3. If exp_1 and exp_2 are expressions, so are
     - $exp_1 + exp_2$
     - $exp_1 * exp_2$
     - (exp_1)

and so on…

Lexical & Syntax Constructs

- Lexical constructs (tokens) not recursive
  - Can recognize tokens via simple linear scan
  - Formalized by *regular grammars*
- Syntactic constructs often recursive
  - Formalized by *context-free grammars* (CFG)
  - Linear scan not powerful to analyze expressions/statements

Parsing ...contd

- Suppose we are given,
  - (i) a grammar G over an alphabet Σ
  - (ii) a string x in Σ*
- For example
  - (i) a set of syntax rules for a programming language
  - (ii) a string that could be a program
- Parsing the string x means: we find a derivation of x in G or determine that there is no derivation

Parsing ...contd

- Consider the top-down and bottom-up approaches we discussed to build a PDA
  - Can they be used as parsing algorithms?

- Neither cannot be used by itself because the PDA we obtain is non-deterministic
  - (Guessing involved)

Parsing ...contd

- Two approaches

1. **Top-down parsing**
   - Non-determinism eliminated using *lookahead*
   - Lookahead: use the next input symbol (in addition to the stack symbol) to decide
   - LL(*k*) grammar: looking ahead *k* symbols in the input is sufficient to choose a move
   - For LL(1) grammars, a parsing method called "recursive descent" can also be used

Parsing ...contd

2. **Bottom-up (shift-reduce) parsing**
   - Non-determinism exists in the previous bottom-up approach in two ways
     - There can be choice: whether to shift or reduce?
     - If more than one reduction possible: which one?
   - Can eliminate non-determinism by introducing precedence relations, look ahead
   - Various forms: operator-precedence parsing and more general LR parsing

PART 3

Outline:

Lecture 9

Context-free Languages (CFLs) - 4

- Conclusion of CFLs/PDAs
- **Parsing in Compilers**
  - Overview of compilation
  - **Top-down parsing**
  - Bottom-up parsing

Recursive-Descent Parsing

- A Top-down approach (leftmost derivation)
  - Executes a set of recursive procedures to process input; may involve backtracking
- A (simple form of) recursive-descent method is referred to as *predictive parsing*
  - No backtracking required
- Some grammars can be easily parsed using predictive parsing approach

Example 1

- Consider the following grammar

S →	***if*** E ***then*** S ***else*** S
S →	***begin*** S L
S →	***print*** E
L →	***end***
L →	**;** S L
E →	***num*** = ***num***

Example 1 ...contd

- A recursive-descent parser for this grammar has
  - One function for each non-terminal
  - One clause for each production

- Sample program code next 2 slides

```
enum token {IF,THEN,ELSE,BEGIN,END,PRINT,SEMI,NUM,EQ};
enum token getToken(void);

enum token tok;
void advance() { tok = getToken(); }
void eat(enum token t)
  { if (tok == t) advance(); else error();}

void S(void) {
  switch (tok) {
    case IF: eat(IF); E(); eat(THEN); S();
                eat(ELSE); S(); break;
    case BEGIN: eat(BEGIN); S(); L(); break;
    case PRINT: eat(PRINT); E(); break;
    default: error();
  }
}
```

S →	if E then S else S
S →	begin S L
S →	print E
L →	end
L →	; S L
E →	num = num

```
void L(void) {
  switch (tok) {
    case END: eat(END); break;
    case SEMI: eat(SEMI); S(); L(); break;
    default: error();
  }
}

void E(void) {
  eat(NUM); eat(EQ); eat(NUM);
}

/* assume error() and getToken() are suitably implemented */
```

$S \rightarrow$	if E then S else S
$S \rightarrow$	begin S L
$S \rightarrow$	print E
$L \rightarrow$	end
$L \rightarrow$	; S L
$E \rightarrow$	num = num

Example 2

- Now consider the following grammar

$S \rightarrow E\ \$$ (\$=end marker)

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow \textbf{\textit{id}}$

$F \rightarrow \textbf{\textit{num}}$

$F \rightarrow (E)$

Example 2 …contd

- Try the same approach as in Example 1
  - One function for each non-terminal
  - One clause for each production
  - Try S( ), E( ) and T( )
- Sample program next slide

```
void S(void) { E(); eat(EOF); }

void E(void) { switch (tok) {
      case ?: E(); eat(PLUS); T(); break;
      case ?: E(); eat(MINUS); T(); break;
      case ?: T(); break;
      default: error();
   }
}

void T(void) { switch (tok) {
      case ?: T(); eat(TIMES); F(); break;
      case ?: T(); eat(DIV); F(); break;
      case ?: F(); break;
      default: error();
   }
}
```

We have *conflicts* !

  i.e., no way to know which clause to use in E( ) and T( ) functions

Predictive Parsing: Pros/Cons

- Algorithm is simple so that we can construct parsers by hand (without using parser-generator tools)

- Works only with some grammars
  - The ***first*** terminal symbol of each sub-expression provides enough information to choose which production to use

How to Obtain Predictive Parsers

- One ***may*** be able to obtain a grammar that can be parsed by a predictive parser by

1. carefully writing the grammar
2. eliminating *left-recursion* from it
3. *left factoring* the resulting grammar

Left Recursion

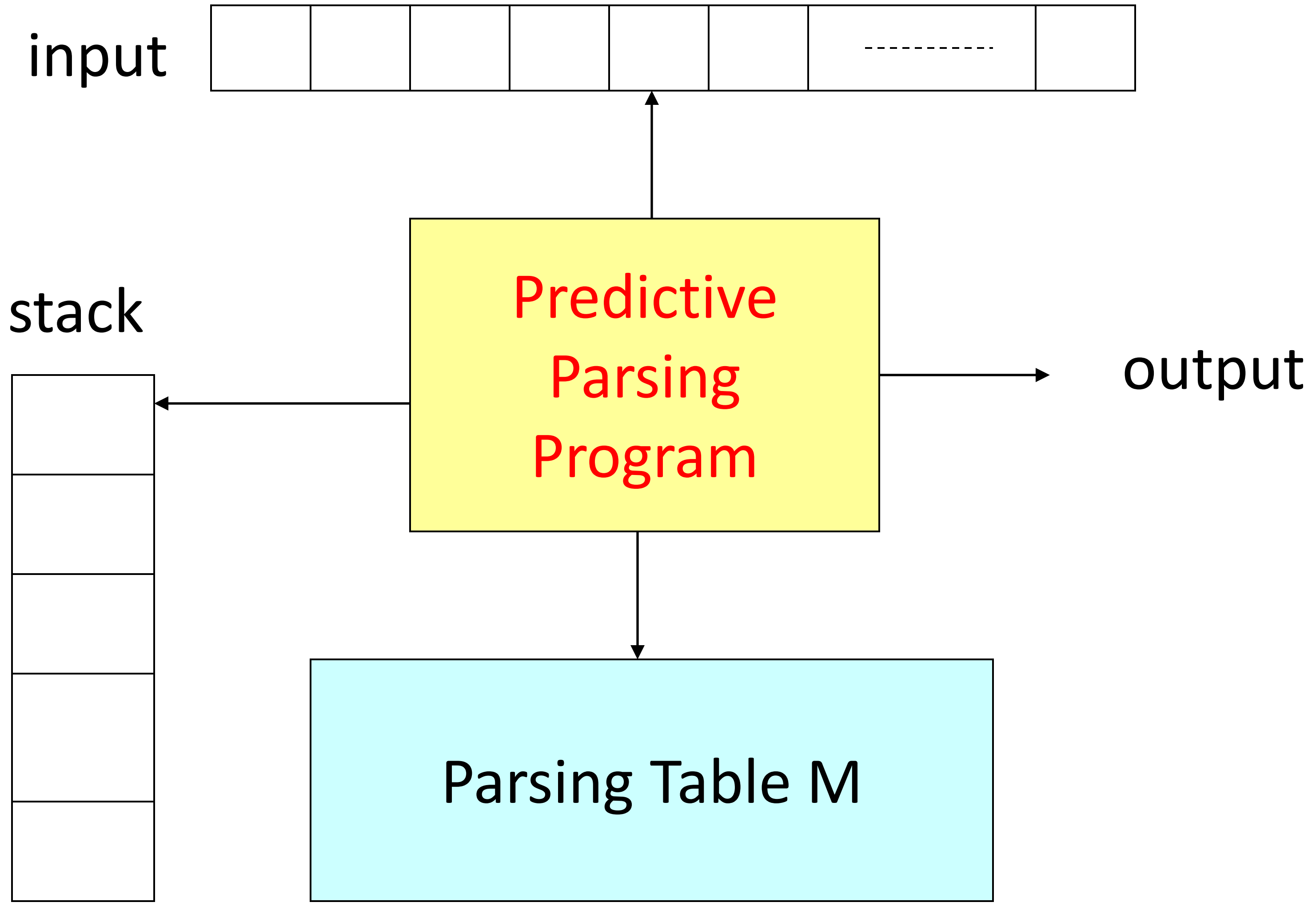
- Top-down parsing methods cannot handle left-recursive grammars [they are not LL(1)]

- Right-recursion is OK for LL(1)

Table-Driven Predictive Parsing

- Driven from a *parsing table* M[V, T]
- Lookup table for production to apply
  - V is a non-terminal
  - T is a token (terminal or $)
- A non-recursive predictive parser by maintaining a stack explicitly
  - Recursive method has implicit stack
  - (Recall: CFG recognition requires a stack)

Model of a non-recursive predictive parser



Example Parsing Table

Non-terminal	Input symbol (terminal or token)				
	id	+	*	…	$
E	$E \rightarrow T\,E'$				
E'		$E' \rightarrow +TE'$			$E' \rightarrow \Lambda$
…					
F	$F \rightarrow id$				

- Blank: error entry
- Non-blank: production to expand top of stack non-terminal

Table-Driven Predictive Parsing

- Parsing Algorithm
  - Algorithm considers X, symbol on top of stack, and *a*, current input symbol (token)
  - If X=*a*=$ → halt (success)
  - If X=*a*≠$ → pop X, advance to next input symbol
  - If X is a non-terminal → lookup M[X,*a*] in parsing table (either error or production)
    - If M[X, a]=$Y_1Y_2\ldots Y_k$, then: pop X, push $Y_1Y_2\ldots Y_k$ onto stack with Y_1 on top, output production

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1)
  - Left-to-right parse, Leftmost-derivation, 1-symbol look-ahead

- If a parsing table entry contains > 1 production, then the grammar
  - Is either ambiguous or left-recursive
  - Cannot be parsed by predictive parsing

LL(*k*) Grammars

- *k* symbol look-ahead (rarely used)
- Parsing table: columns are every sequence of *k* terminals
- Every LL(1) grammar is a LL(2) grammar
- No ambiguous grammar is LL(*k*) for any *k*

PART 4

Outline:

Lecture 9

Context-free Languages (CFLs) - 4

- Conclusion of CFLs/PDAs
- **Parsing in Compilers**
  - Overview of compilation
  - Top-down parsing
  - **Bottom-up parsing**

Bottom-up Parsing

- Shift-reduce parsing
  - A general style of bottom-up parsing
  - Attempts to construct a parse tree for an input string
    - Beginning at the leaves (the bottom)
    - Working up towards the root (the top)
  - A process of "*reducing*" the input string to the start symbol

Introduction …contd

- At each reduction step, a substring matching the RHS of a production is replaced by the symbol on the LHS

- If the substring is chosen correctly at each step, a *rightmost derivation* is traced in *reverse*

Example

- Consider the following grammar

  S → ***a***AB***e***　　A → A***bc*** | ***b***　　B → ***d***

- The sentence "***abbcde***" can be *reduced* to S by the following steps

  abbcde → ***a***A***bcde*** → ***a***A***de*** → ***a***AB***e*** → S

  ←――――――――――――
  rightmost derivation (in reverse)

Introduction …contd

- Two problems to be solved
  - How to locate the substring to be reduced?
  - What production to choose if there are many productions with that substring on the RHS?

Stack Implementation (of Shift-Reduce Parsing)

- A stack is used to hold grammar symbols and an input buffer to hold the input string
  - ($ marks bottom of stack and end of input)

- Initially stack is empty and we have a string, say ***w***, as input ($ is end of string)

STACK	INPUT
$	***w***$

Stack Implementation ...contd

- The parser *shifts* zero or more input symbols onto the stack until a substring β (called a “handle”) is on top of the stack
  - Then β is *reduced* to the LHS of a production
- Repeat until (an error is seen or) the stack has the start symbol and input is empty

STACK	INPUT
S$	$

At this point, parser halts with success

Example

- Suppose we have the (ambiguous) CFG:

  $\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E} \mid \mathrm{E} * \mathrm{E} \mid (\mathrm{E}) \mid \boldsymbol{id}$

- Consider the input string $\boldsymbol{id}_1 + \boldsymbol{id}_2 * \boldsymbol{id}_3$ that can be derived (rightmost) as:

  $\mathrm{E} \Rightarrow \mathrm{E} + \mathrm{E} \Rightarrow \mathrm{E} + \mathrm{E} * \mathrm{E} \Rightarrow \mathrm{E} + \mathrm{E} * \boldsymbol{id}_3 \Rightarrow \mathrm{E} + \boldsymbol{id}_2 * \boldsymbol{id}_3 \Rightarrow \boldsymbol{id}_1 + \boldsymbol{id}_2 * \boldsymbol{id}_3$

- Show the actions of a shift-reduce parser

Stack	Input	Action
\$	$id_1 + id_2 * id_3\$$	shift
id_1 \$	$+ id_2 * id_3\$$	reduce by E→*id*
E \$	$+ id_2 * id_3\$$	shift
+ E \$	$id_2 * id_3\$$	shift
id_2 + E \$	$* id_3\$$	reduce by E→*id*
E + E \$	$* id_3\$$	shift
* E + E\$	$id_3\$$	shift
id_3 * E + E \$	\$	reduce by E→*id*
E * E + E \$	\$	reduce by E→E*E
E + E \$	\$	reduce by E→E+E
E \$	\$	accept

Actions

- *Shift*: shift next symbol onto top of stack
- *Reduce*: locate the end of a handle within the stack and decide the non-terminal to replace handle
- *Accept*: announce successful completion
- *Error*: discover syntax error and call error recovery routine

Conflicts during Parsing

- Shift-reduce conflict
  - Cannot decide whether to shift or reduce
- Reduce-reduce conflict
  - Cannot decide which of the several reductions to make (multiple productions to choose from)

Conflict Resolution

- Conflict resolution by adapting the parsing algorithm (e.g., in parser generators)

- Shift-reduce conflict
  - *Resolve in favor of shift*

- Reduce-reduce conflict
  - *Use the production that appears earlier*

LR Parsing: Introduction

- An efficient bottom-up parsing technique
  - Can parse a large set of CFGs
- Technique is called LR(*k*) parsing
  - “L” for left-to-right scanning of input
  - “R” for rightmost derivation (in reverse)
  - “*k*” for number of tokens of look-ahead
    - (when *k* is omitted, it implies *k*=1)

LL(*k*) vs. LR(*k*)

- LL(*k*): must predict which production to use having seen only first *k* tokens of RHS
  - Works only with some grammars
  - Simple algorithm (can construct by hand)

- LR(*k*): more powerful
  - Can postpone decision until seen tokens of entire RHS of a production and *k* more beyond

More on LR(*k*)

- Can recognize virtually all programming language constructs (if CFG can be given)
- Most general non-backtracking shift-reduce method known, but can be implemented efficiently
- Class of grammars can be parsed is a superset of grammars parsed by LL(*k*)
- Can detect syntax errors as soon as possible

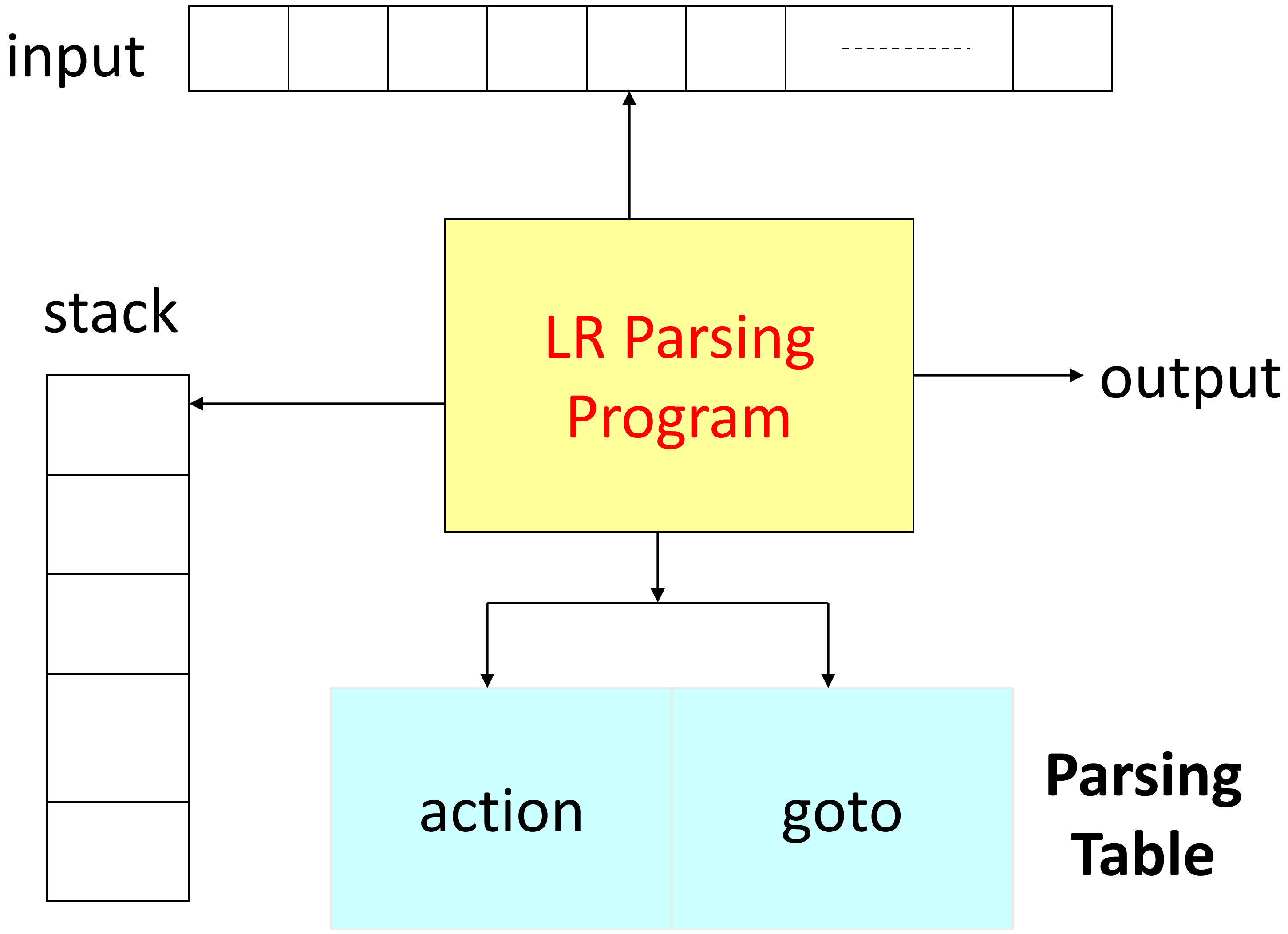
More on LR(*k*) ...contd

- Main drawback: too tedious to do by hand for typical programming language grammars
  - We need a LR parser generator
  - Many available
    - *Yacc* (*yet another compiler compiler*) or *bison* for C/C++ environment
    - *CUP* (*Construction of Useful Parsers*) for Java environment
    - *PLY* (*Python Lex-Yacc*) for Python
  - We write the grammar and the generator produces the parser for that grammar

LR Parsing

- [Ref: Tiger book]
- Parser has stack and input
  - Model: next slide
  - Driver program same for all LR parsers
- First *k* tokens of the input are *lookahead*
- Based on stack contents and lookahead
  - Either *shift*: move 1st input token onto stack
  - Or, *reduce*: use a grammar rule $X \rightarrow ABC$

Model of an LR Parser



LR Parsing Engine

- Deciding when to shift and when to reduce is based on a DFA applied to the stack
  - Edges of DFA labeled by symbols that can be on stack (terminals + non-terminals)
- Transition table defines transitions (and characterizes the type of LR parser)
  - Table indexed by [*state*, *terminal (token) or non-terminal*]

Entries in Transition Table

Entry	Meaning
s*n*	Shift into state *n* (advance input pointer to next token)
g*n*	Goto state *n*
r*k*	Reduce by rule (production) *k*; corresponding g*n* gives next state
a	Accept
	Error (denoted by blank entry)

LR(*k*) Parsing Tables

- An LR(*k*) parser uses contents of stack and next *k* tokens of input to decide
- For k=2, table has columns for every 2-token combination
  - k > 1 not used in practice for compilation
- k=0 $\rightarrow$ LR(0) grammar $\rightarrow$ Can be parsed by looking only at stack (no lookahead)

LALR(1) Parsing Table

- LR(1) parsing tables can be very large
- *LALR* means *L*ook-*A*head-*LR*
  - Tables can be significantly smaller
  - Yet most language constructs can be parsed
  - Smaller table by merging states
- Most programming languages have a LALR(1) grammar
  - Standard for programming languages and automatic parser generators

Conclusion

- Summary of this lecture
  - Pumping lemma for CFLs
  - Conclusion of CFL/PDA
  - Parsing in compilers
    - Overview of Compilation
    - Top-down parsing
    - Bottom-up parsing

- Next topic: Turing machines!