

Compiler Design - Introduction

Sitara K.

sitara@nitt.edu

Objectives

- To introduce the major concept areas in compiler design and know the various phases of the compiler
- To understand the various parsing algorithms and comparison of the same
- To provide practical programming skills necessary for designing a compiler
- To gain knowledge about the various code generation principles
- To understand the necessity for code optimization

Syllabus

UNIT I Introduction to Compilation

Compilers - Analysis of the source program - Phases of a compiler - Cousins of the Compiler - Grouping of Phases - Compiler construction tools - Lexical Analysis - Role of Lexical Analyzer - Input Buffering - Specification of Tokens.

Lab Component: Tutorial on LEX / FLEX tool, Tokenization exercises using LEX.

UNIT II Syntax Analysis

Role of the parser - Writing Grammars - Context-Free Grammars - Top Down parsing - Recursive Descent Parsing - Predictive Parsing - Bottom-up parsing - Shift Reduce Parsing - Operator Precedent Parsing - LR Parsers - SLR Parser - Canonical LR Parser - LALR Parser.

Lab Component: Tutorial on YACC tool, Parsing exercises using YACC tool.

UNIT III Intermediate Code Generation

Intermediate languages - Declarations - Assignment Statements - Boolean Expressions - Case Statements - Back patching - Procedure calls.

Lab Component: A sample language like C-lite is to be chosen. Intermediate code generation exercises for assignment statements, loops, conditional statements using LEX/YACC.

UNIT IV Code Optimization and Run Time Environments

Introduction - Principal Sources of Optimization - Optimization of basic Blocks - DAG representation of Basic Blocks - Introduction to Global Data Flow Analysis - Runtime Environments - Source Language issues - Storage Organization - Storage Allocation strategies - Access to non-local names - Parameter Passing - Error detection and recovery.

Lab Component: Local optimization to be implemented using LEX/YACC for the sample language.

UNIT V Code Generation

Issues in the design of code generator - The target machine - Runtime Storage management - Basic Blocks and Flow Graphs - Next-use Information - A simple Code generator - DAG based code generation - Peephole Optimization.

Lab Component: DAG construction, Simple Code Generator implementation, DAG based code generation using LEX/YACC for the sample language.

Course Outcomes

- Apply the knowledge of LEX & YACC tool to develop a scanner and parser
- Design and develop software system for backend of the compiler
- Suggest the necessity for appropriate code optimization techniques
- Conclude the appropriate code generator algorithm for a given source language
- Design a compiler for any programming language

Text Books

- Alfred V. Aho, Jeffrey D Ullman, “Compilers: Principles, Techniques and Tools”, Pearson Education Asia, 2012.
- Jean Paul Tremblay, Paul G Serenson, “The Theory and Practice of Compiler Writing”, BS Publications, 2005.
- Dhamdhere, D. M., “Compiler Construction Principles and Practice”, Second Edition, Macmillan India Ltd., New Delhi, 2008.

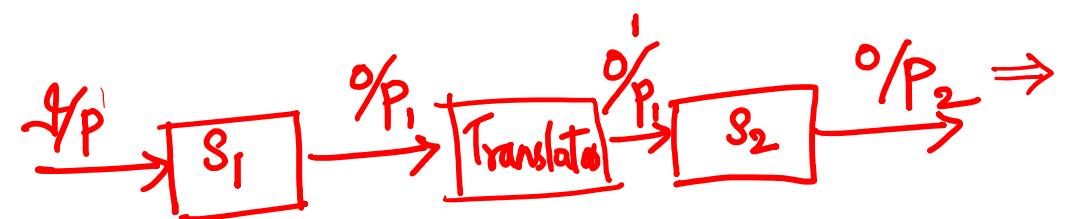
History

- Software – essential component of the current scenario.
- Early software was written in assembly languages
- Drawbacks
 - Very difficult to remember instructions
- The benefits of reusing software on different CPUs became greater than the cost of designing compiler
- Very cumbersome to write
- Need for a software that will understand human language

Language Processors.



- A **translator** inputs and then converts a **source program** into an **object or target program**.
- **Source program** is written in one language
- **Object program** belongs to an object language
- A translators could be: **Assembler**, **Compiler**, **Interpreter**



Options

- Design an interpreter / translator to convert human language to machine language
 - Difficult – Parsing, interpreting, ambiguous

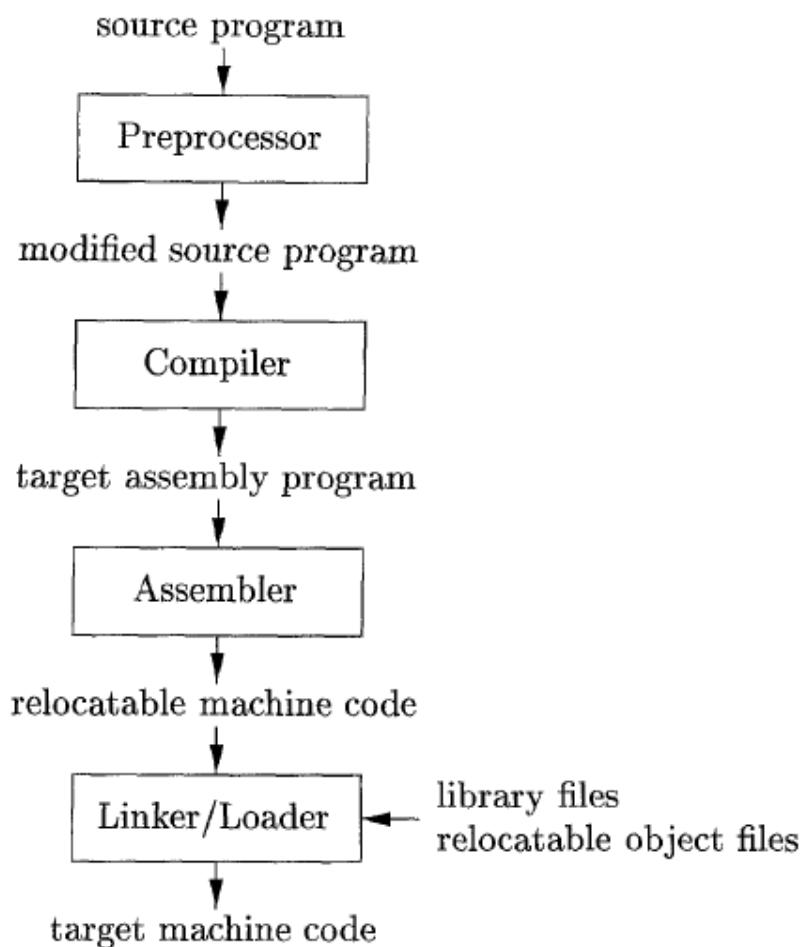
Options

- Design a compiler that will understand high level (not necessarily English) language to assembly language
 - Relatively simpler, but need a mapping of the high level language to assembly language

Options

- Design an assembler that converts assembly language to machine language
 - Target language need to be specified. Output of the various compilers to be known prior time

Language Processing System



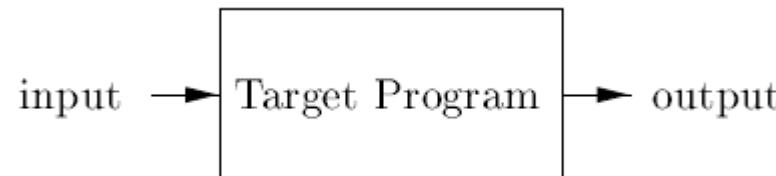
#define MAX 100

Compiler

- The first real compiler
 - FORTRAN compilers of the late 1950s
 - 18 person-years to build

What are Compilers?

- A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages.
- No concern about machine-dependent details for programmer



Compiler

- Processes source program
- Prompts errors in source program
- Recovers / Corrects the errors
- Produce assembly language program
- Compiler + assembler – Converts this to relocatable machine code

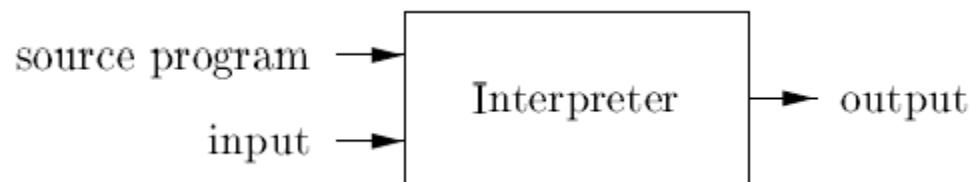
```
f1( )  
{   _____> p;  
    p = f2();  
}  
f2( )  
{  
    f3( );  
    a = a;
```

Compiler - Overview

- Translates a source program written in a High-Level Language (HLL) such as Pascal, C++ into computer's machine language (Low-Level Language (LLL))
- The time of conversion from source program into object program is called **compile time**
- The object program is executed at **run time**

Interpreter

- Language processor that executes the operation as specified in the source program
- Inputs are supplied by the user
- Processes an internal form of the source program and data at the same time (at run time); no object program is generated.

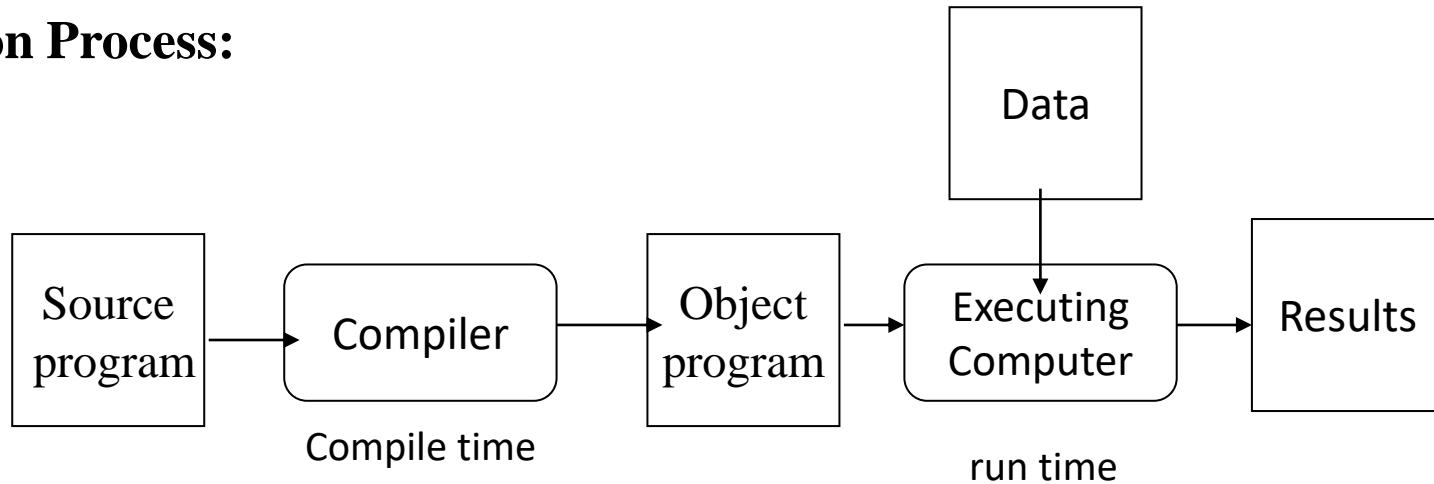


Compiler vs Interpreter

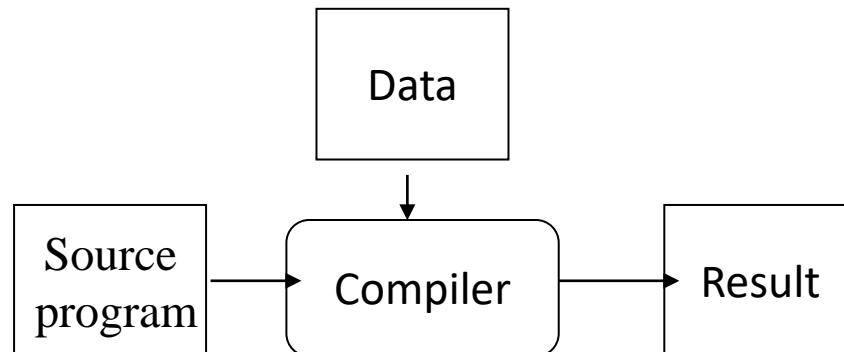
- The machine-language target program produced by a compiler is much faster than an interpreter at mapping inputs to outputs
- An interpreter, is better with error diagnostics as it executes the source program statement by statement

Overview of Compilers

Compilation Process:



Interpretive Process:



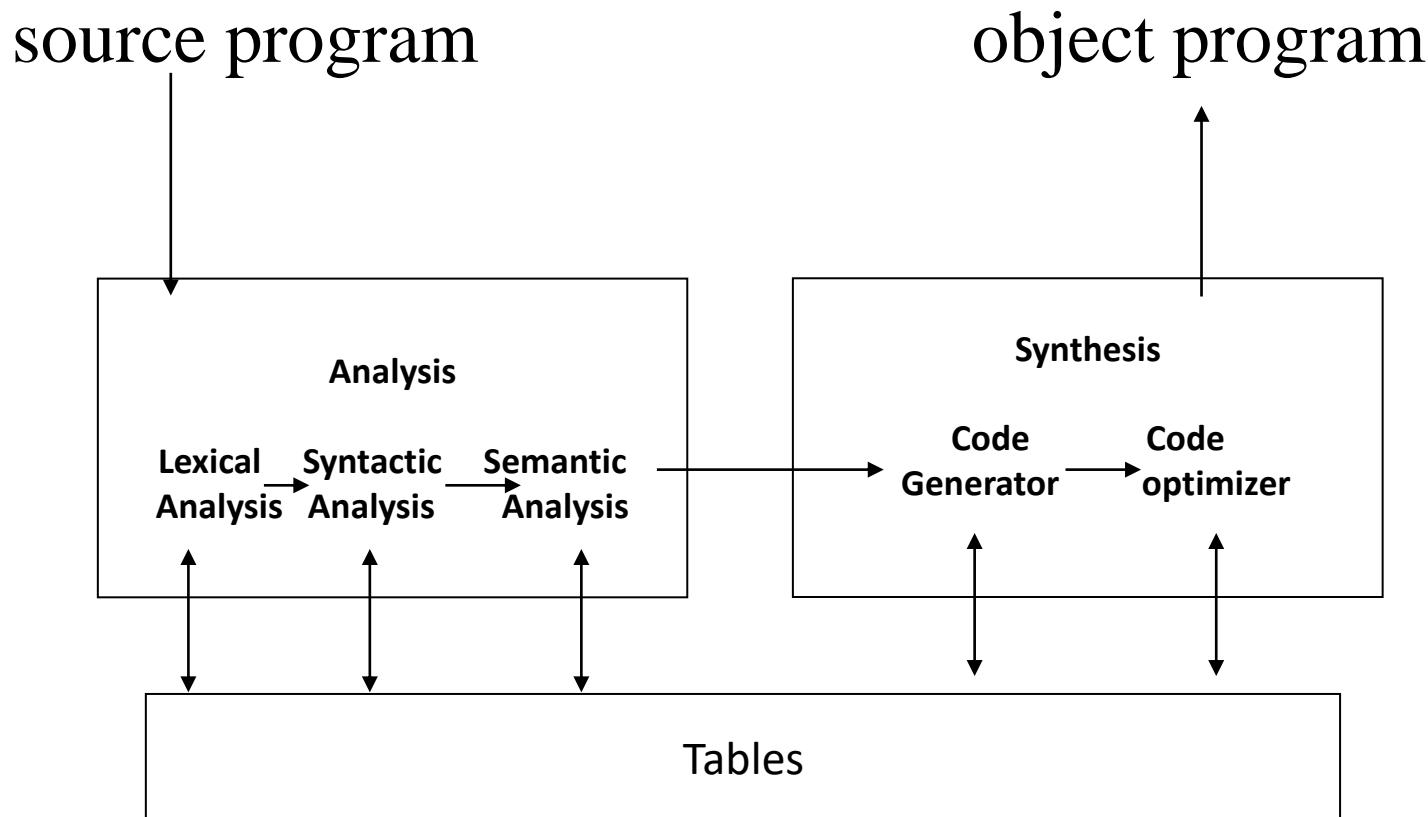
Compiler

- **Analysis** of source program: The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.
- **Synthesis** of its corresponding program: constructs the desired target program from the intermediate representation and the information in the symbol table.
- The analysis part is often called the **front end** of the compiler; the synthesis part is the **back end**.

Compiler

- Front End – Language Dependent – Depends on the source language and Target Independent
- Back End – Target Dependent – Depends on the target language but Source independent

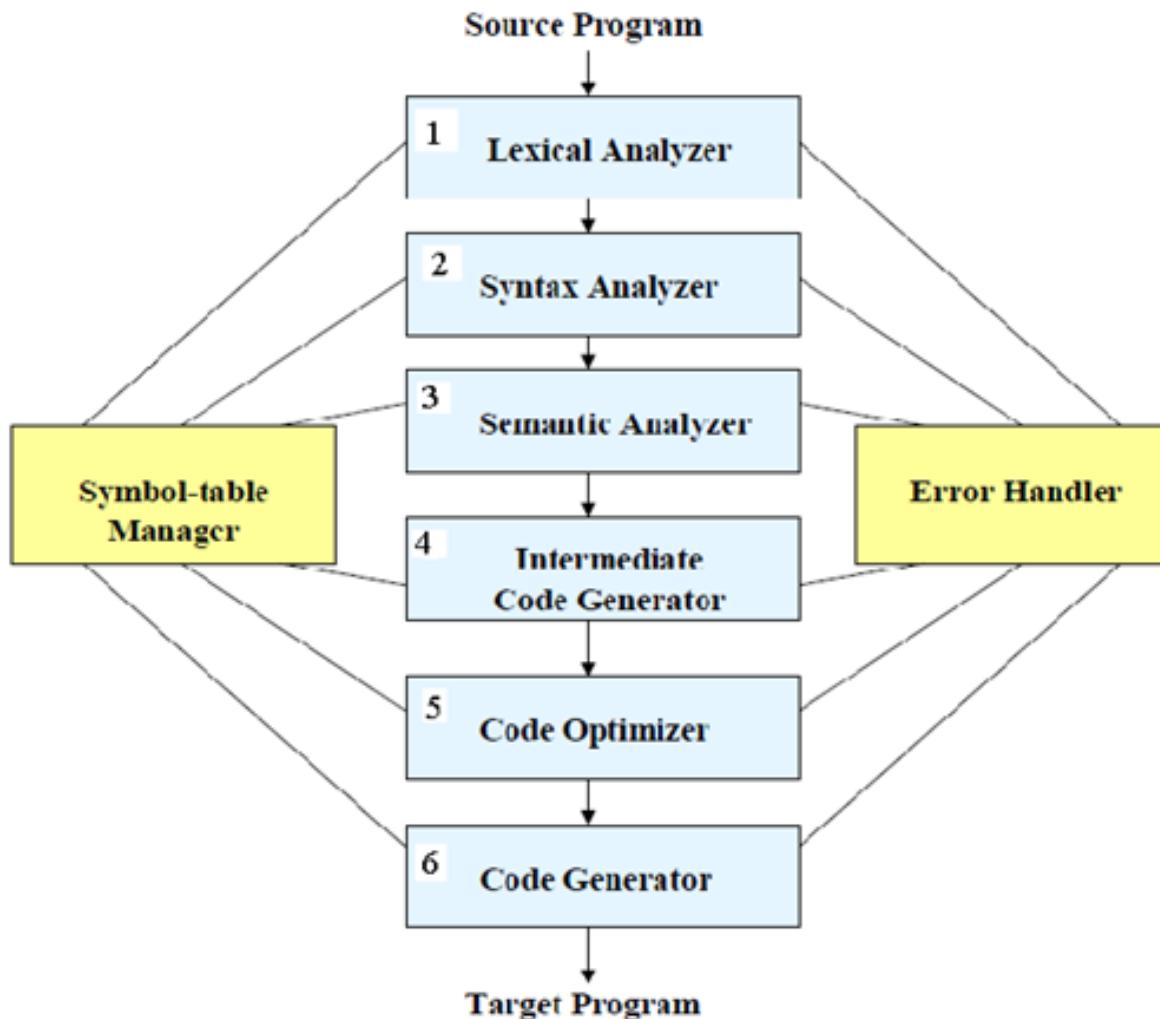
Flow of Compiler



Compiler Passes

- How many passes should the compiler go through?
- One for analysis and one for synthesis?
- One for each division of the analysis and synthesis?
- The work done by a compiler is grouped into phases

Phases of the compiler



Lexical Analysis (scanner): The first phase of a compiler

- Lexical analyzer reads the stream of characters from the source program and combines the characters into meaningful sequences called ***lexeme***
- For every lexeme, the lexer produces a token of the form which is passed to the next phase of the compiler

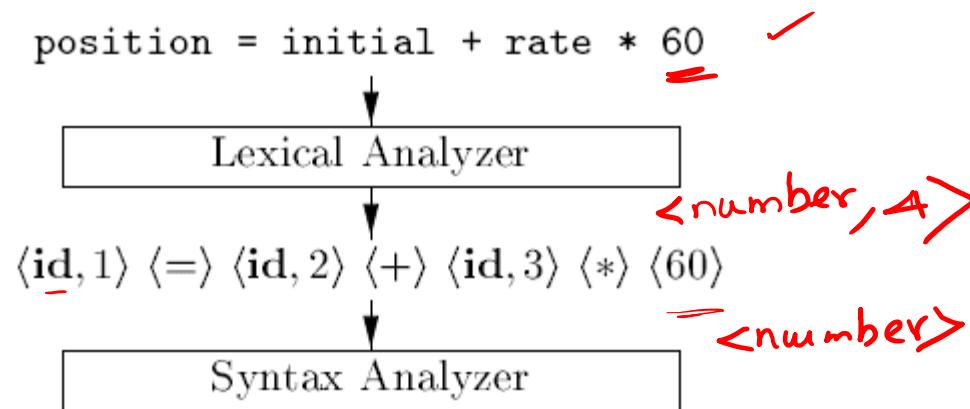
(token-name, attribute-value) ✓

C identifiers.
✓ area }
✓ area-1 }
✗ /area

Lexical Analysis (scanner): The first phase of a compiler

- Token-name: an abstract symbol is used during syntax analysis, an attribute-value: points to an entry in the symbol table for this token.
- Blanks will be discarded by the lexical analyser

$\langle, \langle=, \rangle, \rangle=$
 loop
 $\langle \text{loop} \rangle$
 $\langle \text{loop}, 5 \rangle$
 $\langle \text{loop}, \text{GET} \rangle$



SYMBOL TABLE	
1	position
2	initial
3	rate
4	number 60

$\frac{-a=b+c}{\rightarrow a = b + c}$

Example: position =initial + rate * 60

1. "position" is a lexeme mapped into a token (id, 1), where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. For notational convenience, the lexeme itself is used as the name of the abstract symbol.
3. "initial" is a lexeme that is mapped into the token (id, 2), where 2 points to the symbol-table entry for initial

4. + is a lexeme that is mapped into the token (+).
5. “rate” is a lexeme mapped into the token (id, 3), where 3 points to the symbol-table entry for rate.
6. * is a lexeme that is mapped into the token (*) .
7. 60 is a lexeme that is mapped into the token (60)

Lexical Analysis

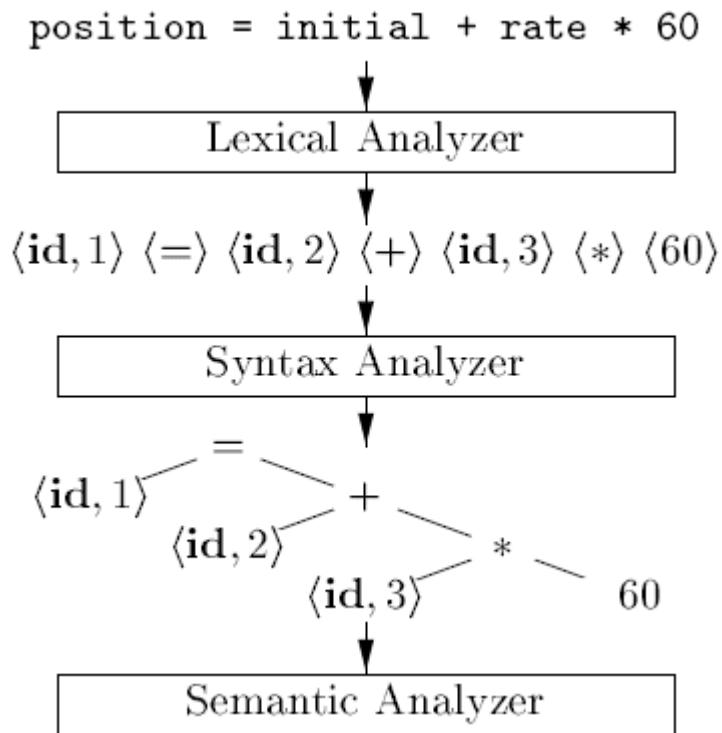
- Interface of the compiler to the outside world
- Scans input program, identifies valid words of the language in it
- Removes extra white spaces, comments etc
- Expand user defined macros
- Reports presence of foreign words
- May perform case conversions
- Generates tokens
- Generally implemented as finite automata

Syntax Analysis (parser) : The second phase of the compiler

- The parser uses the tokens produced by the lexer to create a tree-like intermediate representation that verifies the grammatical structure of the sequence of tokens
- Works hand-in-hand with lexical analyzer
- Identifies sequence of grammar rules to derive the input program from the start symbol
- A parse tree is constructed
- Error messages are flashed for syntactically incorrect programs

Syntax Analysis (parser) : The second phase of the compiler

- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation



Syntax phase

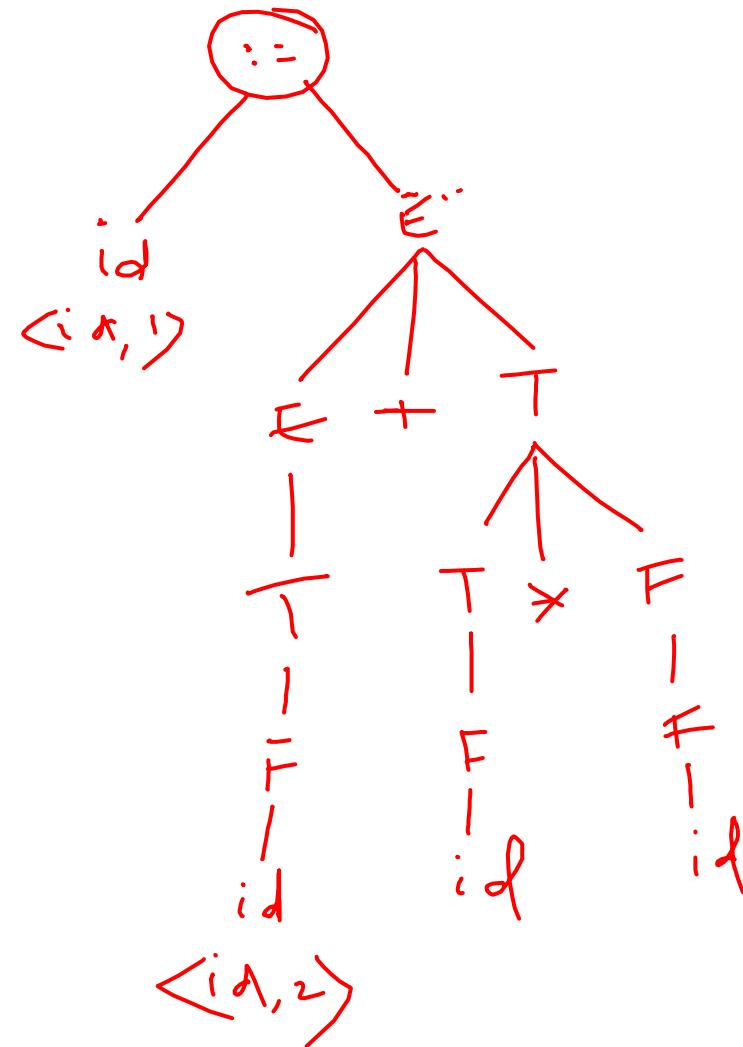
- Grammar for assignment expression

$$S \rightarrow id = E \quad -$$

$$\underline{E \rightarrow E + T \mid T} \quad -$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$



Semantic Analysis: Third phase of the compiler

- The semantic analyzer uses the output of the parser – syntax tree and the information in the symbol table to check for semantic consistency in the source program
- Gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

Semantic Analysis: Third phase of the compiler

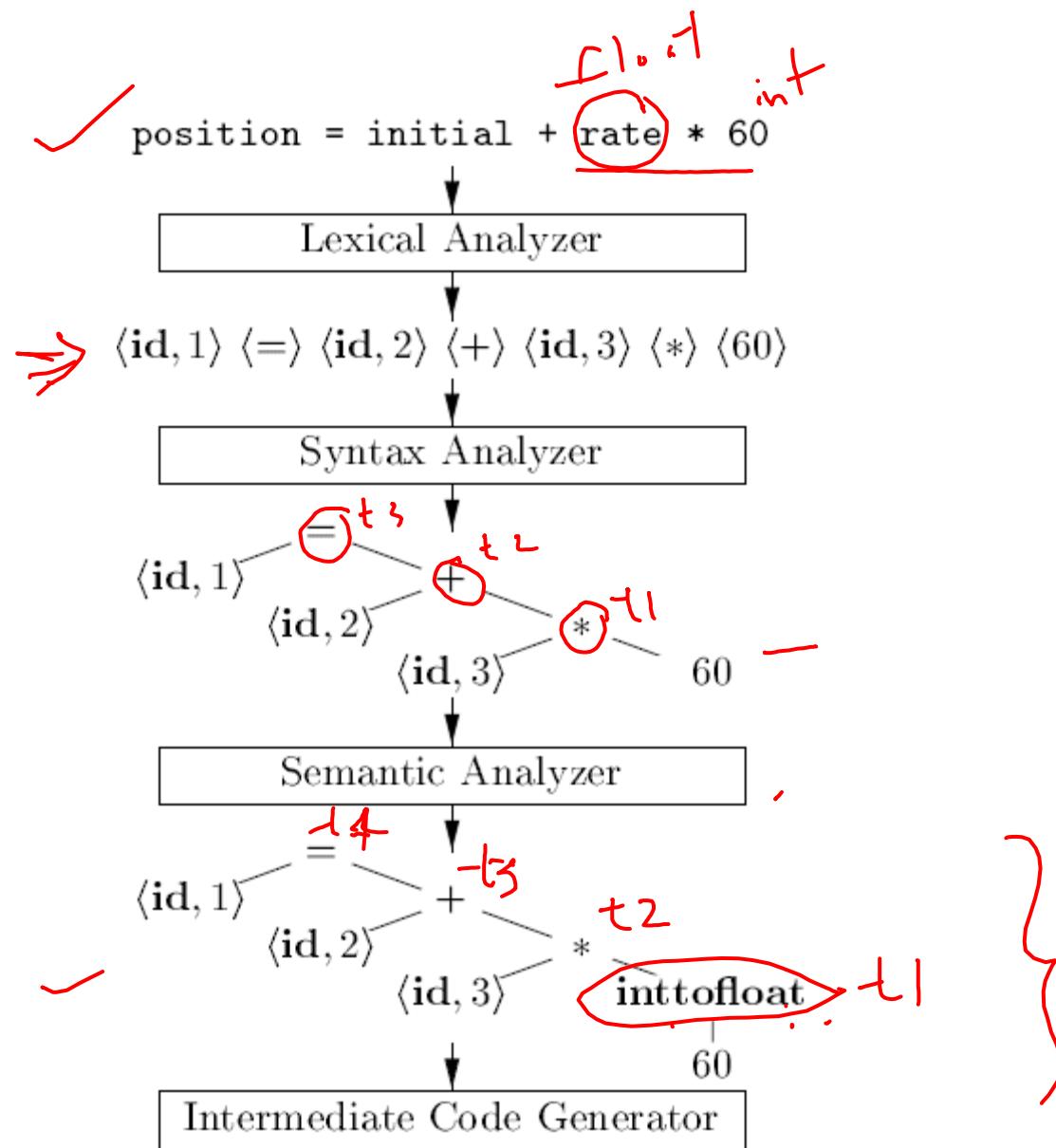
- Type checking, where the compiler checks that each operator has matching operands.
 - Array index need to be an integer; the compiler must identify an error if a floating-point number is used to as an array index
- Scope rules of the language are applied to determine types – static scope or dynamic scope

for (
 float
{ int a[10]; i=0;
 a[i] = 5; }

Semantic Analysis: Third phase of the compiler

- Coercions – a way of type conversion
- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

f1. + : + float -
a = b + c;
- z



Intermediate Code Generation: Fourth phase of the compiler

- Optional towards target code generation
- Compilers generate an explicit low-level or machine-like intermediate representation (a program for an abstract machine). This intermediate representation:
 - should be easy to produce
 - should be easy to translate into the target machine
 - Powerful enough to express the programming language constructs
- Helps to retarget the code from one processor to another

Intermediate code Generation : Three address code

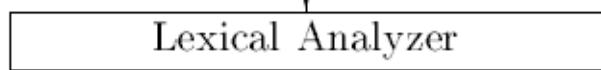
- A convention for Intermediate code generation is three address code
- Three operands at the most and 2 operators
- Example:
 - $x = y \text{ op } z$
 - $x = \text{op } y$

$$x = z - y$$

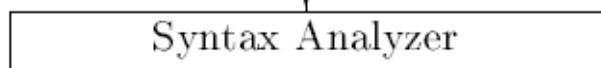
$$\cancel{x = z + y}$$

y = y + 1

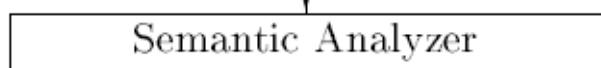
```
position = initial + rate * 60 ✓
```



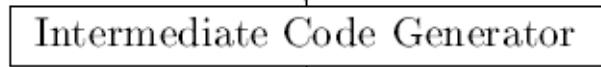
```
⟨id,1⟩ ⟨=⟩ ⟨id,2⟩ ⟨+⟩ ⟨id,3⟩ ⟨*⟩ ⟨60⟩
```



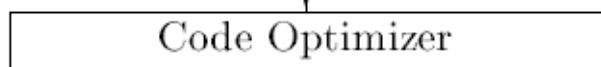
```
⟨id,1⟩ =  
        |  
        <id,2> +  
        |  
        <id,3> *  
        |  
        60
```



```
⟨id,1⟩ =  
        |  
        <id,2> +  
        |  
        <id,3> *  
        |  
        inttofloat  
        |  
        60
```

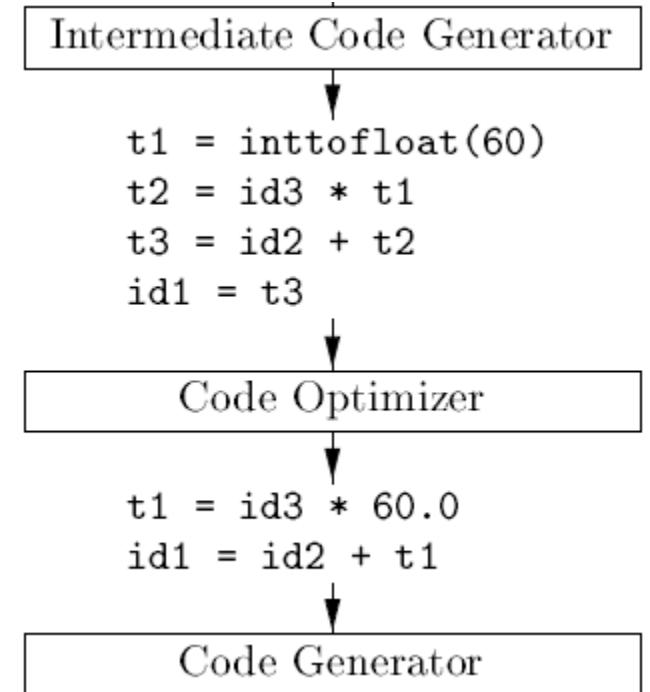


```
t1 = inttofloat(60) ✓ .  
t2 = id3 * t1  
t3 = id2 + t2  
id1 = t3
```



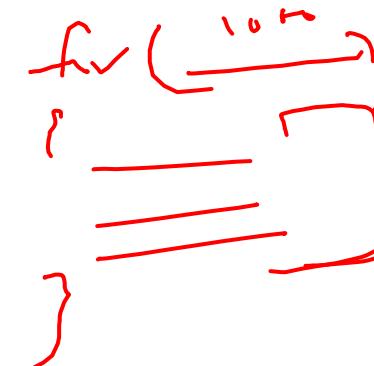
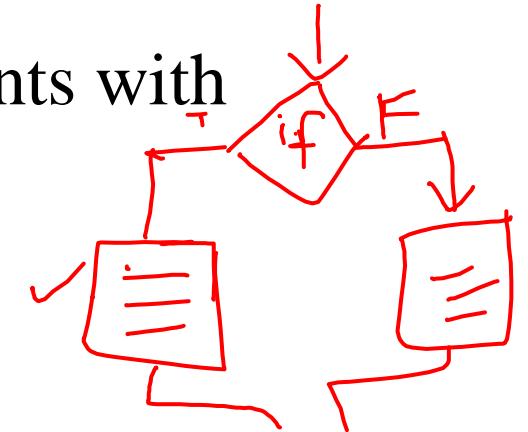
Code Optimization: - Fifth phase of the compiler

- attempts to improve the intermediate code for better target code
 - faster, shorter code, or target code that consumes less power.
 - simple optimizations that significantly improve the running time of the target program without slowing down compilation



Code Optimization: - Fifth phase of the compiler

- Automated steps of compiler generate lots of redundant code that can possibly eliminated
- Code is divided into *basic blocks* – a sequence of statements with single entry and exit
- *Local optimizations* restrict within a single basic block
- *Global optimizations* spans across basic blocks
- Optimize loops, algebraic simplifications, elimination of load-and-store are common optimizations

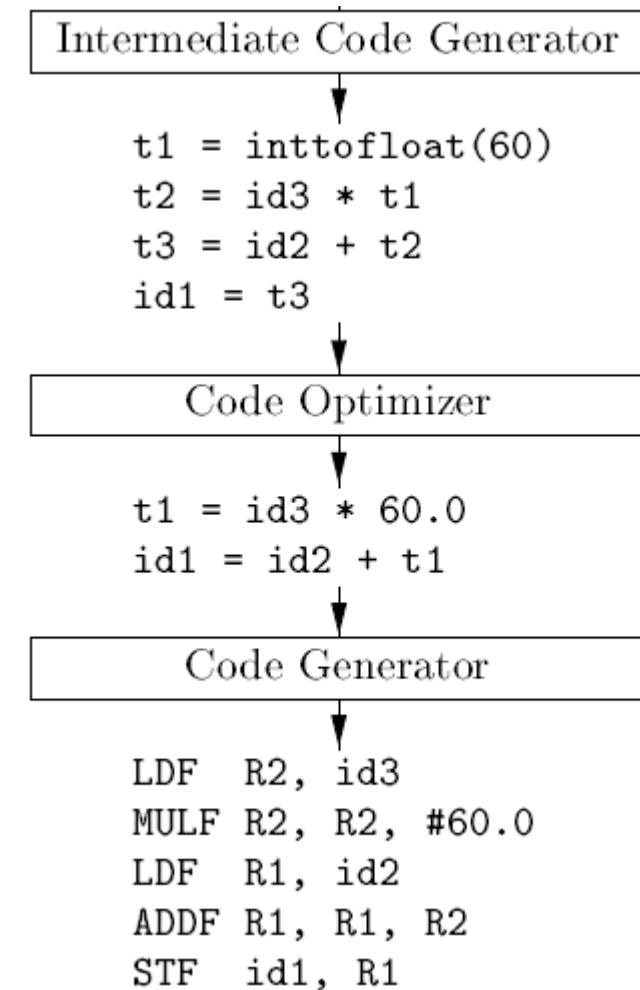


$$\begin{aligned} & \text{a} = b + d / 10; \\ & c = a + d / 10; \\ & \vdots \\ & \text{d} = \text{d} / 10; \end{aligned}$$

A hand-drawn sequence of assignments. The first assignment is $a = b + d / 10;$. The second is $c = a + d / 10;$. Below these is a vertical ellipsis. The final assignment is $d = d / 10;$. Red annotations show green circles with 't1' and 't2' placed over the terms $d / 10$ in each assignment, suggesting temporary variable optimization.

Code Generation: Sixth phase of the compiler

- If the target language is machine code, then registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions to complete an operation



Code Generation: Sixth phase of the compiler

- Important consideration of code generation is the assignment of registers to hold variables.
- Choice of instructions involving registers, memory or a mix of the two

Symbol-Table Management: - Interaction with all the compiler's phases

- The symbol table is a data structure containing a record for each variable name (all symbols defined in the source program), with fields for the attributes of the name.
- The data structure is designed to help the compiler to identify and fetch the record for each name quickly
- To store or retrieve data from that record quickly
- Not part of the final code, but used as reference by all phases
- Generally created by lexical and syntax analyzer

Symbol-Table Management: - Interaction with all the compiler's phases

- attributes may provide information about the storage allocated for a name, its type, its scope, size, relative offset of variables
- Function or Procedure names, such things as the number and types of its arguments, the method of passing each argument and the return type

Error Handling and Recovery

- An important criteria for selecting the quality of the compiler
- For semantic errors, compiler can proceed
- For syntax errors, parser enters into erroneous state
- Needs to undo some processing already carried out by parser
- A few tokens may need to be discarded to reach a descent state
- Recovery is essential to provide a bunch of errors to the users

Compiler Phases vs Passes

- Several phases can be implemented as a single pass consist of reading an input file and writing an output file.

Compiler Phases vs Passes

- A typical multi-pass compiler looks like:
 - First pass: preprocessing, macro expansion
 - Second pass: syntax-directed translation, IR code generation
 - Third pass: optimization
 - Last pass: target machine code generation

Cousins of Compilers

- Preprocessors
- Assemblers
 - Compiler may produce assembly code instead of generating relocatable machine code directly.

Cousins of the Compiler

- Loaders and Linkers
 - Loader copies code and data into memory, allocates storage, setting protection bits, mapping virtual addresses, .. Etc
 - Linker handles relocation and resolves symbol references.
- Debugger

Lexical Analysis

Lexical Phase

- Scanning
 - Deletion of comments, and compaction of consecutive white space characters into one
- Lexical Analysis
 - Complex portion, to produce tokens from the output of the scanner

Lexical Analysis

- Input
 - program text (file)
- Output
 - sequence of tokens
- Read input file
- Identify language keywords and standard identifiers

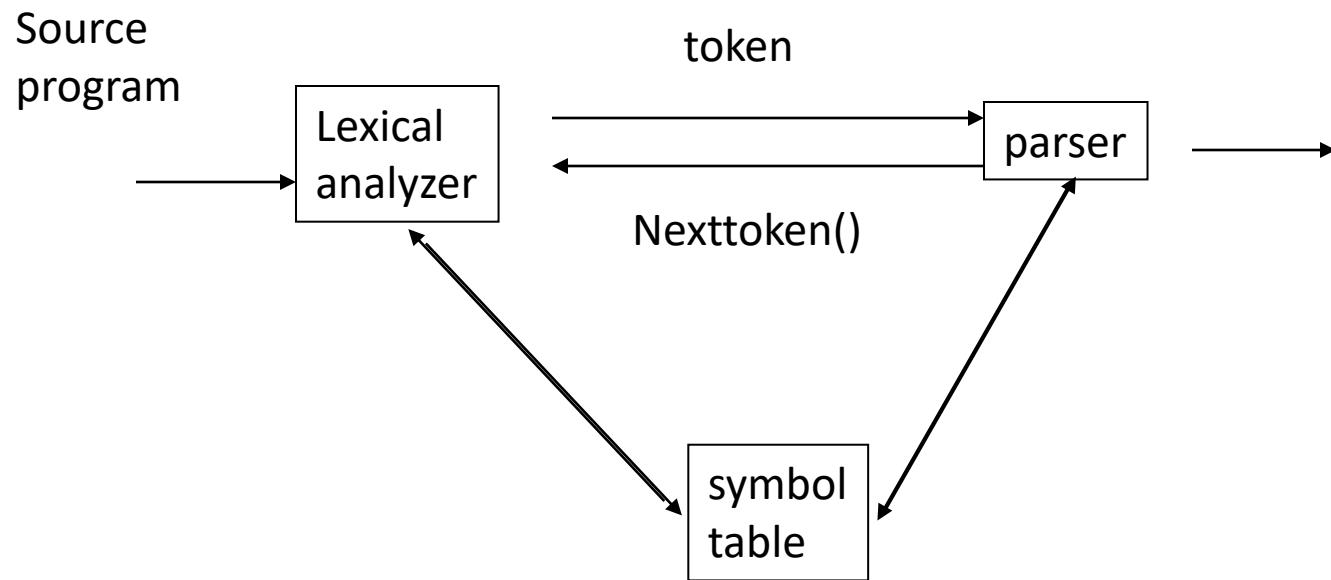
Lexical Analysis

- Handle include files and macros
- Count line numbers
- Remove whitespaces
- Report illegal symbols
- Create symbol table

Lexical Analyzer

- Lexical analyzer does not have to be an individual phase.
- But having a separate phase simplifies the design and improves the efficiency and portability.

Interaction of Lexical analyzer with parser



Why Lexical Analysis

- Simplifies the syntax analysis
 - And language definition
- Modularity / Portability
- Reusability
- Efficiency

Definitions

- Lexeme is a particular instant of a token.
- Token: a group of characters having a collective meaning.
 - token: identifier, lexeme: area, rate etc.
- Pattern: the rule describing how a token can be formed.
 - identifier: $([a-z] | [A-Z]) ([a-z] | [A-Z] | [0-9])^*$

Issues in lexical Analyzer

- How to identify tokens?
 - Patterns as RE, NFA, DFA
- How to recognize the tokens giving a token specification (how to implement the nexttoken() routine)?
 - Integrate the first two phases of the compiler

The Lexical Analysis Problem

- Given
 - A set of token descriptions
 - Token name
 - Regular expression defining the pattern for a lexeme
 - An input string
- Partition the strings into tokens
(class, value)

Lexical Analysis problem

- Ambiguity resolution
 - The longest matching token
 - Between two equal length tokens select the first

$a \mid \Rightarrow < \text{id}, z >$

a b

a $\leq b$

<

$= = > > =$

Example of Token

TOKEN	Description	Sample lexeme
if	Character i, f	If
else	Characters e, l, s, e	else
Comparison	< or > or < = or >= or == or !=	<=
id	Letter followed by letters and digits	Pi, score, a123
Number	Any numeric constant	3.14, 9.08
Literal	Anything within “ ”	“Seg fault”

Classes covering most of the tokens

- One token for each keyword. The pattern for a keyword is the same as the keyword itself.
- Tokens for the operators, either individually or in classes such as the token comparison
- One token representing all identifiers.
- One or more tokens representing constants, such as numbers and literal strings.
- Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Attributes for Tokens

- A pointer to the symbol-table entry in which the information about the token is kept

E.g E=M*C**2

<**id**, pointer to symbol-table entry for E>

<**assign_op**>

<**id**, pointer to symbol-table entry for M>

<**mult_op**>

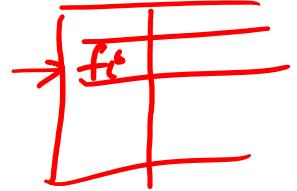
<**id**, pointer to symbol-table entry for C>

<**exp_op**>

<**num**, integer value 2>

Lexical Errors

⇒ A_i



- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error

else₂

- Ex: fi (a == f(x)) . . . if (condition)

- simplest recovery strategy is “panic mode” recovery

else₁ ✓

- Other possible error-recovery actions are:

else

else

else₁

- Delete one character from the remaining input.
- Insert a missing character into the remaining input.
- Replace a character by another character.
- Transpose two adjacent characters

fi
if

abc
abd
c

Strings and Languages

- An alphabet is any finite set of symbols
 - Typical examples of symbols are letters, digits, and punctuation
- A string over an alphabet is a finite sequence of symbols drawn from that alphabet
- The length of a string s , $|s|$, is the number of occurrences of symbols in s
- The empty string, denoted ϵ , is the string of length zero
- A language is any countable set of strings over some fixed alphabet

Regular Expressions

Basic patterns	Matching
x	The character x
.	Any character except newline
[xyz]	Any of the characters x, y, z
R?	An optional R

ϵ/R

Regular expression

R^*	Zero or more occurrences of R
R^+	One or more occurrences of R
R_1R_2	R_1 followed by R_2
$R_1 R_2$	Either R_1 or R_2
(R)	R itself

$$R^* = \bigcup_{i=0}^{\infty} R^i$$

$$\{ R^0, R^1, R^2, R^3, \dots, R^\infty \}$$

$$R = ab$$

$$R = a/b$$

$$R^* = \{ \epsilon, ab, abab, ababab, \dots \}$$

$$R^* = \{ \epsilon, a, b, \underbrace{aa, ab, ba, bb}, \dots \}$$

$$R^+ = \bigcup_{i=1}^{\infty} R^i$$

$$\{ R^1, R^2, R^3, \dots, R^\infty \}$$

Properties of Regular Expression

- $L(r) \cup L(s)$ is also a RE
- $L(r) \cap L(s)$ is also RE
- R^* is also RE if R is one
- If $\Sigma = \{a, b\}$, then
- $L_1 = a^* = \{\epsilon, a, aa, aaa, \dots\}$
- $L_2 = a \mid b = \{a, b\}$

Regular Expression

- Pascal language identifiers

$L(r) = \text{letter} (\text{letter} \mid \text{digit})^*$

Language for defining C language identifiers

- * has the highest precedence, followed by concatenation followed by |
- ϵ is a regular expression which is a string of length 0

Regular Definitions

- Names given to certain regular expressions and use these names later
- Regular definition is a sequence of the form
 $d_1 \rightarrow r_1, d_2 \rightarrow r_2, d_3 \rightarrow r_3\dots$
- Each d_i is a symbol not in the input alphabet
- Each r_i is a regular expression

$$\underline{d_1} \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$$d_3 \rightarrow r_3$$

$$\Sigma$$

$$\Sigma \cup d_1$$

$$\Sigma \cup d_1 \cup d_2$$

Regular Definitions

• letter $\rightarrow A \mid B \mid \dots \mid z \mid -$

• digit $\rightarrow 0 \mid 1 \mid 2 \mid 3 \dots \mid 9$

• id $\rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$

id $\rightarrow ([a-z] \mid [A-Z] \mid -) ([a-z][A-Z] \mid - \mid [0-9])^*$

Example

1234
1

10.23 ✓
10E2

- digit → 0 | 1 | ... | 9
- digits → digit digit*
- optionalFraction → .digits | ε
- optionalExponent → (E(+ | - | ε)digits) | ε
- number → digits optionalFraction optionalExponent

$$z = y + \underline{\underline{3.29}}$$

Token Recognition

if (expr) stmt if (expr) stmt else stmt

if ()
{ } ==

• Stmt \rightarrow if expr then Stmt | if expr then Stmt else Stmt | ϵ ✓

if ()
{ } ==

• expr \rightarrow term relop term | term

{ } else

• term \rightarrow id | number

if (a)

• id \rightarrow letter (letter|digits)*

{ } ==

• relop \rightarrow < | > | <= | >= | == | !=

if (a > b)

• number \rightarrow digits

{ } ==

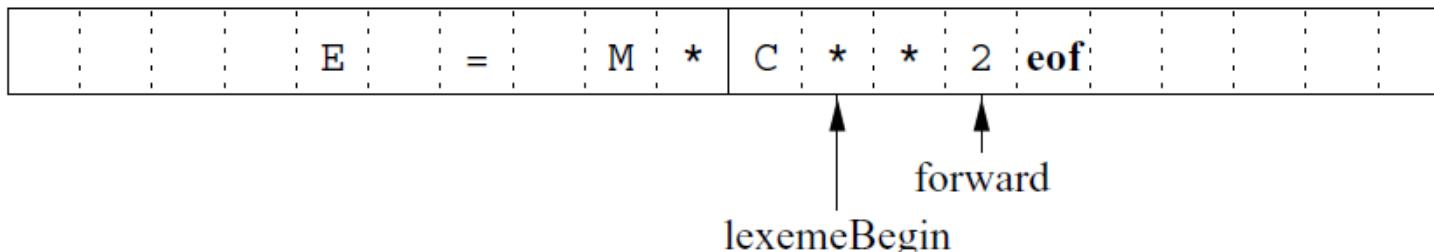
s \rightarrow s ; s | s

Input Buffering

- Have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme
 - Ex: can't determine the end of an identifier until we see a character that is not a letter or digit
 - Ex: In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=

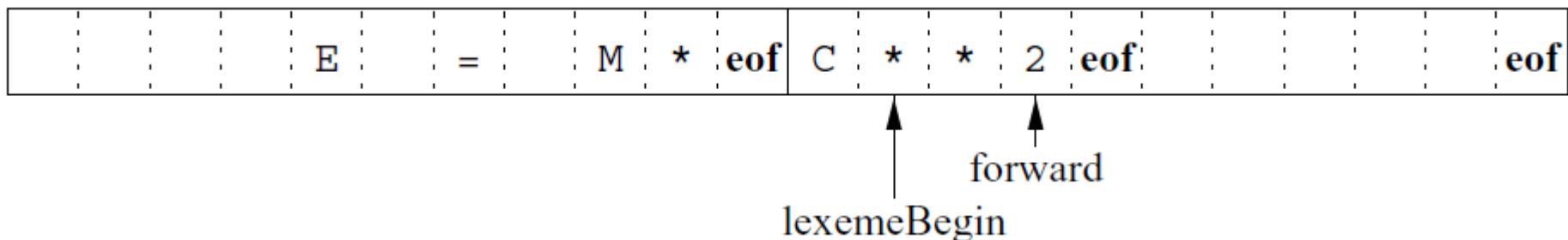
Buffer Pairs

- two buffers that are alternately reloaded
- Each buffer is of the same size N, and N is usually the size of a disk block
- Using one system read command we can read N characters into a buffer, rather than using one system call per character
- If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file
- Two pointers to the input are maintained:
 - *lexemeBegin*, marks the beginning of the current lexeme, whose extent we are attempting to determine
 - *forward* scans ahead until a pattern match is found



Sentinels

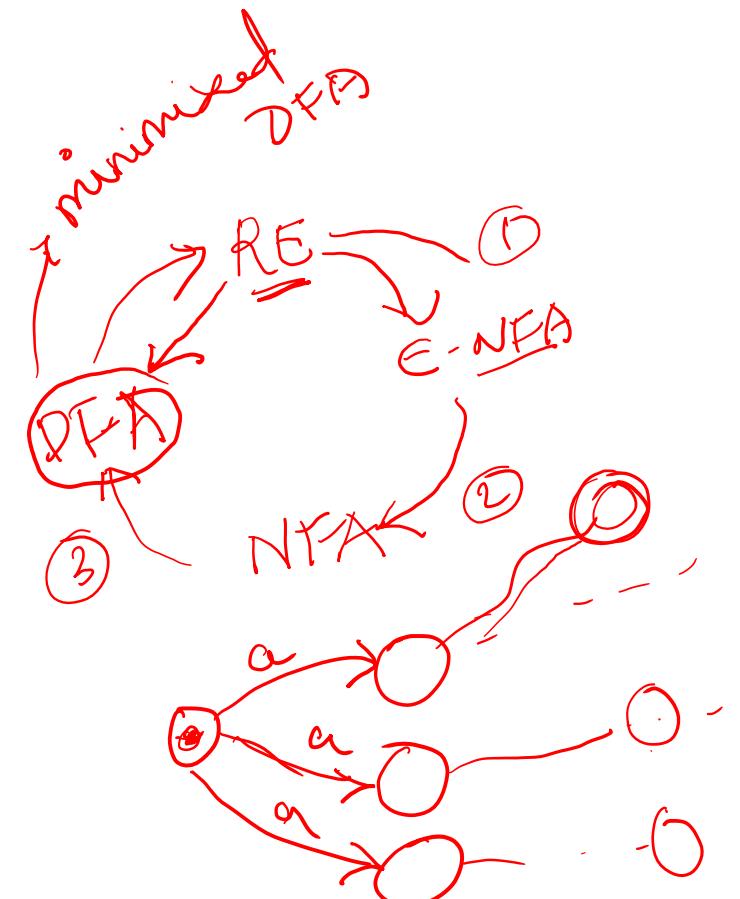
- Before advancing ***forward***, test whether end of one of the buffers is reached, if so, reload the other buffer from the input, and move ***forward*** to the beginning of the newly loaded buffer
- For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read
- Can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof
- Any eof that appears other than at the end of a buffer means end of input



```
switch ( *forward++ ) {
    case EOF:
        if (forward is at end of first buffer) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* EOF within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

Why Automata?

- It may be hard to specify regular expressions for certain constructs
 - Examples
 - Strings
 - Comments
- Writing automata may be easier
- Can combine both



Why Automata?

- Specify partial automata with regular expressions on the edges
 - No need to specify all states
 - Different actions at different states

Constructing Automaton from Specification

- Create a non-deterministic automaton (NDFA) from every regular expression
- Merge all the automata using epsilon moves (like the \mid construction)
- Construct a deterministic finite automaton (DFA)
 - State priority
- Minimize the automaton starting with separate accepting states

Finite Automata

- By default a Deterministic one.
- Five tuple representation
 $(Q, \Sigma, \delta, q_0, F)$, q_0 belongs to Q and F is a subset of Q
 δ is a mapping from $Q \times \Sigma$ to Q
- Every string has exactly one path and hence faster string matching

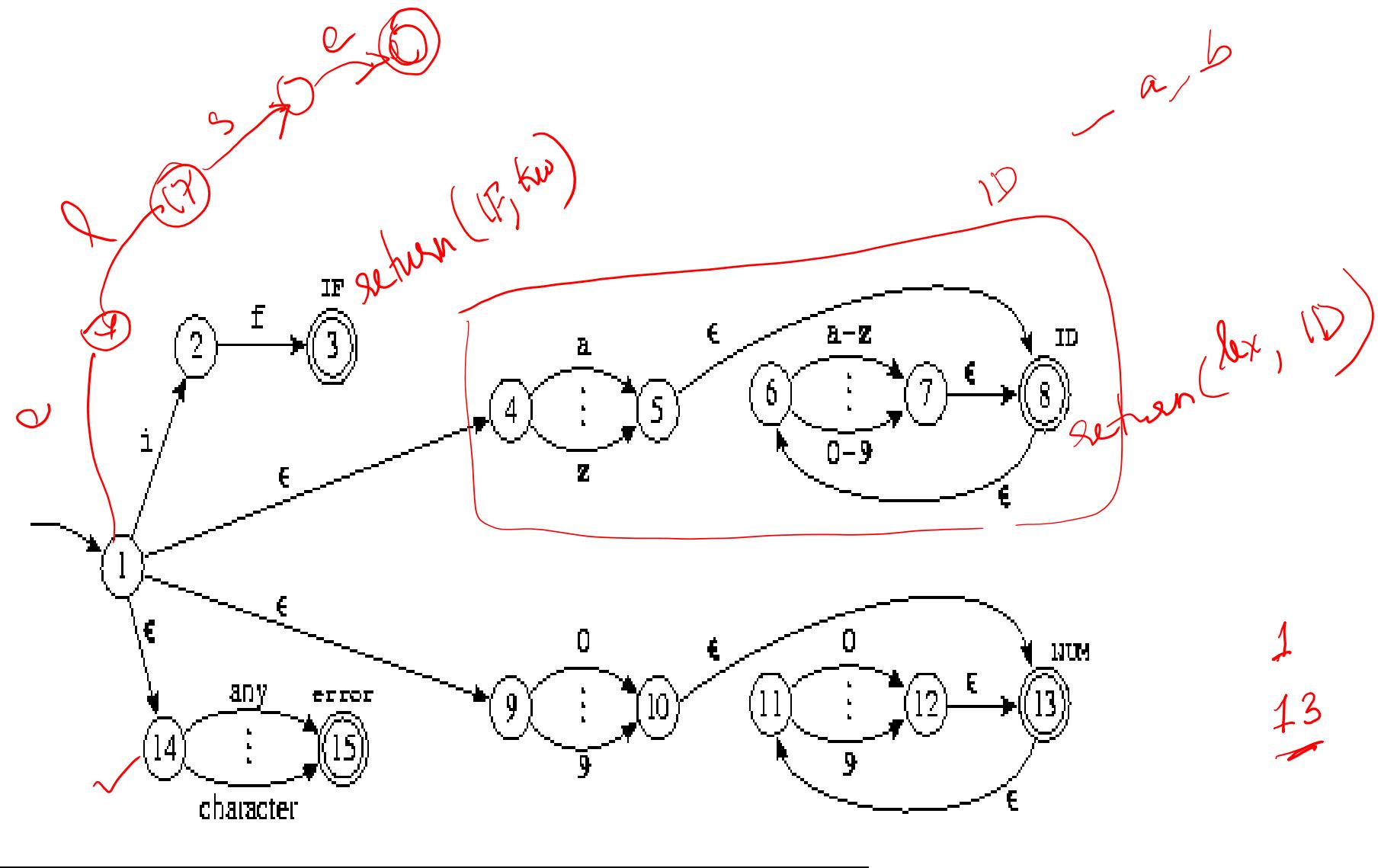
Non-deterministic Finite automata

- Same as deterministic, gives some flexibility.
- Five tuple representation
 $(Q, \Sigma, \delta, q_0, F)$, q_0 belongs to Q and F is a subset of Q
 δ is a mapping from $Q \times \Sigma$ to 2^Q
- More time for string matching as multiple paths exist.

Non-Deterministic Finite automata with ϵ

- Same as NFA. Still more flexible in allowing to change state without consuming any input symbol.
- δ is a mapping from $Q \times \Sigma \cup \underline{\{ \epsilon \}}$ to 2^Q
- Slower than NFA for string matching

Example

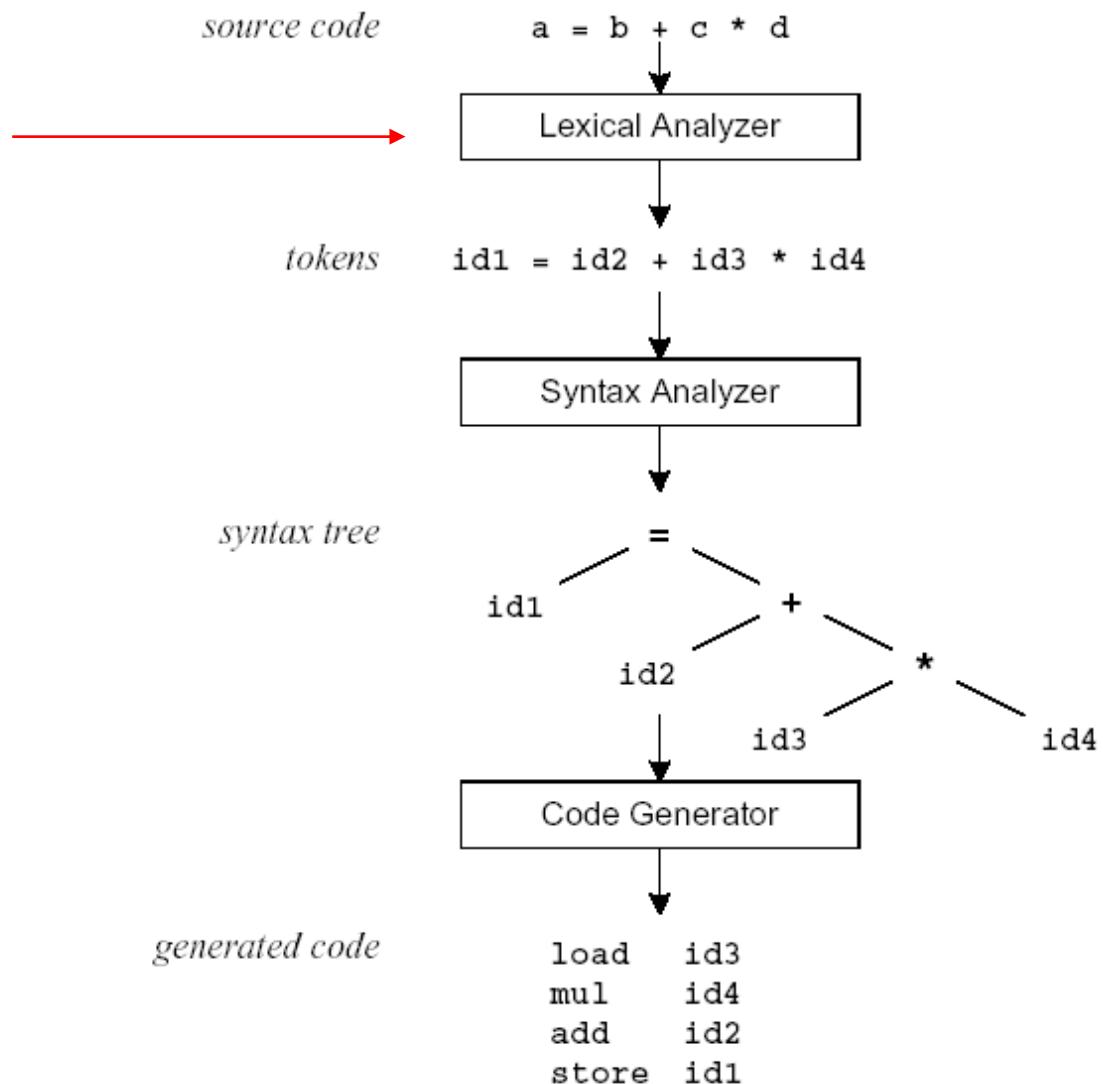


Summary

- The work involved in lexical phase
- Constructing Regular expression
- Introduction to DFA, NFA and NFA with ϵ

Lex tutorial

Compilation Sequence



What is Lex?

- The main job of a *lexical analyzer (scanner)* is to break up an input stream into more usable elements (*tokens*)

a = b + c * d;

ID ASSIGN ID PLUS ID MULT ID SEMI

- Lex is an utility to help you rapidly generate your scanners

Why a Tool?

- Starting from scratch is difficult
- Use by defining patterns

Standard tools

- LEX
- FLEX
- JLEX

Lex Source Program

- Lex source is a table of
 - regular expressions and
 - corresponding program fragments

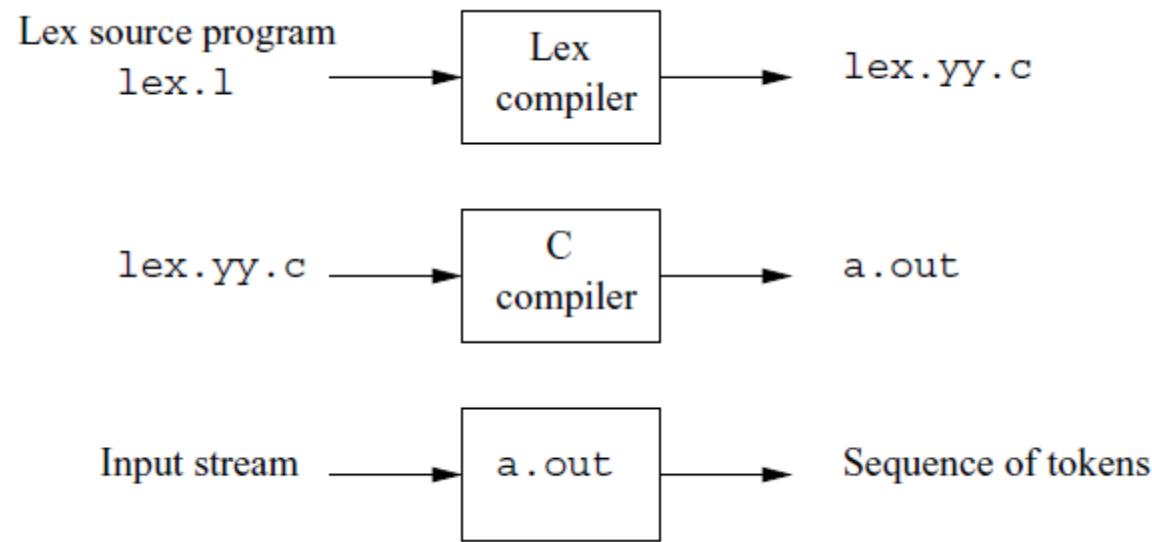
```
digit  [0-9]
letter [a-zA-Z]
%%
{letter}({letter}|{digit})*
\n
%%
main() {
    yylex();
}
```

```
printf("id: %s\n", yytext);
printf("new line\n");
```

Lex Source to C Program

- The table is translated to a C program (`lex.yy.c`) which
 - reads an input stream
 - partitioning the input into strings which match the given expressions and
 - copying it to an output stream if necessary

An Overview of Lex



Lex Source

- Lex source is separated into **three sections** by **%%** delimiters
- The general format of Lex source is

```
{definitions}                                (required)
%%                                          
{transition rules}
%%                                          
{user subroutines}                           (optional)
```

- The absolute minimum Lex program is thus

```
%%
```

Regular Expressions

Lex Regular Expressions (Extended Regular Expressions)

- A regular expression matches a set of strings
- Regular expression
 - Operators
 - Character classes
 - Arbitrary character
 - Optional expressions
 - Alternation and grouping
 - Context sensitivity
 - Repetitions and definitions

Operators

" \ [] ^ - ? . * + | () \$ / { } % < >

- If they are to be used as text characters, an escape should be used

\\$ = \$"

\\" = "\\\"

- Every character but *blank*, *tab* (\t), *newline* (\n) and the list above is always a text character

Character Classes []

- `[abc]` matches a single character, which may be `a`, `b`, or `c`
- Every operator meaning is ignored except `\` – and `^`
- e.g.

`[ab]` => a or b

`[a-z]` => a or b or c or ... or z

`[-+0-9]` => all the digits and the two signs

`[^a-zA-Z]` => any character which is not a letter

Arbitrary Character

- To match almost character, the operator character `.` is the class of all characters except newline
- `[\x40-\x176]` matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde~)

Optional & Repeated Expressions

- $a^?$ => zero or one instance of a
- a^* => zero or more instances of a
- a^+ => one or more instances of a
- E.g.
 - $ab^?c$ => ac or abc
 - $[a-z]^+$ => all strings of lower case letters
 - $[a-zA-Z][a-zA-Z0-9]^*$ => all alphanumeric strings with a leading alphabetic character

Precedence of Operators

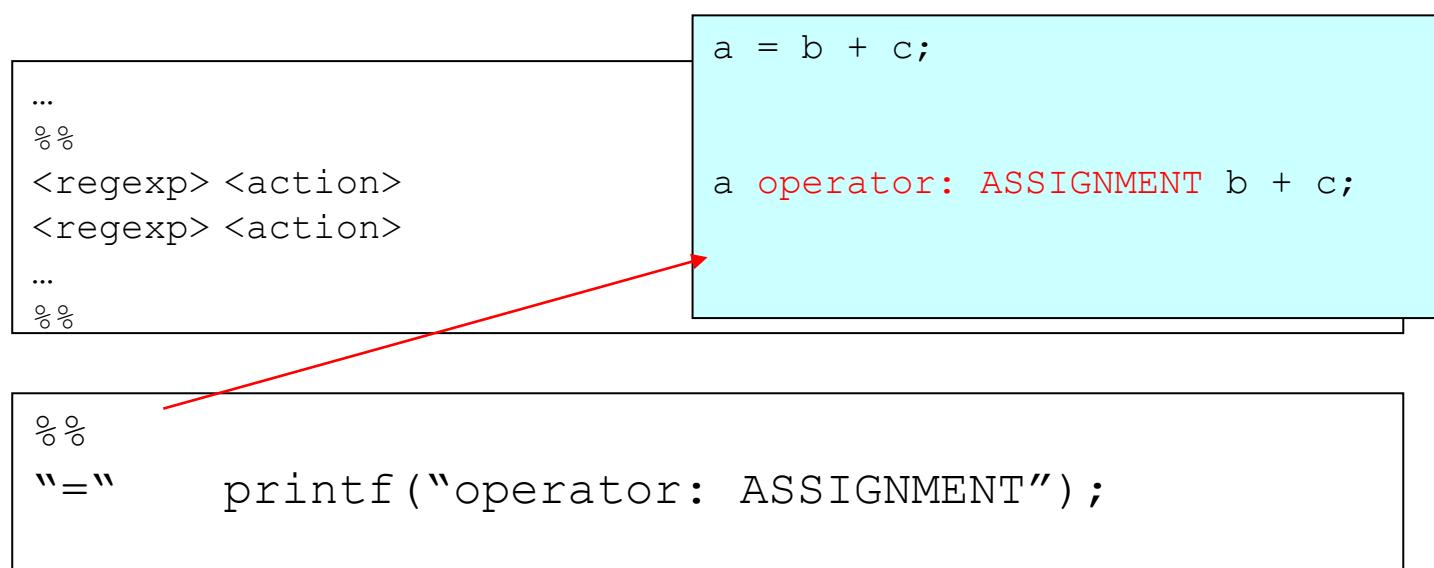
- Level of precedence
 - Kleene closure ($*$) , ?, +
 - concatenation
 - alternation ($|$)
- All operators are left associative.
- Ex: $a^*b \mid cd^* = ((a^*)b) \mid (c(d^*))$

Pattern Matching Primitives

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line / complement
\$	end of line
a b	a or b
(ab) +	one or more copies of ab (grouping)
[ab]	a or b
a { 3 }	3 instances of a
"a+b"	literal "a+b" (C escapes still work)

Recall: Lex Source

- Lex source is a table of
 - regular expressions and
 - corresponding program fragments (actions)



```
/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id          {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?)?{digit}+)??
%%
```

Transition Rules

- regexp <one or more blanks> action (C code);
- regexp <one or more blanks> { actions (C code) }
- A null statement ; will ignore the input (no actions)
[\t\n] ;
 - Causes the three spacing characters to be ignored

```
a = b + c;  
d = b * c;
```

↓ ↓

```
a=b+c; d=b*c;
```

```
{ws}          {/* no action and no return */}  
if           {return(IF);}  
then          {return(THEN);}  
else          {return(ELSE);}  
{id}          {yyval = (int) installID(); return(ID);}  
{number}      {yyval = (int) installNum(); return(NUMBER);}  
"<"          {yyval = LT; return(RELOP);}  
"<="         {yyval = LE; return(RELOP);}  
"="          {yyval = EQ; return(RELOP);}  
"<>"        {yyval = NE; return(RELOP);}  
">"          {yyval = GT; return(RELOP);}  
">="         {yyval = GE; return(RELOP);}
```

%%

Transition Rules (cont'd)

- Four special options for actions:
|, ECHO;, BEGIN, and REJECT;
- | indicates that the action for this rule is from the action for the next rule
 - [\t\n] ;
 - " " |
 - "\t" |
 - "\n" ;
- The unmatched token is using a default action that ECHO from the input to the output

Transition Rules (cont'd)

- REJECT
 - Go do the next alternative

```
...
%%
pink    {npink++; REJECT;}
ink     {nink++; REJECT;}
pin     {npin++; REJECT;}
. |
\n      ;
%%
...
```

Lex Predefined Variables

- **yytext** -- a string containing the lexeme
- **yyleng** -- the length of the lexeme
- **yyin** -- the input stream pointer
 - the default input of default main() is **stdin**
- **yyout** -- the output stream pointer
 - the default output of default main() is **stdout**.
- **./a.out < inputFile > outputFile**
- E.g.

[a-zA-Z] +	printf ("%s", yytext);
[a-zA-Z] +	ECHO;
[a-zA-Z] +	{words++; chars += yyleng; }

Lex Library Routines

- **yylex()**
 - The default main() contains a call of yylex()
- **yymore()**
 - return the next token
- **yyless(n)**
 - retain the first n characters in yytext
- **yywarp()**
 - is called whenever Lex reaches an end-of-file
 - The default yywarp() always returns 1

Review of Lex Predefined Variables

Name	Function
char *yytext	pointer to matched string
int yyleng	length of matched string
FILE *yyin	input stream pointer
FILE *yyout	output stream pointer
int yylex(void)	call to invoke lexer, returns token
char* yymore(void)	return the next token
int yyless(int n)	retain the first n characters in yytext
int yywrap(void)	wrapup, return 1 if done, 0 if not done
ECHO	write matched string
REJECT	go to the next alternative rule
INITIAL	initial start condition
BEGIN	condition switch start condition

User Subroutines Section

- You can use your Lex routines in the same ways you use routines in other programming languages.

```
% {
    void foo();
%
letter      [a-zA-Z]
%%
{letter}+    foo();
%%
...
void foo()  {
    ...
}
```

```
int installID() /* function to install the lexeme, whose
first character is pointed to by yytext,
and whose length is yyleng, into the
symbol table and return a pointer
thereto */
}

int installNum() /* similar to installID, but puts numerical
constants into a separate table */
}
```

User Subroutines Section (cont'd)

- The section where **main()** is placed

```
% {  
    int counter = 0;  
}  
letter [a-zA-Z]  
  
%%  
{letter}+      {printf("a word\n"); counter++; }  
  
%%  
main ()  {  
    yylex();  
    printf("There are total %d words\n", counter);  
}
```

Usage

- To run Lex on a source file, type

```
lex scanner.l
```

- It produces a file named lex.yy.c which is a C program for the lexical analyzer.

- To compile lex.yy.c, type

```
cc lex.yy.c -lI
```

- To run the lexical analyzer program, type

```
./a.out < inputFile
```

Versions of Lex

- AT&T -- lex
http://www.combo.org/lex_yacc_page/lex.html
- Lex on different machines is not created equal.

Example

```
int num_lines = 0;  
%%  
\n    ++num_lines;  
.  
;  
%%  
main()  
{ yylex();  
printf( "# of lines = %d\n", num_lines); }
```

Example

```
%{int s=0,c=0,l=0;%}
%%
[ \t] {s++;}
[a-zA-Z0-9] {c++;}
[\n] {l++;}
EOF {printf("\n\t\t Characters = %d \n\n\t Words = %d Lines =%d",c,s,l);exit(0);}
%%
int main(int argc , char *argv[])
{system("clear");
yyin=fopen(argv[1],"r");    //printf("Enter the String=\n");
yylex();
printf("\n\t\t Characters = %d \n\n\t Words = %d Lines =%d",c,s,l);
fclose(yyin);
}
```

A COMPACT GUIDE TO LEX & YACC

by Tom Niemann

epaperpress.com

Contents

Contents	2
Preface	3
Introduction	4
Lex	6
Theory	6
Practice	7
Yacc	11
Theory	11
Practice, Part I	12
Practice, Part II	15
Calculator	18
Description	18
Include File	21
Lex Input	22
Yacc Input	23
Interpreter	27
Compiler	28
Graph	30
More Lex	34
Strings	34
Reserved Words	35
Debugging Lex	35
More Yacc	36
Recursion	36
If-Else Ambiguity	37
Error Messages	38
Inherited Attributes	39
Embedded Actions	39
Debugging Yacc	40
Bibliography	41

Preface

This document explains how to construct a compiler using lex and yacc. Lex and yacc are tools used to generate lexical analyzers and parsers. I assume you can program in C, and understand data structures such as linked-lists and trees.

The introduction describes the basic building blocks of a compiler and explains the interaction between lex and yacc. The next two sections describe lex and yacc in more detail. With this background we can construct a sophisticated calculator. Conventional arithmetic operations and control statements, such as **if-else** and **while**, are implemented. With minor changes we will convert the calculator into a compiler for a stack-based machine. The remaining sections discuss issues that commonly arise in compiler writing. Source code for examples may be downloaded from the web site listed below.

Permission to reproduce portions of this document is given provided the web site listed below is referenced, and no additional restrictions apply. Source code, when part of a software project, may be used freely without reference to the author.

Tom Niemann
Portland, Oregon
web site: epaperpress.com

Introduction

Before 1975 writing a compiler was a very time-consuming process. Then Lesk [1975] and Johnson [1975] published papers on lex and yacc. These utilities greatly simplify compiler writing. Implementation details for lex and yacc may be found in Aho [1986]. Lex and yacc are available from

- Mortice Kern Systems (MKS), at www.mks.com,
- GNU flex and bison, at www.gnu.org,
- Cygwin, at www.cygwin.com

The version from MKS is a high-quality commercial product that retails for about \$300US. GNU software is free. Output from flex may be used in a commercial product, and, as of version 1.24, the same is true for bison. Cygwin is a 32-bit Windows ports of the GNU software. In fact Cygwin is a port of the Unix operating system to Windows, complete with compilers gcc and g++. To install download and run the setup executable. Under devel install bison, flex, gcc-g++, and make. Under editors install vim. Lately I've been using flex and bison under the cygwin environment.

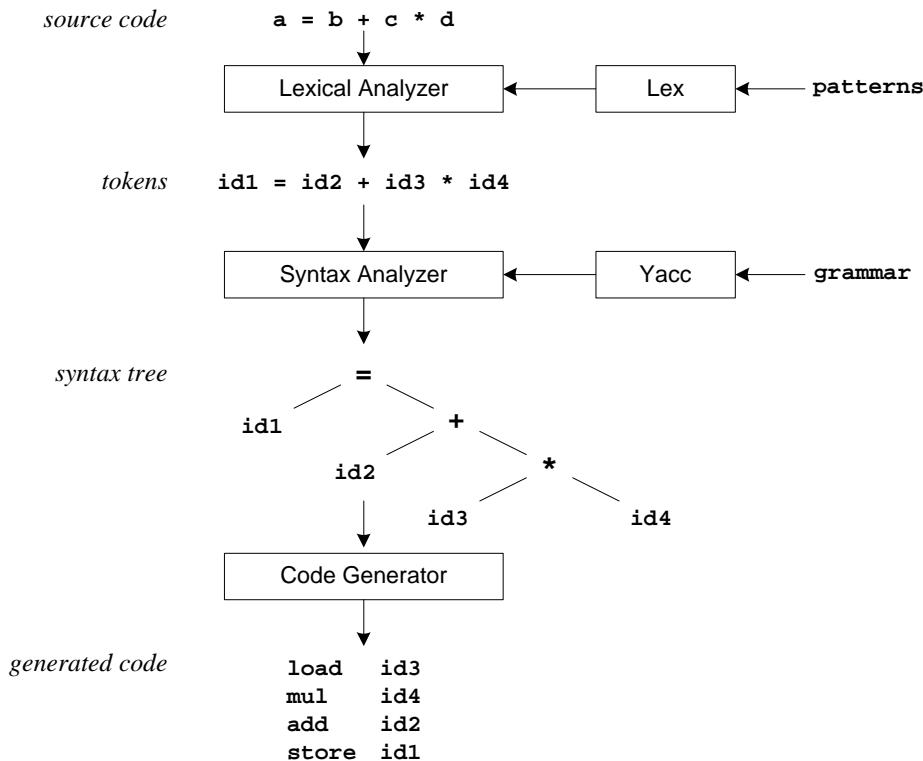


Figure 1: Compilation Sequence

You code patterns and input them to lex. It will read your patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing. This is illustrated in Figure 1.

When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and

location of the variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

You code a grammar and input it to yacc. Yacc will read your grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure on the tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language.

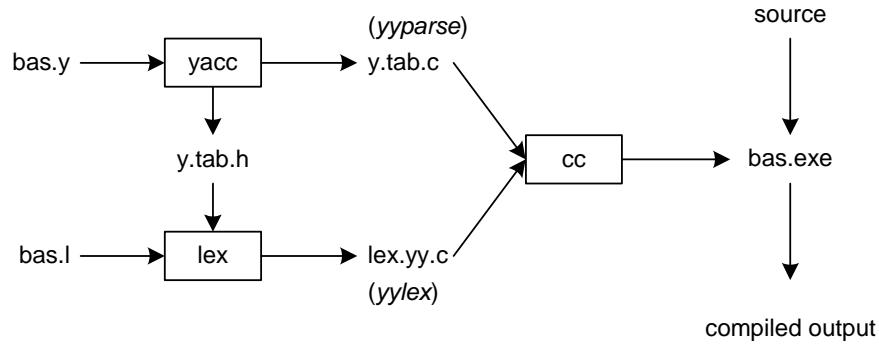


Figure 2: Building a Compiler with Lex/Yacc

Figure 2 illustrates the file naming conventions used by lex and yacc. We'll assume our goal is to write a BASIC compiler. First, we need to specify all pattern matching rules for lex (**bas.l**) and grammar rules for yacc (**bas.y**). Commands to create our compiler, **bas.exe**, are listed below:

```

yacc -d bas.y          # create y.tab.h, y.tab.c
lex bas.l              # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe # compile/link
  
```

Yacc reads the grammar descriptions in **bas.y** and generates a syntax analyzer (parser), that includes function **yyparse**, in file **y.tab.c**. Included in file **bas.y** are token declarations. The **-d** option causes yacc to generate definitions for tokens and place them in file **y.tab.h**. Lex reads the pattern descriptions in **bas.l**, includes file **y.tab.h**, and generates a lexical analyzer, that includes function **yylex**, in file **lex.yy.c**.

Finally, the lexer and parser are compiled and linked together to form the executable, **bas.exe**. From **main**, we call **yyparse** to run the compiler. Function **yyparse** automatically calls **yylex** to obtain each token.

Lex

Theory

The first phase in a compiler reads the input source and converts strings in the source to tokens. Using regular expressions, we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

```
letter(letter|digit)*
```

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the “*” operator
- alternation, expressed by the “|” operator
- concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state, and one or more final or accepting states.

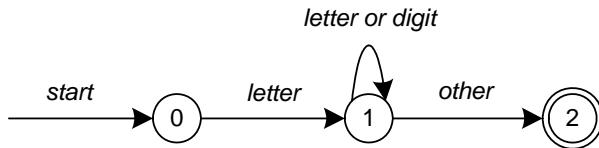


Figure 3: Finite State Automaton

In Figure 3, state 0 is the start state, and state 2 is the accepting state. As characters are read, we make a transition from one state to another. When the first letter is read, we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit, we transition to state 2, the accepting state. Any FSA may be expressed as a computer program. For example our 3-state machine is easily programmed:

```
start: goto state0

state0: read c
        if c = letter goto state1
        goto state0

state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2

state2: accept string
```

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next *input* character, and *current state*, the next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of lex's limitations. For example, lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "(", we push it on the stack. When a ")" is encountered, we match it with the top of the stack, and pop the stack. However lex only has states and transitions between states. Since it has no stack, it is not well suited for parsing nested structures. Yacc augments an FSA with a stack, and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

Practice

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
\$	end of line
a b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcabc abcabcabc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a b]	one of: a, , b
a b	one of: a, b

Table 2: Pattern Matching Examples

Regular expressions in lex are composed of metacharacters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class, normal operators lose their meaning.

Two operators allowed in a character class are the hyphen (“-”) and circumflex (“^”). When used between two characters, the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Input to Lex is divided into three sections, with `%%` dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output, one character at a time. The first `%%` is always required, as there must always be a rules section. However, if we don’t specify any rules, then the default action is to match everything and copy it to output. Defaults for input and output are `stdin` and `stdout`, respectively. Here is the same example, with defaults explicitly coded:

```
%%
/* match everything except newline */
. ECHO;
/* match newline */
\n ECHO;

%%
int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline), and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, “.” and “\n”, with an `ECHO` action associated for each pattern. Several macros and variables are predefined by lex. `ECHO` is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, `ECHO` is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable `yytext` is a pointer to the matched string (NULL-terminated), and `yyleng` is the length of the matched string. Variable `yyout` is the output file, and defaults to `stdout`. Function `yywrap` is called by lex when input is exhausted. Return 1 if you are done, or 0 if more processing is required. Every C program requires a `main` function. In this case, we simply call `yylex`, the main entry-point for lex. Some implementations of lex include copies of `main` and `yywrap` in a library,

eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Name	Function
<code>int yylex(void)</code>	call to invoke lexer, returns token
<code>char *yytext</code>	pointer to matched string
<code>yylen</code>	length of matched string
<code>yyval</code>	value associated with token
<code>int yywrap(void)</code>	wrapup, return 1 if done, 0 if not done
<code>FILE *yyout</code>	output file
<code>FILE *yyin</code>	input file
<code>INITIAL</code>	initial start condition
<code>BEGIN</code>	condition switch start condition
<code>ECHO</code>	write matched string

Table 3: Lex Predefined Variables

Here is a program that does nothing at all. All input is matched, but no action is associated with any pattern, so there will be no output.

```
%%
.
\n
```

The following example prepends line numbers to each line in a file. Some implementations of lex redefine and calculate **yylineno**. The input file for lex is **yyin**, and defaults to **stdin**.

```
%{
    int yylineno;
%
%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file, and must be bracketed with "%{" and "%}" markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```

digit      [0-9]
letter     [A-Za-z]
%{
    int count;
%
%}
/* match identifier */
{letter}({letter}|{digit})*           count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}

```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces (**{letter}**) to distinguish them from literals. When we have a match in the rules section, the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):

```

%{
    int nchar, nword, nline;
%
%}
\n          { nline++; nchar++; }
[^ \t\n]+   { nword++, nchar += yyleng; }
.          { nchar++; }
%%
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    return 0;
}

```

Yacc

Theory

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique was pioneered by John Backus and Peter Naur, and used to describe ALGOL60. A BNF grammar can be used to express *context-free* languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

```
E -> E + E
E -> E * E
E -> id
```

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as **E** (expression) are nonterminals. Terms such as **id** (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

```
E -> E * E      (r2)
--> E * z        (r3)
--> E + E * z    (r1)
--> E + y * z    (r3)
--> x + y * z    (r3)
```

At each step we expanded a term, replacing the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression, we actually need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar, we need to reduce an expression to a single nonterminal. This is known as *bottom-up* or *shift-reduce* parsing, and uses a stack for storing terms. Here is the same derivation, but in reverse order:

```
1   . x + y * z    shift
2   x . + y * z    reduce(r3)
3   E . + y * z    shift
4   E + . y * z    shift
5   E + y . * z    reduce(r3)
6   E + E . * z    shift
7   E + E * . z    shift
8   E + E * z .    reduce(r3)
9   E + E * E .    reduce(r2)      emit multiply
10  E + E .         reduce(r1)      emit add
11  E .             accept
```

Terms to the left of the dot are on the stack, while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production, we replace the matched tokens on the stack with the lhs of the production. Conceptually, the matched tokens of the rhs are popped off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a *handle*, and we are *reducing* the handle to the lhs of the production. This process continues until we have shifted all input to the stack, and only the starting nonterminal remains on the stack. In step 1 we shift the **x** to the stack. Step 2 applies rule r3 to the stack, changing **x** to **E**. We continue shifting and reducing, until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply

instruction. Similarly, the add instruction is emitted in step 10. Thus, multiply has a higher precedence than addition.

Consider, however, the shift at step 6. Instead of shifting, we could have reduced, applying rule r1. This would result in addition having a higher precedence than multiplication. This is known as a *shift-reduce* conflict. Our grammar is *ambiguous*, as there is more than one possible derivation that will yield the expression. In this case, operator precedence is affected. As another example, associativity in the rule

```
E -> E + E
```

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar, or supply yacc with directives that indicate which operator has precedence. The latter method is simpler, and will be demonstrated in the practice section.

The following grammar has a *reduce-reduce* conflict. With an **id** on the stack, we may reduce to **T**, or reduce to **E**.

```
E -> T
E -> id
T -> id
```

Yacc takes a default action when there is a conflict. For shift-reduce conflicts, yacc will shift. For reduce-reduce conflicts, it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

Practice, Part I

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Input to yacc is divided into three sections. The definitions section consists of token declarations, and C code bracketed by "%{" and "%}". The BNF grammar is placed in the rules section, and user subroutines are added in the subroutines section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

```
%token INTEGER
```

This definition declares an **INTEGER** token. When we run yacc, it generates a parser in file **y.tab.c**, and also creates an include file, **y.tab.h**:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex includes this file and utilizes the definitions for token values. To obtain tokens, yacc calls **yylex**. Function **yylex** has a return type of int, and returns the token. Values associated with the token are returned by lex in variable **yylval**. For example,

```
[0-9]+      {
            yylval = atoi(yytext);
            return INTEGER;
}
```

would store the value of the integer in **yylval**, and return token **INTEGER** to yacc. The type of **yylval** is determined by **YYSTYPE**. Since the default type is integer, this works well in this case. Token values 0-255 are reserved for character values. For example, if you had a rule such as

```
[-+]      return *yytext;      /* return operator */
```

the character value for minus or plus is returned. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator. Generated token values typically start around 258, as lex reserves several values for end-of-file and error processing. Here is the complete lex input specification for our calculator:

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%
[0-9]+      {
            yylval = atoi(yytext);
            return INTEGER;
}

[-+\n]      return *yytext;

[\t]        ; /* skip whitespace */

.          yyerror("invalid character");

%%
int yywrap(void) {
    return 1;
}
```

Internally, yacc maintains two stacks in memory; a parse stack and a value stack. The parse stack contains terminals and nonterminals, and represents the current parsing state. The value stack is an array of **YYSTYPE** elements, and associates a value with each element in the parse stack. For example, when lex returns an **INTEGER** token, yacc shifts this token to the parse stack. At the same time, the corresponding **yylval** is shifted to the value stack. The parse and value stacks are always synchronized, so finding a value related to a token on the stack is easily accomplished. Here is the yacc input specification for our calculator:

```

%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
}

%token INTEGER

%%

program:
    program expr '\n'          { printf("%d\n", $2); }
    |
;

expr:
    INTEGER                  { $$ = $1; }
    | expr '+' expr          { $$ = $1 + $3; }
    | expr '-' expr          { $$ = $1 - $3; }
    ;
;

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}

```

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified, followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces.

By utilizing left-recursion, we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected, we print the value of the expression. When we apply the rule

```
expr: expr '+' expr          { $$ = $1 + $3; }
```

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case, we pop “**expr '+' expr**” and push “**expr**”. We have reduced the stack by popping three terms off the stack, and pushing back one term. We may reference positions in the value stack in our C code by specifying “\$1” for the first term on the right-hand side of the production, “\$2” for the second, and so on. “\$\$” designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. Thus, the parse and value stacks remain synchronized.

Numeric values are initially entered on the stack when we reduce from **INTEGER** to **expr**. After **INTEGER** is shifted to the stack, we apply the rule

```
expr: INTEGER      { $$ = $1; }
```

The **INTEGER** token is popped off the parse stack, followed by a push of **expr**. For the value stack, we pop the integer value off the stack, and then push it back on again. In other words, we do nothing. In fact, this is the default action, and need not be specified. Finally, when a newline is encountered, the value associated with **expr** is printed.

In the event of syntax errors, yacc calls the user-supplied function **yyerror**. If you need to modify the interface to **yyerror**, you can alter the canned file that yacc includes to fit your needs. The last function in our yacc specification is **main** ... in case you were wondering where it was. This example still has an ambiguous grammar. Yacc will issue shift-reduce warnings, but will still process the grammar using shift as the default operation.

Practice, Part II

In this section we will extend the calculator from the previous section to incorporate some new functionality. New features include arithmetic operators multiply, and divide. Parentheses may be used to over-ride operator precedence, and single-character variables may be specified in assignment statements. The following illustrates sample input and calculator output:

```
user: 3 * (4 + 5)
calc: 27
user: x = 3 * (4 + 5)
user: y = 5
user: x
calc: 27
user: y
calc: 5
user: x + 2*y
calc: 37
```

The lexical analyzer returns **VARIABLE** and **INTEGER** tokens. For variables, **yylval** specifies an index to **sym**, our symbol table. For this program, **sym** merely holds the value of the associated variable. When **INTEGER** tokens are returned, **yylval** contains the number scanned. Here is the input specification for lex:

```
%{
    #include <stdlib.h>
    void yyerror(char *);
    #include "y.tab.h"
%}

%%

/* variables */
[a-z]      {
            yylval = *yytext - 'a';
            return VARIABLE;
        }

/* integers */
[0-9]+     {
            yylval = atoi(yytext);
            return INTEGER;
        }

/* operators */
[--+()=/*\n] { return *yytext; }

/* skip whitespace */
[ \t]        ;

/* anything else is an error */
.           yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

The input specification for yacc follows. The tokens for **INTEGER** and **VARIABLE** are utilized by yacc to create **#defines** in **y.tab.h** for use in lex. This is followed by definitions for the arithmetic operators. We may specify **%left**, for left-associative, or **%right**, for right associative. The last definition listed has the highest precedence. Thus, multiplication and division have higher precedence than addition and subtraction. All four operators are left-associative. Using this simple technique, we are able to disambiguate our grammar.

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'

%{
void yyerror(char *);
int yylex(void);
int sym[26];
%}

%%
program:
    program statement '\n'
    |
;

statement:
    expr
    | VARIABLE '=' expr
    ;
;

expr:
    INTEGER
    | VARIABLE
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '(' expr ')'
    ;
;

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}

int main(void) {
    yyparse();
    return 0;
}
```

Calculator

Description

This version of the calculator is substantially more complex than previous versions. Major changes include control constructs such as **if-else** and **while**. In addition, a syntax tree is constructed during parsing. After parsing, we walk the syntax tree to produce output. Two versions of the tree walk routine are supplied:

- an interpreter that executes statements during the tree walk, and
- a compiler that generates code for a hypothetical stack-based machine.

To make things more concrete, here is a sample program,

```
x = 0;
while (x < 3) {
    print x;
    x = x + 1;
}
```

with output for the interpretive version,

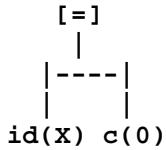
```
0
1
2
```

and output for the compiler version, and

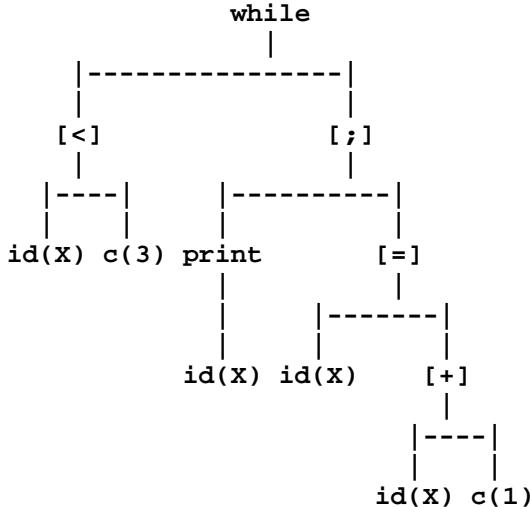
```
push    0
pop     x
L000:
    push    x
    push    3
    compLT
    jz     L001
    push    x
    print
    push    x
    push    1
    add
    pop     x
    jmp     L000
L001:
```

a version that generates a syntax tree.

Graph 0:



Graph 1:



The include file contains declarations for the syntax tree and symbol table. The symbol table, **sym**, allows for single-character variable names. A node in the syntax tree may hold a constant (**conNodeType**), an identifier (**idNodeType**), or an internal node with an operator (**oprNodeType**). A union encapsulates all three variants, and **nodeType.type** is used to determine which structure we have.

The lex input file contains patterns for **VARIABLE** and **INTEGER** tokens. In addition, tokens are defined for 2-character operators such as **EQ** and **NE**. Single-character operators are simply returned as themselves.

The yacc input file defines **YYSTYPE**, the type of **yyval**, as

```
%union {  
    int iValue; /* integer value */  
    char sIndex; /* symbol table index */  
    nodeType *nPtr; /* node pointer */  
};
```

This causes the following to be generated in **y.tab.h**:

```
typedef union {  
    int iValue; /* integer value */  
    char sIndex; /* symbol table index */  
    nodeType *nPtr; /* node pointer */  
} YYSTYPE;  
extern YYSTYPE yyval;
```

Constants, variables, and nodes can be represented by `yylval` in the parser's value stack. A more accurate representation of decimal integers is given below. This is similar to C/C++ where integers that begin with 0 are classified as octal.

```

0      {
        yylval.iValue = atoi(yytext);
        return INTEGER;
    }

[1-9][0-9]* {
        yylval.iValue = atoi(yytext);
        return INTEGER;
    }

```

Notice the type definitions

```

%token <iValue> INTEGER
%type <nPtr> expr

```

This binds `expr` to `nPtr`, and `INTEGER` to `iValue` in the `YYSTYPE` union. This is required so that yacc can generate the correct code. For example, the rule

```
expr: INTEGER { $$ = con($1); }
```

should generate the following code. Note that `yyvsp[0]` addresses the top of the value stack, or the value associated with `INTEGER`.

```
yyval.nPtr = con(yyvsp[0].iValue);
```

The unary minus operator is given higher priority than binary operators as follows:

```

%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

```

The `%nonassoc` indicates no associativity is implied. It is frequently used in conjunction with `%prec` to specify precedence of a rule. Thus, we have

```
expr: '-' expr %prec UMINUS { $$ = node(UMINUS, 1, $2); }
```

indicating that the precedence of the rule is the same as the precedence of token `UMINUS`. And, as defined above, `UMINUS` has higher precedence than the other operators. A similar technique is used to remove ambiguity associated with the if-else statement (see If-Else Ambiguity).

The syntax tree is constructed bottom-up, allocating the leaf nodes when variables and integers are reduced. When operators are encountered, a node is allocated and pointers to previously allocated nodes are entered as operands.

After the tree is built, function `ex` is called to do a depth-first walk of the syntax tree. A depth-first walk visits nodes in the order that they were originally allocated. This results in operators being applied in the order that they were encountered during parsing. Three versions of `ex` are included: an interpretive version, a compiler version, and a version that generates a syntax tree.

Include File

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* constants */
typedef struct {
    int value;                      /* value of constant */
} conNodeType;

/* identifiers */
typedef struct {
    int i;                          /* subscript to sym array */
} idNodeType;

/* operators */
typedef struct {
    int oper;                      /* operator */
    int nops;                      /* number of operands */
    struct nodeTypeTag *op[1];     /* operands (expandable) */
} oprNodeType;

typedef struct nodeTypeTag {
    nodeEnum type;                 /* type of node */

    /* union must be last entry in nodeType */
    /* because oprNodeType may dynamically increase */
    union {
        conNodeType con;          /* constants */
        idNodeType id;           /* identifiers */
        oprNodeType opr;         /* operators */
    };
} nodeType;

extern int sym[26];
```

Lex Input

```
%{
#include <stdlib.h>
#include "calc3.h"
#include "y.tab.h"
void yyerror(char *);
%}

%%
[a-z]      {
    yylval.sIndex = *yytext - 'a';
    return VARIABLE;
}

0          {
    yylval.iValue = atoi(yytext);
    return INTEGER;
}

[1-9][0-9]* {
    yylval.iValue = atoi(yytext);
    return INTEGER;
}

[-()<>=+*/;{}.] {
    return *yytext;
}

">="         return GE;
"<="         return LE;
"=="         return EQ;
"!="         return NE;
"while"      return WHILE;
"if"          return IF;
"else"        return ELSE;
"print"       return PRINT;

[ \t\n]+      ;      /* ignore whitespace */

.              yyerror("Unknown character");
%%

int yywrap(void) {
    return 1;
}
```

Yacc Input

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calc3.h"

/* prototypes */
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);

void yyerror(char *s);
int sym[26]; /* symbol table */
%}

%union {
    int iValue; /* integer value */
    char sIndex; /* symbol table index */
    nodeType *nPtr; /* node pointer */
};

%token <iValue> INTEGER
%token <sIndex> VARIABLE
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE

%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type <nPtr> stmt expr stmt_list
```

```

%%

program:
    function          { exit(0); }
;

function:
    function stmt      { ex($2); freeNode($2); }
| /* NULL */
;

stmt:
    ';'              { $$ = opr(';', 2, NULL, NULL); }
| expr ';'          { $$ = $1; }
| PRINT expr ';'   { $$ = opr(PRINT, 1, $2); }
| VARIABLE '=' expr ';' { $$ = opr('=', 2, id($1), $3); }
| WHILE '(' expr ')' stmt { $$ = opr(WHILE, 2, $3, $5); }
| IF '(' expr ')' stmt %prec IFX { $$ = opr(IF, 2, $3, $5); }
| IF '(' expr ')' stmt ELSE stmt
    { $$ = opr(IF, 3, $3, $5, $7); }
| '{' stmt_list '}'
;

stmt_list:
    stmt            { $$ = $1; }
| stmt_list stmt   { $$ = opr(';', 2, $1, $2); }
;

expr:
    INTEGER          { $$ = con($1); }
| VARIABLE          { $$ = id($1); }
| '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
| expr '+' expr    { $$ = opr('+', 2, $1, $3); }
| expr '-' expr    { $$ = opr('-', 2, $1, $3); }
| expr '*' expr    { $$ = opr('*', 2, $1, $3); }
| expr '/' expr    { $$ = opr('/', 2, $1, $3); }
| expr '<' expr    { $$ = opr('<', 2, $1, $3); }
| expr '>' expr    { $$ = opr('>', 2, $1, $3); }
| expr GE expr     { $$ = opr(GE, 2, $1, $3); }
| expr LE expr     { $$ = opr(LE, 2, $1, $3); }
| expr NE expr     { $$ = opr(NE, 2, $1, $3); }
| expr EQ expr     { $$ = opr(EQ, 2, $1, $3); }
| '(' expr ')'
;

```

```

%%

#define SIZEOF_NODETYPE ((char *)&p->con - (char *)p)

nodeType *con(int value) {
    nodeType *p;
    size_t nodeSize;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(conNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeCon;
    p->con.value = value;

    return p;
}

nodeType *id(int i) {
    nodeType *p;
    size_t nodeSize;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(idNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeId;
    p->id.i = i;

    return p;
}

```

```

nodeType *opr(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    size_t nodeSize;
    int i;

    /* allocate node */
    nodeSize = SIZEOF_NODETYPE + sizeof(oprNodeType) +
               (nops - 1) * sizeof(nodeType*);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeOpr;
    p->opr.oper = oper;
    p->opr.nops = nops;
    va_start(ap, nops);
    for (i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
    va_end(ap);
    return p;
}

void freeNode(nodeType *p) {
    int i;

    if (!p) return;
    if (p->type == typeOpr) {
        for (i = 0; i < p->opr.nops; i++)
            freeNode(p->opr.op[i]);
    }
    free (p);
}

void yyerror(char *s) {
    fprintf(stdout, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}

```

Interpreter

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"

int ex(nodeType *p) {
    if (!p) return 0;
    switch(p->type) {
        case typeCon:      return p->con.value;
        case typeId:       return sym[p->id.i];
        case typeOpr:
            switch(p->opr.oper) {
                case WHILE:   while(ex(p->opr.op[0]))
                                ex(p->opr.op[1]); return 0;
                case IF:       if (ex(p->opr.op[0]))
                                ex(p->opr.op[1]);
                               else if (p->opr.nops > 2)
                                ex(p->opr.op[2]);
                               return 0;
                case PRINT:    printf("%d\n",      ex(p->opr.op[0]));
                               return 0;
                case ';':     ex(p->opr.op[0]);
                               return ex(p->opr.op[1]);
                case '=':     return sym[p->opr.op[0]->id.i] =
                                ex(p->opr.op[1]);
                case UMINUS:  return -ex(p->opr.op[0]);
                case '+':    return ex(p->opr.op[0]) + ex(p->opr.op[1]);
                case '-':    return ex(p->opr.op[0]) - ex(p->opr.op[1]);
                case '*':    return ex(p->opr.op[0]) * ex(p->opr.op[1]);
                case '/':    return ex(p->opr.op[0]) / ex(p->opr.op[1]);
                case '<':    return ex(p->opr.op[0]) < ex(p->opr.op[1]);
                case '>':    return ex(p->opr.op[0]) > ex(p->opr.op[1]);
                case GE:      return ex(p->opr.op[0]) >= ex(p->opr.op[1]);
                case LE:      return ex(p->opr.op[0]) <= ex(p->opr.op[1]);
                case NE:      return ex(p->opr.op[0]) != ex(p->opr.op[1]);
                case EQ:      return ex(p->opr.op[0]) == ex(p->opr.op[1]);
            }
        }
    return 0;
}
```

Compiler

```
#include <stdio.h>
#include "calc3.h"
#include "y.tab.h"

static int lbl;

int ex(nodeType *p) {
    int lbl1, lbl2;

    if (!p) return 0;
    switch(p->type) {
        case typeCon:
            printf("\tpush\t%d\n", p->con.value);
            break;
        case typeId:
            printf("\tpush\t%c\n", p->id.i + 'a');
            break;
        case typeOp:
            switch(p->opr.oper) {
                case WHILE:
                    printf("L%03d:\n", lbl1 = lbl++);
                    ex(p->opr.op[0]);
                    printf("\tjz\tL%03d\n", lbl2 = lbl++);
                    ex(p->opr.op[1]);
                    printf("\tjmp\tL%03d\n", lbl1);
                    printf("L%03d:\n", lbl2);
                    break;
                case IF:
                    ex(p->opr.op[0]);
                    if (p->opr.nops > 2) {
                        /* if else */
                        printf("\tjz\tL%03d\n", lbl1 = lbl++);
                        ex(p->opr.op[1]);
                        printf("\tjmp\tL%03d\n", lbl2 = lbl++);
                        printf("L%03d:\n", lbl1);
                        ex(p->opr.op[2]);
                        printf("L%03d:\n", lbl2);
                    } else {
                        /* if */
                        printf("\tjz\tL%03d\n", lbl1 = lbl++);
                        ex(p->opr.op[1]);
                        printf("L%03d:\n", lbl1);
                    }
                    break;
                case PRINT:
                    ex(p->opr.op[0]);
                    printf("\tprint\n");
                    break;
            }
    }
}
```

```

    case '=':
        ex(p->opr.op[1]);
        printf("\ttop\t%c\n", p->opr.op[0]->id.i + 'a');
        break;
    case UMINUS:
        ex(p->opr.op[0]);
        printf("\tneg\n");
        break;
    default:
        ex(p->opr.op[0]);
        ex(p->opr.op[1]);
        switch(p->opr.oper) {
            case '+': printf("\tadd\n"); break;
            case '-': printf("\tsub\n"); break;
            case '*': printf("\tmul\n"); break;
            case '/': printf("\tdiv\n"); break;
            case '<': printf("\tcompLT\n"); break;
            case '>': printf("\tcompGT\n"); break;
            case GE:   printf("\tcompGE\n"); break;
            case LE:   printf("\tcompLE\n"); break;
            case NE:   printf("\tcompNE\n"); break;
            case EQ:   printf("\tcompEQ\n"); break;
        }
    }
    return 0;
}

```

Graph

```
/* source code courtesy of Frank Thomas Braun */

#include <stdio.h>
#include <string.h>

#include "calc3.h"
#include "y.tab.h"

int del = 1; /* distance of graph columns */
int eps = 3; /* distance of graph lines */

/* interface for drawing (can be replaced by "real" graphic using GD or
other) */
void graphInit (void);
void graphFinish();
void graphBox (char *s, int *w, int *h);
void graphDrawBox (char *s, int c, int l);
void graphDrawArrow (int c1, int l1, int c2, int l2);

/* recursive drawing of the syntax tree */
void exNode (nodeType *p, int c, int l, int *ce, int *cm);

/**********************/

/* main entry point of the manipulation of the syntax tree */
int ex (nodeType *p) {
    int rte, rtm;

    graphInit ();
    exNode (p, 0, 0, &rte, &rtm);
    graphFinish();
    return 0;
}

/*c----cm---ce----> drawing of leaf-nodes
 l leaf-info
 */

/*c-----cm-----ce----> drawing of non-leaf-nodes
 l           node-info
 *
 *
 *
 *      |
 *      -----
 *      |       |
 *      v       v
 * child1  child2 ...   child-n
 *      che     che        che
 *      cs      cs         cs
 *
 */


```

```

void exNode
(
    nodeType *p,
    int c, int l,           /* start column and line of node */
    int *ce, int *cm         /* resulting end column and mid of node */
)
{
    int w, h;             /* node width and height */
    char *s;              /* node text */
    int cbar;             /* "real" start column of node (centred above
subnodes) */
    int k;                /* child number */
    int che, chm;          /* end column and mid of children */
    int cs;                /* start column of children */
    char word[20];         /* extended node text */

    if (!p) return;

    strcpy (word, "????"); /* should never appear */
    s = word;
    switch(p->type) {
        case typeCon: sprintf (word, "c(%d)", p->con.value); break;
        case typeId:   sprintf (word, "id(%c)", p->id.i + 'A'); break;
        case typeOpr:
            switch(p->opr.oper){
                case WHILE:      s = "while"; break;
                case IF:         s = "if";     break;
                case PRINT:      s = "print";  break;
                case ';':        s = "[;]";   break;
                case '=':        s = "[=]";   break;
                case UMINUS:    s = "[_]";   break;
                case '+':        s = "[+]";   break;
                case '-':        s = "[-]";   break;
                case '*':        s = "[*]";   break;
                case '/':        s = "[/]";   break;
                case '<':       s = "[<]";  break;
                case '>':       s = "[>]";  break;
                case GE:         s = "[>=]"; break;
                case LE:         s = "[<=]"; break;
                case NE:         s = "[!=]"; break;
                case EQ:         s = "[==]"; break;
            }
            break;
    }

    /* construct node text box */
    graphBox (s, &w, &h);
    cbar = c;
    *ce = c + w;
    *cm = c + w / 2;

    /* node is leaf */
    if (p->type == typeCon || p->type == typeId || p->opr.nops == 0) {
        graphDrawBox (s, cbar, 1);
        return;
    }
}

```

```

/* node has children */
cs = c;
for (k = 0; k < p->opr.nops; k++) {
    exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
    cs = che;
}

/* total node width */
if (w < che - c) {
    cbar += (che - c - w) / 2;
    *ce = che;
    *cm = (c + che) / 2;
}

/* draw node */
graphDrawBox (s, cbar, 1);

/* draw arrows (not optimal: children are drawn a second time) */
cs = c;
for (k = 0; k < p->opr.nops; k++) {
    exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
    graphDrawArrow (*cm, l+h, chm, l+h+eps-1);
    cs = che;
}
}

/* interface for drawing */

#define lmax 200
#define cmax 200

char graph[lmax][cmax]; /* array for ASCII-Graphic */
int graphNumber = 0;

void graphTest (int l, int c)
{
    int ok;
    ok = 1;
    if (l < 0) ok = 0;
    if (l >= lmax) ok = 0;
    if (c < 0) ok = 0;
    if (c >= cmax) ok = 0;
    if (ok) return;
    printf ("\n+++error: l=%d, c=%d not in drawing rectangle 0, 0 ... %d, %d",
           l, c, lmax, cmax);
    exit (1);
}

void graphInit (void) {
    int i, j;
    for (i = 0; i < lmax; i++) {
        for (j = 0; j < cmax; j++) {
            graph[i][j] = ' ';
        }
    }
}

```

```

void graphFinish() {
    int i, j;
    for (i = 0; i < lmax; i++) {
        for (j = cmax-1; j > 0 && graph[i][j] == ' '; j--);
        graph[i][cmax-1] = 0;
        if (j < cmax-1) graph[i][j+1] = 0;
        if (graph[i][j] == ' ') graph[i][j] = 0;
    }
    for (i = lmax-1; i > 0 && graph[i][0] == 0; i--);
    printf ("\n\nGraph %d:\n", graphNumber++);
    for (j = 0; j <= i; j++) printf ("\n%s", graph[j]);
    printf("\n");
}

void graphBox (char *s, int *w, int *h) {
    *w = strlen (s) + del;
    *h = 1;
}

void graphDrawBox (char *s, int c, int l) {
    int i;
    graphTest (l, c+strlen(s)-1+del);
    for (i = 0; i < strlen (s); i++) {
        graph[l][c+i+del] = s[i];
    }
}

void graphDrawArrow (int c1, int l1, int c2, int l2) {
    int m;
    graphTest (l1, c1);
    graphTest (l2, c2);
    m = (l1 + l2) / 2;
    while (l1 != m) {
        graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
    }
    while (c1 != c2) {
        graph[l1][c1] = '-'; if (c1 < c2) c1++; else c1--;
    }
    while (l1 != l2) {
        graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
    }
    graph[l1][c1] = '|';
}

```

More Lex

Strings

Quoted strings frequently appear in programming languages. Here is one way to match a string in lex:

```
%{
    char *yyval;
    #include <string.h>
}
%%
\"[^"\n]*\"[\n] {
    yyval = strdup(yytext+1);
    if (yyval[yylen-2] != '\"')
        warning("improperly terminated string");
    else
        yyval[yylen-2] = 0;
    printf("found '%s'\n", yyval);
}
```

The above example ensures that strings don't cross line boundaries, and removes enclosing quotes. If we wish to add escape sequences, such as \n or \\", start states simplify matters:

```
%{
char buf[100];
char *s;
%}
%STRING

%%
\"                  { BEGIN STRING; s = buf; }
<STRING>\\n      { *s++ = '\n'; }
<STRING>\\t      { *s++ = '\t'; }
<STRING>\\\"     { *s++ = '\"'; }
<STRING>\"      {
    *s = 0;
    BEGIN 0;
    printf("found '%s'\n", buf);
}
<STRING>\\n      { printf("invalid string"); exit(1); }
<STRING>.        { *s++ = *yytext; }
```

Exclusive start state **STRING** is defined in the definition section. When the scanner detects a quote, the **BEGIN** macro shifts lex into the **STRING** state. Lex stays in the **STRING** state, recognizing only patterns that begin with **<STRING>**, until another **BEGIN** is executed. Thus, we have a mini-environment for scanning strings. When the trailing quote is recognized, we switch back to state 0, the initial state.

Reserved Words

If your program has a large collection of reserved words, it is more efficient to let lex simply match a string, and determine in your own code whether it is a variable or reserved word. For example, instead of coding

```
"if"           return IF;
"then"         return THEN;
"else"         return ELSE;

{letter}({letter}|{digit})* {
    yylval.id = symLookup(yytext);
    return IDENTIFIER;
}
```

where **symLookup** returns an index into the symbol table, it is better to detect reserved words and identifiers simultaneously, as follows:

```
{letter}({letter}|{digit})* {
    int i;

    if ((i = resWord(yytext)) != 0)
        return (i);
    yylval.id = symLookup(yytext);
    return (IDENTIFIER);
}
```

This technique significantly reduces the number of states required, and results in smaller scanner tables.

Debugging Lex

Lex has facilities that enable debugging. This feature may vary with different versions of lex, so you should consult documentation for details. The code generated by lex in file **lex.y.c** includes debugging statements that are enabled by specifying command-line option “**-d**”. Debug output in flex (a GNU version of lex) may be toggled on and off by setting **yy_flex_debug**. Output includes the rule applied and corresponding matched text. If you’re running lex and yacc together, specify the following in your yacc input file:

```
extern int yy_flex_debug;
int main(void) {
    yy_flex_debug = 1;
    yyparse();
}
```

Alternatively, you may write your own debug code by defining functions that display information for the token value, and each variant of the **yylval** union. This is illustrated in the following example. When **DEBUG** is defined, the debug functions take effect, and a trace of tokens and associated values is displayed.

```
%union {
    int ivalue;
    ...
};

%{
#endif DEBUG
```

```

int dbgToken(int tok, char *s) {
    printf("token %s\n", s);
    return tok;
}
int dbgTokenIvalue(int tok, char *s) {
    printf("token %s (%d)\n", s, yyval.intValue);
    return tok;
}
#define RETURN(x) return dbgToken(x, #x)
#define RETURN_ivalue(x) return dbgTokenIvalue(x, #x)
#else
#define RETURN(x) return(x)
#define RETURN_ivalue(x) return(x)
#endif
}

%%
[0-9]+      {
    yyval.intValue = atoi(yytext);
    RETURN_ivalue(INTEGER);
}

"if"        RETURN(IF);
"else"       RETURN(ELSE);

```

More Yacc

Recursion

When specifying a list, we may do so using left recursion,

```

list:
    item
    | list ',' item
;
```

or right recursion:

```

list:
    item
    | item ',' list
```

If right recursion is used, all items on the list are pushed on the stack. After the last item is pushed, we start reducing. With left recursion, we never have more than three terms on the stack, since we reduce as we go along. For this reason, it is advantageous to use left recursion.

If-Else Ambiguity

A shift-reduce conflict that frequently occurs involves the **if-else** construct. Assume we have the following rules:

```
stmt:  
  IF expr stmt  
  | IF expr stmt ELSE stmt  
  ...
```

and the following state:

```
IF expr stmt IF expr stmt . ELSE stmt
```

We need to decide if we should shift the **ELSE**, or reduce the **IF expr stmt** at the top of the stack. If we shift, then we have

```
IF expr stmt IF expr stmt . ELSE stmt  
IF expr stmt IF expr stmt ELSE . stmt  
IF expr stmt IF expr stmt ELSE stmt .  
IF expr stmt stmt .
```

where the second **ELSE** is paired with the second **IF**. If we reduce, we have

```
IF expr stmt IF expr stmt . ELSE stmt  
IF expr stmt stmt . ELSE stmt  
IF expr stmt . ELSE stmt  
IF expr stmt ELSE . stmt  
IF expr stmt ELSE stmt .
```

where the second **ELSE** is paired with the first **IF**. Modern programming languages pair an **ELSE** with the most recent unpaired **IF**, so the former behavior is expected. This works well with yacc, since default behavior, when a shift-reduce conflict is encountered, is to shift.

Although yacc does the right thing, it also issues a shift-reduce warning message. To remove the message, give **IF-ELSE** a higher precedence than the simple **IF** statement:

```
%nonassoc IFX  
%nonassoc ELSE  
  
stmt:  
  IF expr stmt %prec IFX  
  | IF expr stmt ELSE stmt
```

Error Messages

A nice compiler gives the user meaningful error messages. For example, not much information is conveyed by the following message:

```
syntax error
```

If we track the line number in lex, then we can at least give the user a line number:

```
void yyerror(char *s) {
    fprintf(stderr, "line %d: %s\n", yylineno, s);
}
```

When yacc discovers a parsing error, default action is to call **yyerror**, and then return from yylex with a return value of one. A more graceful action flushes the input stream to a statement delimiter, and continues to scan:

```
stmt:
  ';' 
  | expr ';'
  | PRINT expr ';'
  | VARIABLE '=' expr ;
  | WHILE '(' expr ')' stmt
  | IF '(' expr ')' stmt %prec IFX
  | IF '(' expr ')' stmt ELSE stmt
  | '{' stmt_list '}'
  | error ;
  | error '}'
;
```

The error token is a special feature of yacc that will match all input until the token following error is found. For this example, when yacc detects an error in a statement it will call **yyerror**, flush input up to the next semicolon or brace, and resume scanning.

Inherited Attributes

The examples so far have used synthesized attributes. At any point in a syntax tree we can determine the attributes of a node based on the attributes of its children. Consider the rule

```
expr: expr '+' expr      { $$ = $1 + $3; }
```

Since we are parsing bottom-up, the values of both operands are available, and we can determine the value associated with the left-hand side. An inherited attribute of a node depends on the value of a parent or sibling node. The following grammar defines a C variable declaration:

```
decl: type varlist
type: INT | FLOAT
varlist:
    VAR          { setType($1, $0); }
    | varlist ',' VAR { setType($3, $0); }
```

Here is a sample parse:

```
. INT VAR
INT . VAR
type . VAR
type VAR .
type varlist .
decl .
```

When we reduce **VAR** to **varlist**, we should annotate the symbol table with the type of the variable. However, the **type** is buried in the stack. This problem is resolved by indexing back into the stack. Recall that **\$1** designates the first term on the right-hand side. We can index backwards, using **\$0**, **\$-1**, and so on. In this case, **\$0** will do just fine. If you need to specify a token type, the syntax is **\$<tokentype>0**, angle brackets included. In this particular example, care must be taken to ensure that type always precedes **varlist**.

Embedded Actions

Rules in yacc may contain embedded actions:

```
list: item1 { do_item1($1); } item2 { do_item2($3); } item3
```

Note that the actions take a slot in the stack, so **do_item2** must use **\$3** to reference **item2**. Actually, this grammar is transformed by yacc into the following:

```
list: item1 _rule01 item2 _rule02 item3
_rule01: { do_item1($0); }
_rule02: { do_item2($0); }
```

Debugging Yacc

Yacc has facilities that enable debugging. This feature may vary with different versions of yacc, so you should consult documentation for details. The code generated by yacc in file **y.tab.c** includes debugging statements that are enabled by defining **YYDEBUG** and setting it to a non-zero value. This may also be done by specifying command-line option "-t". With **YYDEBUG** properly set, debug output may be toggled on and off by setting **yydebug**. Output includes tokens scanned and shift/reduce actions.

```
%{
#define YYDEBUG 1
%
%%
...
%%
int main(void) {
    #if YYDEBUG
        yydebug = 1;
    #endif
    yylex();
}
```

In addition, you can dump the parse states by specifying command-line option "-v". States are dumped to file **y.output**, and are often useful when debugging a grammar. Alternatively, you can write your own debug code by defining a **TRACE** macro, as illustrated below. When **DEBUG** is defined, a trace of reductions, by line number, is displayed.

```
%{
#ifndef DEBUG
#define TRACE printf("reduce at line %d\n", __LINE__);
#else
#define TRACE
#endif
%}

statement_list:
    statement
        { TRACE $$ = $1; }
    | statement_list statement
        { TRACE $$ = newNode(';', 2, $1, $2); }
;
```

Bibliography

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman [1986]. [Compilers, Principles, Techniques and Tools](#). Addison-Wesley, Reading, Massachusetts.

Gardner, Jim, Chris Retterath and Eric Gisin [1988]. MKS Lex & Yacc. Mortice Kern Systems Inc., Waterloo, Ontario, Canada.

Johnson, Stephen C. [1975]. Yacc: Yet Another Compiler Compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey. A PDF version is available at ePaperPress.

Lesk, M. E. and E. Schmidt [1975]. Lex – A Lexical Analyzer Generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey. A PDF version is available at ePaperPress.

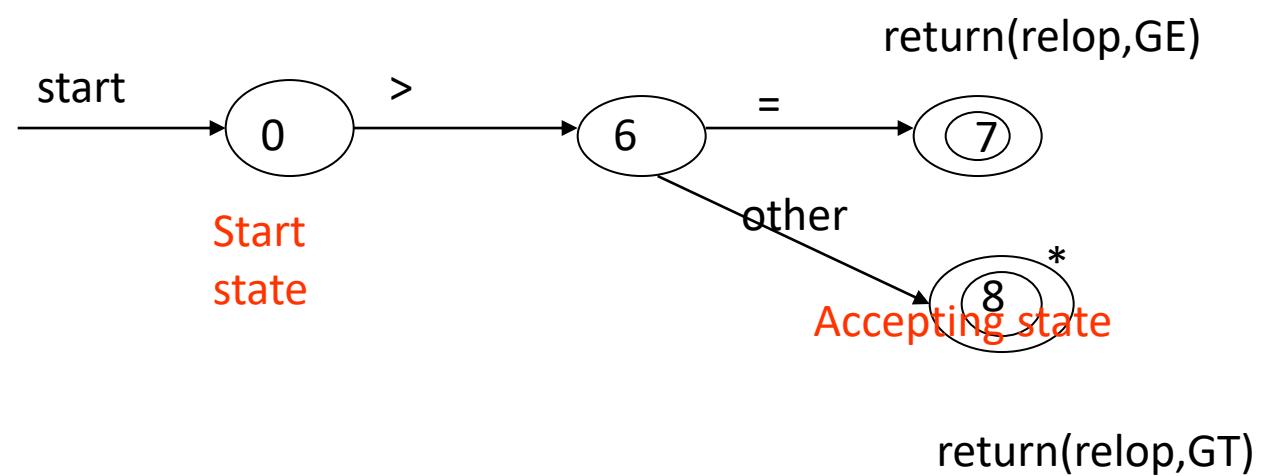
Levine, John R., Tony Mason and Doug Brown [1992]. [Lex & Yacc](#). O'Reilly & Associates, Inc. Sebastopol, California.

Lexical-Analyser: Automata

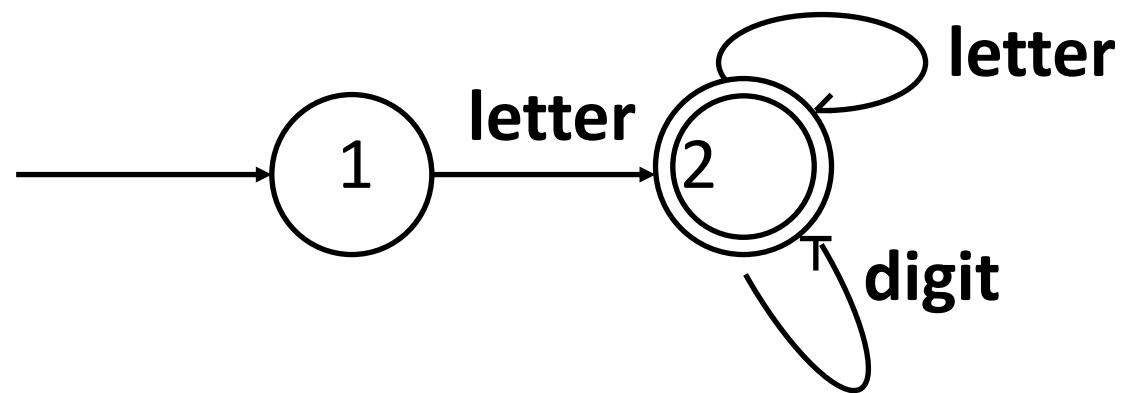
Automata – Transition Diagrams

- Transition Diagram(Stylized flowchart)
 - Depict the actions that take place when a lexical analyzer is called by the parser to get the next token

Example NFA



Example for Identifier



- Which represent the rule:
identifier=letter(letter|digit)*

Finite Automata

- By default a Deterministic one.
- Five tuple representation
 $(Q, \Sigma, \delta, q_0, F)$, q_0 belongs to Q and F is a subset of Q
 δ is a mapping from $Q \times \Sigma$ to Q
- Every string has exactly one path and hence faster string matching

DFA

- In a DFA, no state has an ϵ -transition
- In a DFA, for each state s and input symbol a , there is at most one edge labeled a leaving s
- To describe a FA, we use the transition graph or transition table

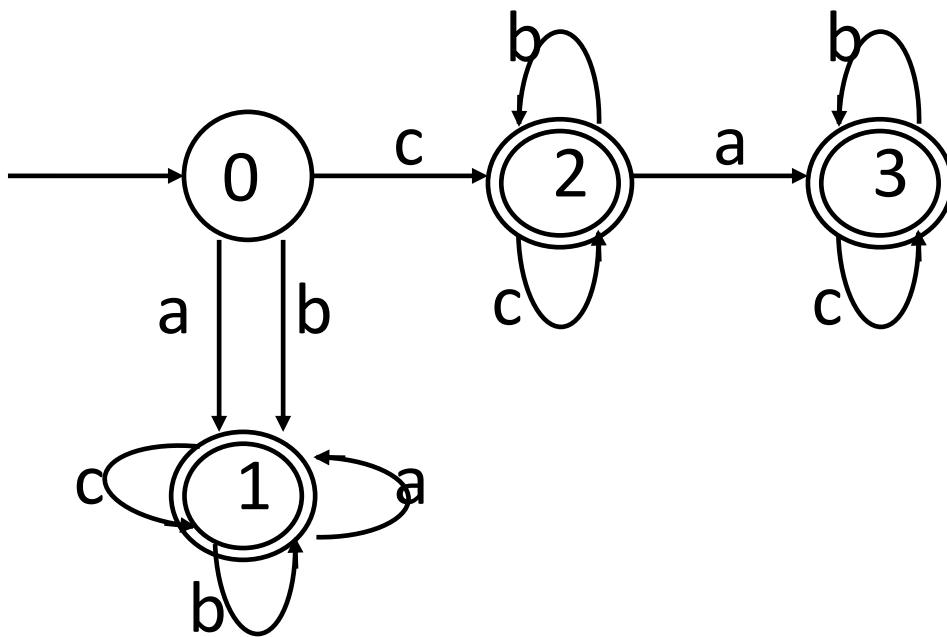
DFA

- A DFA accepts an input string x if and only if there is some path in the transition graph from start state to some accepting state

Example

- Recognition of Tokens
- Construct a DFA M , which can accept the strings which begin with a or b , or begin with c and contain at most one a .

Example



c bbcc
cccba
cccaab x

Non-deterministic Finite automata

- Same as deterministic, gives some flexibility.
- Five tuple representation
 $(Q, \Sigma, \delta, q_0, F)$, q_0 belongs to Q and F is a subset of Q
 δ is a mapping from $Q \times \Sigma$ to 2^Q
- More time for string matching as multiple paths exist.

Non-Deterministic Finite automata with ϵ

- Same as NFA. Still more flexible in allowing to change state without consuming any input symbol.
- δ is a mapping from $Q \times \Sigma \cup \{\epsilon\}$ to 2^Q
- Slower than NFA for string matching

NFA Some Observations

- In a NFA, the same character can label two or more transitions out of one state;
- In a NFA, ϵ is a legal input symbol.
- A DFA is a special case of a NFA

NFA Some Observations

- A NFA accepts an input string ‘x’ if and only if there is some path in the transition graph from start state to some accepting state. A path can be represented by a sequence of state transitions called moves.
- The language defined by a NFA is the set of input strings it accepts

RE to DFA

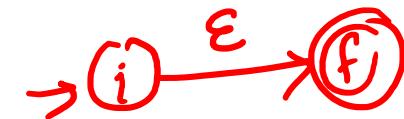
- Regular Expression could be converted to E-NFA using Thompson Construction Algorithm
- E-NFA could be converted to DFA using Subset construction algorithm

Basic Regular Expression and its NFA

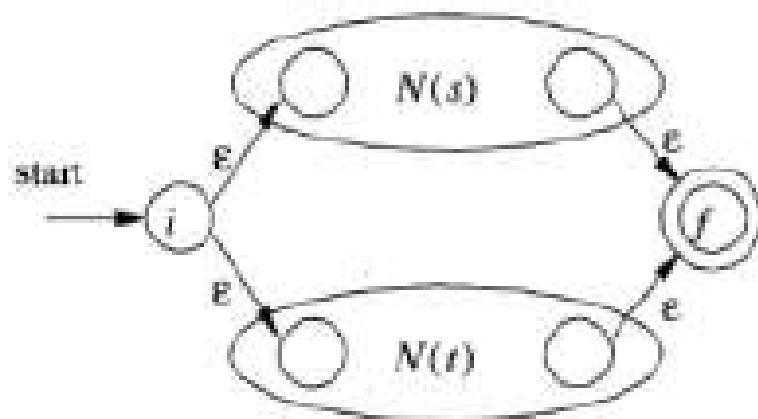


$r = a$

ϵ



Regular expression – Union operator and its corresponding NFA



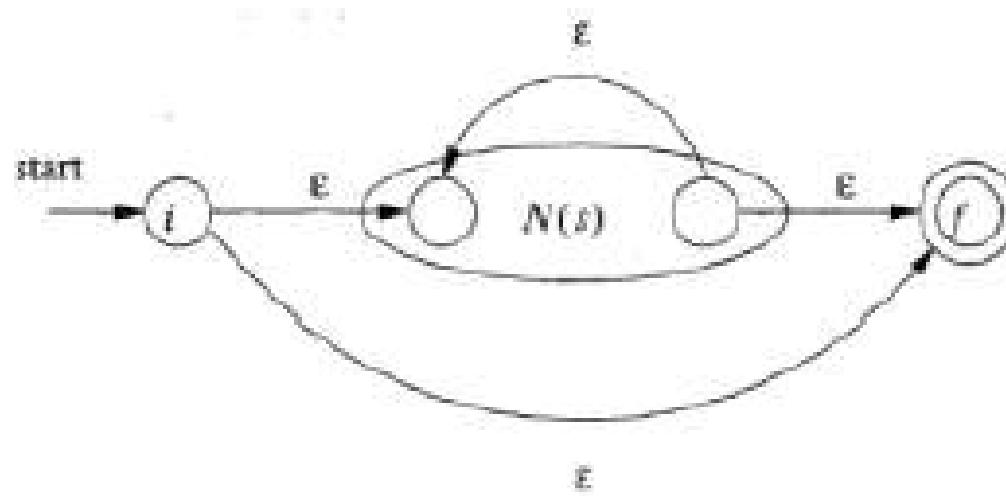
$$r = s | t$$

Regular expression with concatenation operator and its corresponding NFA



$$r = s t$$

Regular expression involving kleene closure operator and its NFA

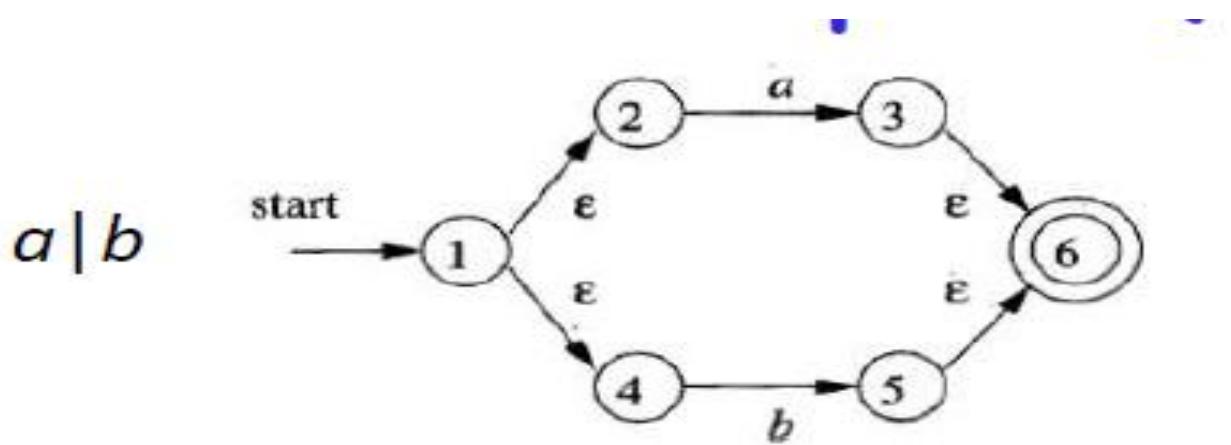


$$r = s^*$$

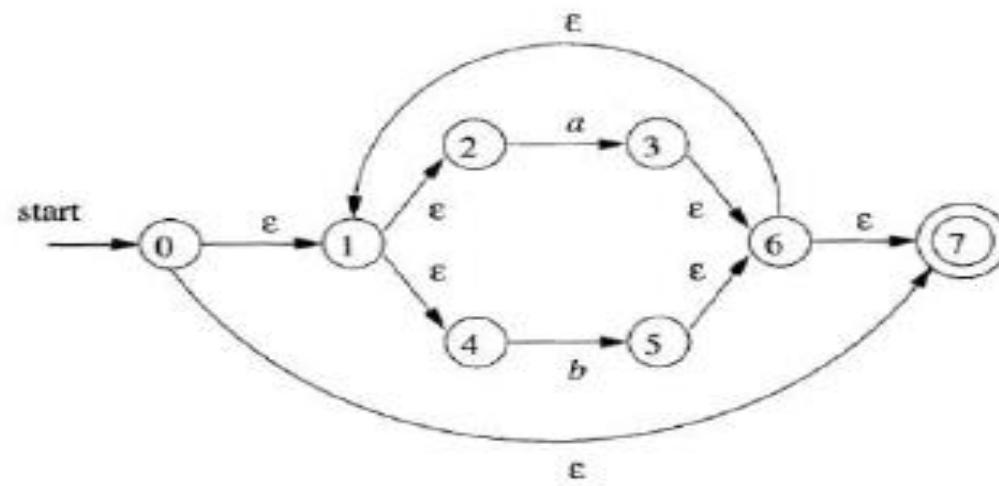
Algorithm

- Construct the basic NFA for each of the input symbols
- Prioritize the operators $()$, $*$, \cdot , $|$
- Use the discussed variations and form an NFA.

Example : $(a|b)^*abb$

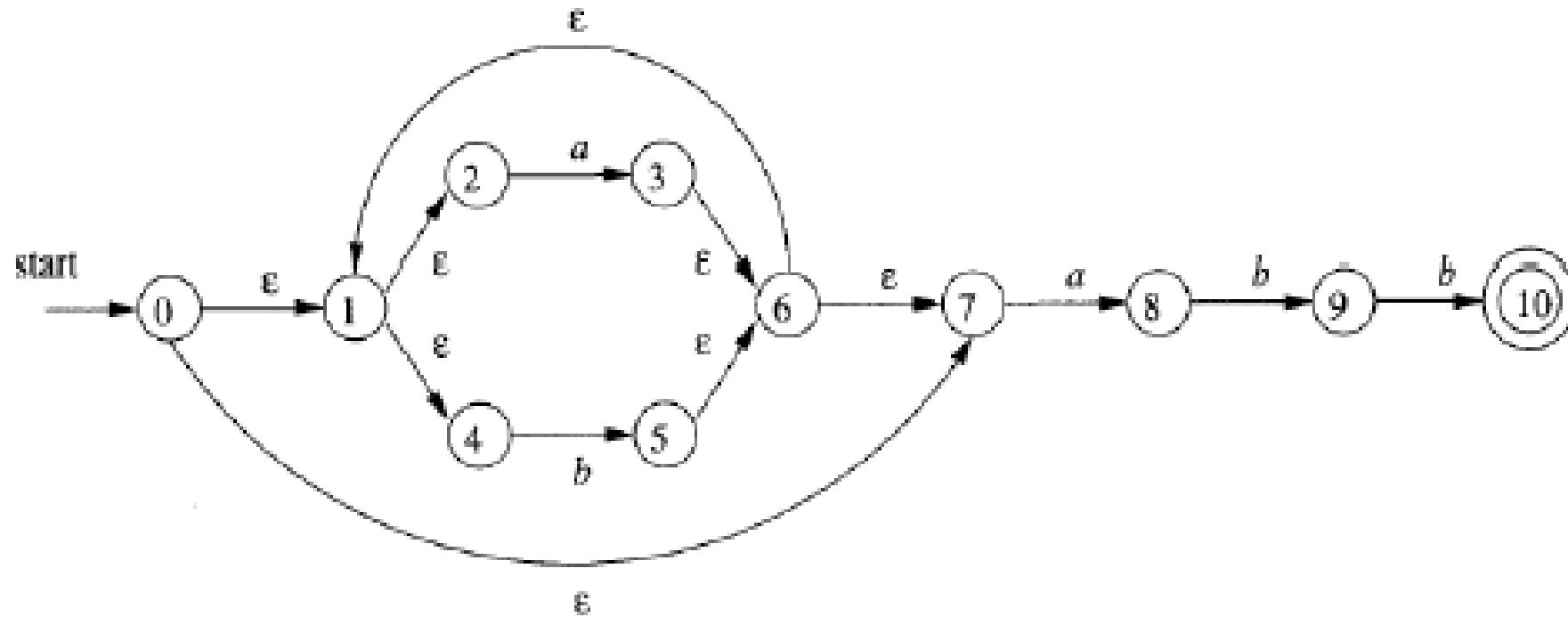


$(a|b)^*$



Example

$(a(b)^*)^*abb$



Conversion from NFA to DFA

- Reasons to conversion
 - Avoiding ambiguity
- The algorithm idea

Subset construction: The state set of a state in a NFA is thought of as a following STATE of the state in the converted DFA

Subset Construction algorithm

- Input. An NFA $N=(S,\Sigma,move,S_0,Z)$
- Output. A DFA $D= (Q,\Sigma,\delta,I_0,F)$, accepting the same language
- Requires Pre-processing - Determination of E-Closure

Pre-process-- ε -closure(T)

- Obtain ε -closure(T) $T \subseteq S$
- ε -closure(T) definition
 - A set of NFA states reachable from NFA state s in T on ε -transitions alone

Conversion from NFA to DFA – The pre-process--- ϵ -closure(T)

- ϵ -closure(T) algorithm
 - push all states in T onto stack;
 - initialize ϵ -closure(T) to T ;
 - while stack is not empty do {
 - pop the top element of the stack into t ;
 - for each state u with an edge from t to u labeled ϵ do {
 - if u is not in ϵ -closure(T) {
 - add u to ϵ -closure(T)
 - push u into stack}}

Subset Construction Algorithm

- $I_0 = \varepsilon\text{-closure}(S_0), I_0 \in Q$
- For each $I_i, I_i \in Q$,
 let $I_t = \varepsilon\text{-closure}(\text{move}(I_i, a))$
 if $I_t \notin Q$, then put I_t into Q
- Repeat above step until there are no new states to put into Q
- Let $F = \{I \mid I \in Q, \text{ such that } I \cap Z > \emptyset\}$

Example

$$A \text{ or } \mathcal{G}_0 \rightarrow \mathcal{E}\text{-closure}(0) = \{0, \underline{1}, \underline{2}, \underline{4}, T\}$$

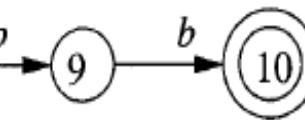
$$\mathcal{E}\text{-closure}\{3, 8\}$$

$$= \{3, 6, T, 1, 2, 4, 8\}$$

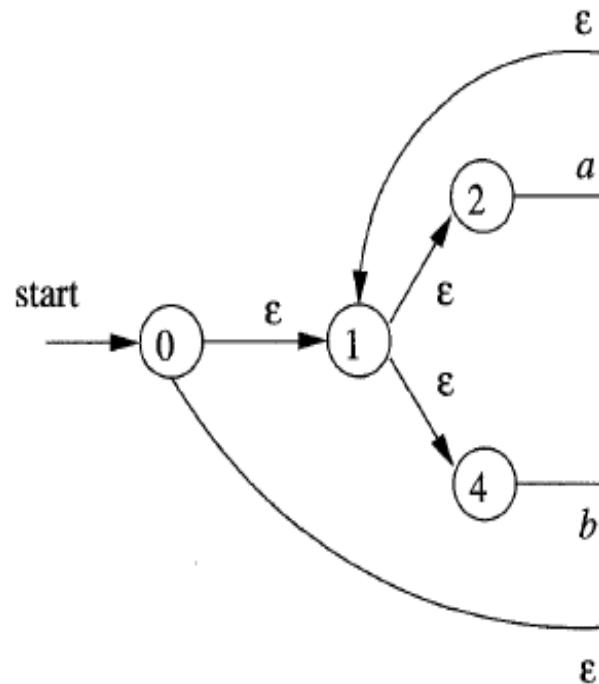
$$B = \{1, 2, 3, 4, 6, T, 8\}$$

$$\delta(B, b) = \mathcal{E}\text{-closure}(5)$$

$$= \{5, 6, T, 1, 2, 4\}$$



$$C = \{1, 2, 4, 5, 6, T\}$$

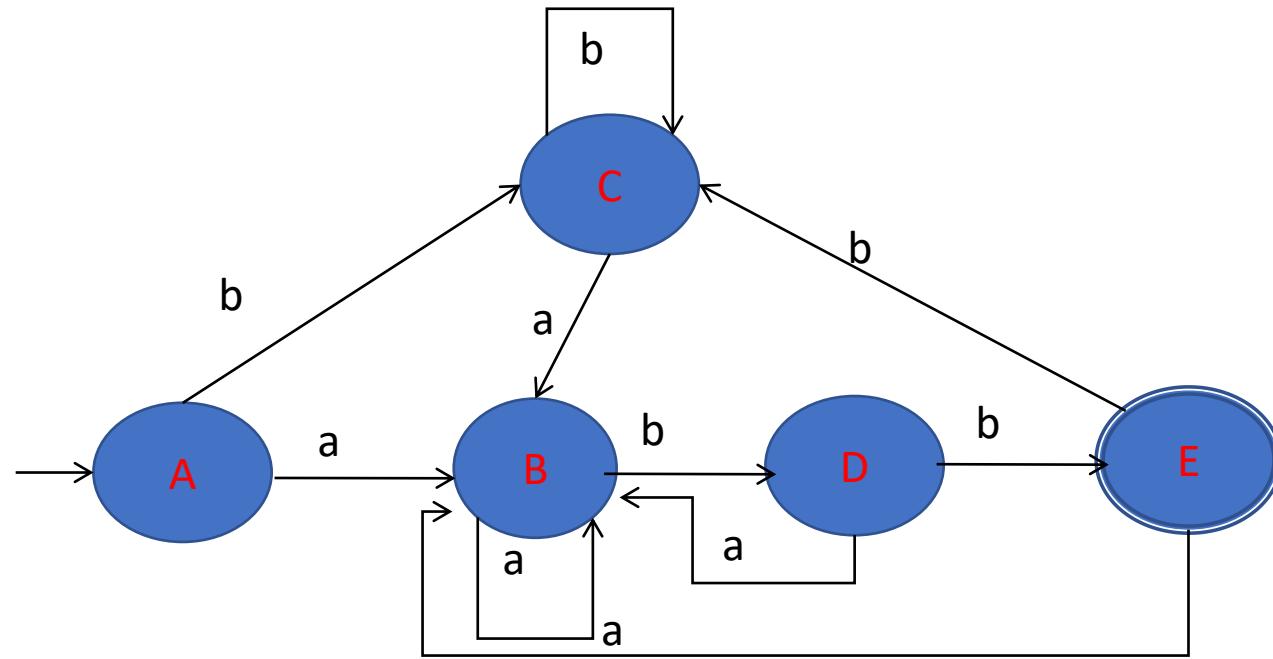


Result

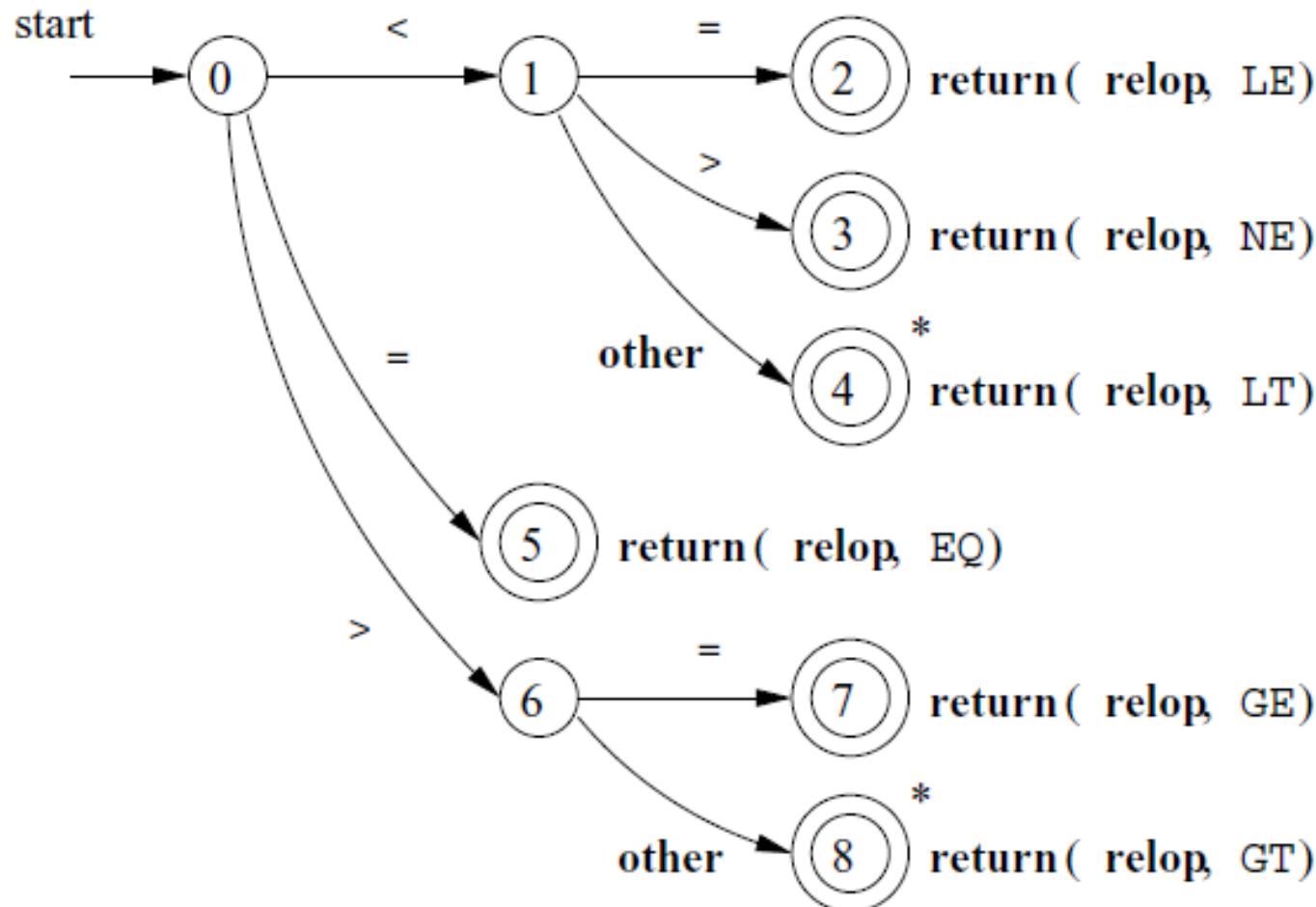
$$\begin{aligned}\delta(A, a) \\ = \text{\textcolor{red}{\(\varepsilon\)-closure}}(\text{move}(A, a)) \\ = B\end{aligned}$$

I	a	b
A={0,1,2,4,7}	B={1,2, 3, 4, 6, 7, 8}	C = {1,2,4,5,6,7}
B={1,2, 3, 4, 6, 7, 8}	B={1,2, 3, 4, 6, 7, 8}	D = {1,2,4,5,6,7,9}
C = {1,2,4,5,6,7}	B={1,2, 3, 4, 6, 7, 8}	C = {1,2,4,5,6,7}
D = {1,2,4,5,6,7,9}	B={1,2, 3, 4, 6, 7, 8}	E = {1,2,3,5,6,7,10}
E = {1,2,3,5,6,7,10}	B={1,2, 3, 4, 6, 7, 8}	C = {1,2,4,5,6,7}

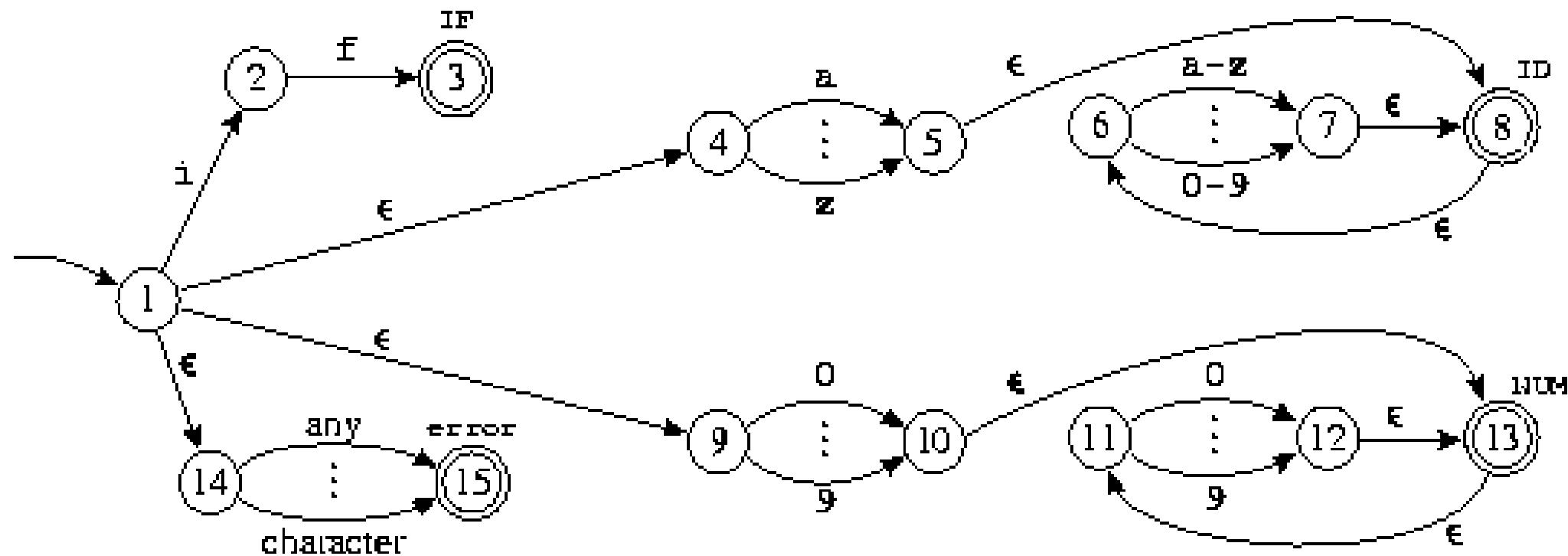
$$\delta(A, b)$$



Example



Example



Subset Construction Algorithm

- RE to E-NFA and then to DFA is time consuming and results in redundant states in the DFA
- Need to minimize the DFA for faster string matching

Summary till now

- DFA,NFA and NFA with ϵ as ways of defining patterns.
- DFA is faster, but construction is difficult
- NFA construction is easier but slower during string matching
- Conversion of RE to E-NFA
- Convert NFA to DFA

NFA and DFA

- Constructing NFA is easier. But string matching with DFA is faster.
- RE to DFA – done by converting to E-NFA and then to DFA
- This results in an increased number of states in the DFA – need for minimization

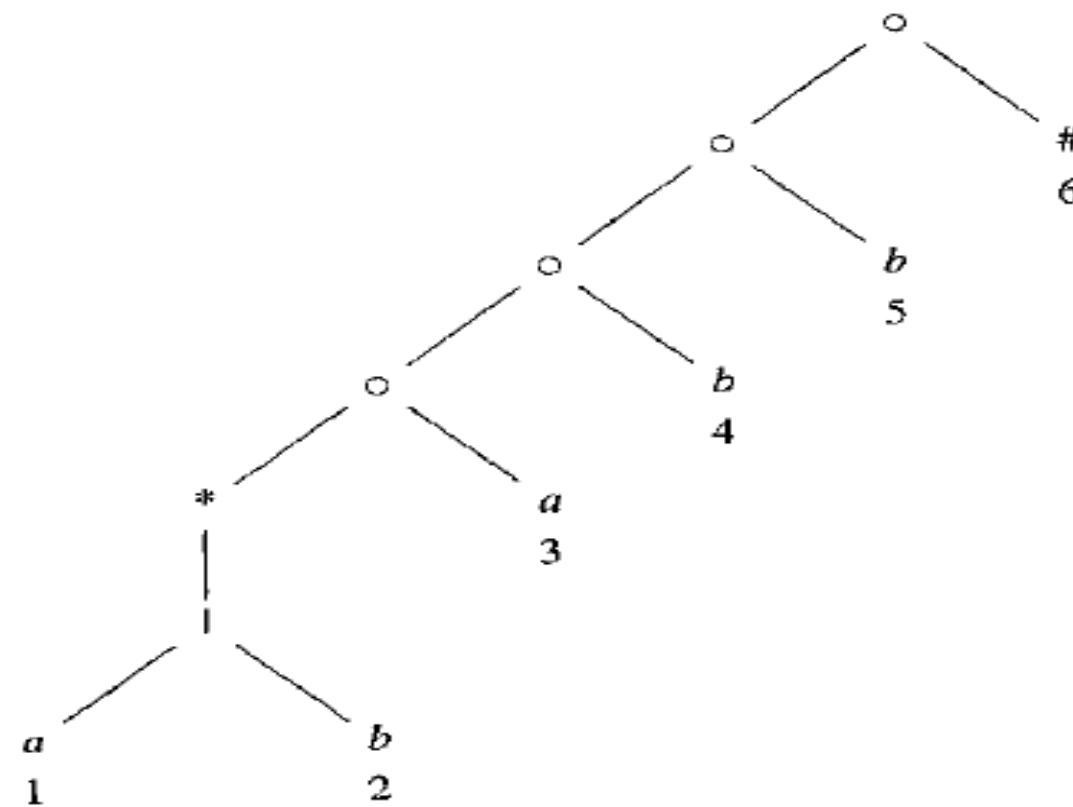
Minimized DFA

- Construct the DFA directly from RE by using a new algorithm
- Table filling minimization algorithm
 - Construct DFA and then use a procedure to eliminate redundant state

From Regular Expression to DFA Directly (Algorithm)

- Augment the regular expression r with a special end symbol $\#$ to make accepting states important: the new expression is $r \#$
- Construct a syntax tree for $r\#$
- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*

Example Syntax tree for $(a \mid b)^* abb$



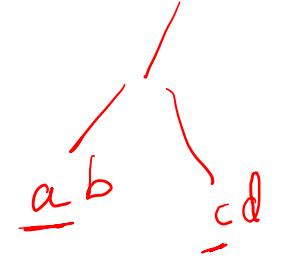
From Regular Expression to DFA Directly: Annotating the Tree

- $\text{nullable}(n)$: the subtree at node n generates languages including the empty string
- $\text{firstpos}(n)$: set of positions that can match the first symbol of a string generated by the subtree at node n

Algorithm

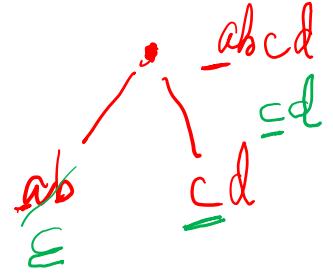
- $\text{lastpos}(n)$: the set of positions that can match the last symbol of a string generated by the subtree at node n
- $\text{followpos}(i)$: the set of positions that can follow position i in the tree

Annotating tree

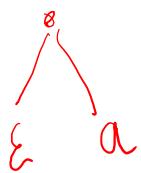


Node n	$\text{nullable}(n)$	$\text{firstpos}(n)$	$\text{lastpos}(n)$
Leaf ϵ	true	\emptyset	\emptyset
Leaf i	false	$\{i\}$	$\{i\}$
$c_1 \quad \quad c_2$ ε ↗ ↘ ↙ ↖ ↗ ↘ ↙ ↖	$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$

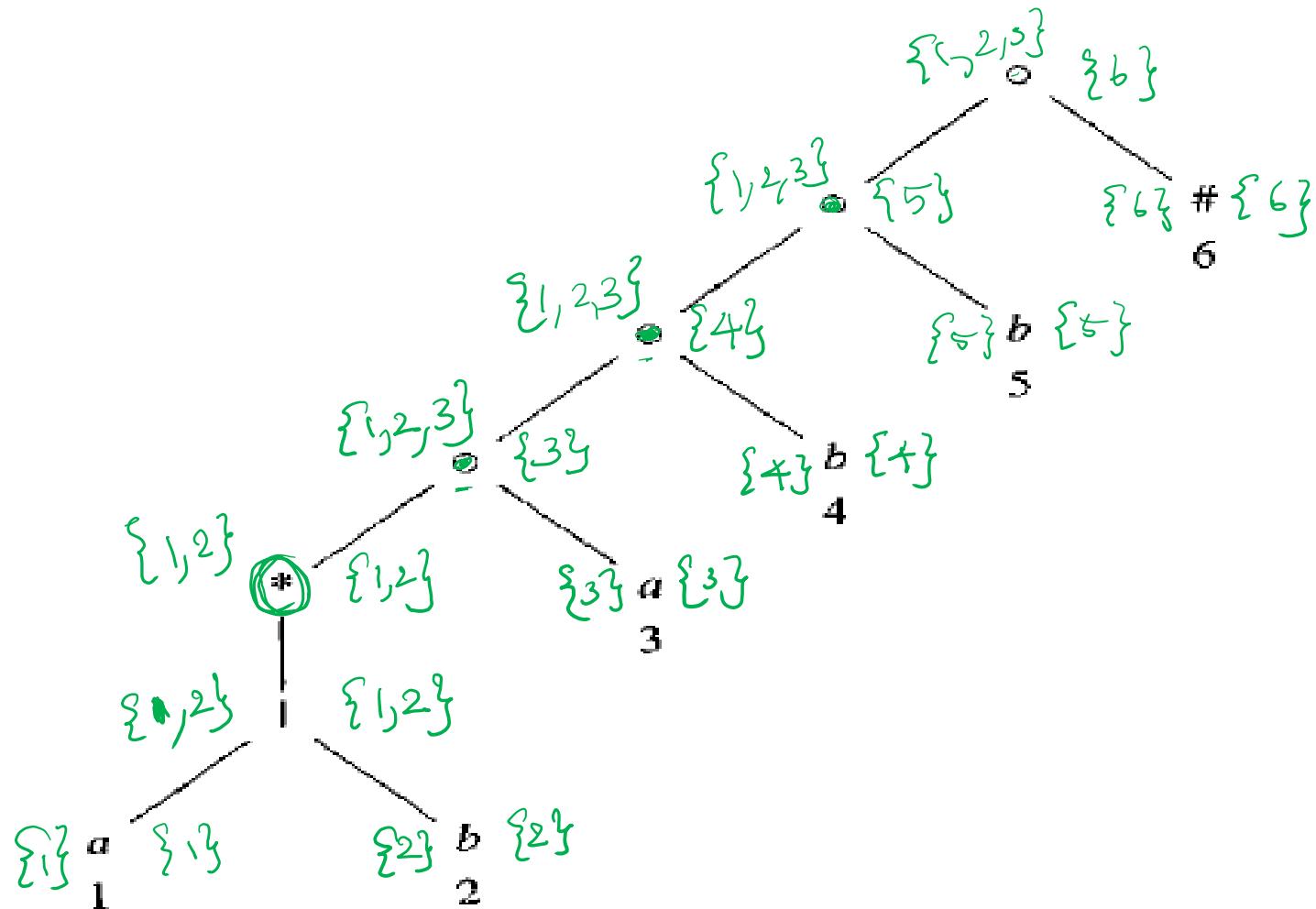
Annotating tree



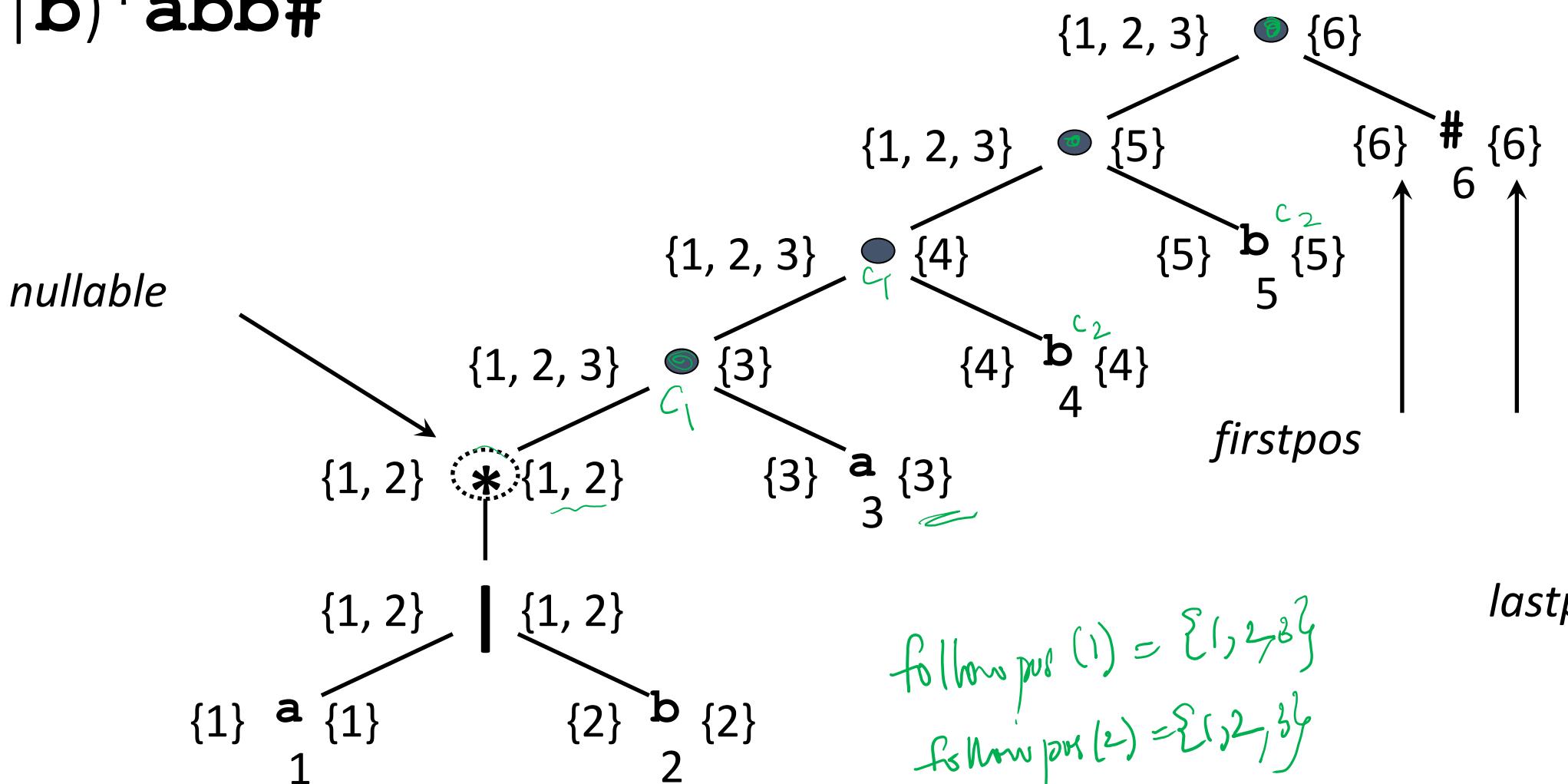
Node n	$\text{nullable}(n)$	$\text{firstpos}(n)$	$\text{lastpos}(n)$
\bullet $/ \backslash$ $c_1 \quad c_2$	$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$	if $\text{nullable}(c_1)$ then $\text{firstpos}(c_1) \cup$ $\text{firstpos}(c_2)$ else $\text{firstpos}(c_1)$	if $\text{nullable}(c_2)$ then $\text{lastpos}(c_1) \cup$ $\text{lastpos}(c_2)$ else $\text{lastpos}(c_2)$
$*$ $ $ c_1	true	$\text{firstpos}(c_1)$	$\text{lastpos}(c_1)$



Syntax tree for $(a|b)^* abb$



$(a|b)^*abb\#$



$$\text{followpos}(1) = \{1, 2, 3\}$$

$$\text{followpos}(2) = \{1, 2, 3\}$$

$$\text{followpos}(3) = \{4\}$$

followpos

```
for each node  $n$  in the tree do
    if  $n$  is a cat-node with left child  $c_1$  and right child  $c_2$  then
        for each  $i$  in  $\text{lastpos}(c_1)$  do
             $\text{followpos}(i) := \text{followpos}(i) \cup \text{firstpos}(c_2)$ 
        end do
    else if  $n$  is a star-node
        for each  $i$  in  $\text{lastpos}(n)$  do
             $\text{followpos}(i) := \text{followpos}(i) \cup \text{firstpos}(n)$ 
        end do
    end if
end do
```

Follow pos

Node	Followpos(n)
1	{1, 2, 3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	Φ

Algorithm

$s_0 := \text{firstpos}(\text{root})$ where root is the root of the syntax tree

$D\text{states} := \{s_0\}$ and is unmarked

while there is an unmarked state T in $D\text{states}$ **do**

 mark T

for each input symbol $a \in \Sigma$ **do**

 let U be the set of positions that are in $\text{followpos}(p)$

 for some position p in T ,

 such that the symbol at position p is a

if U is not empty and not in $D\text{states}$ **then**

 add U as an unmarked state to $D\text{states}$

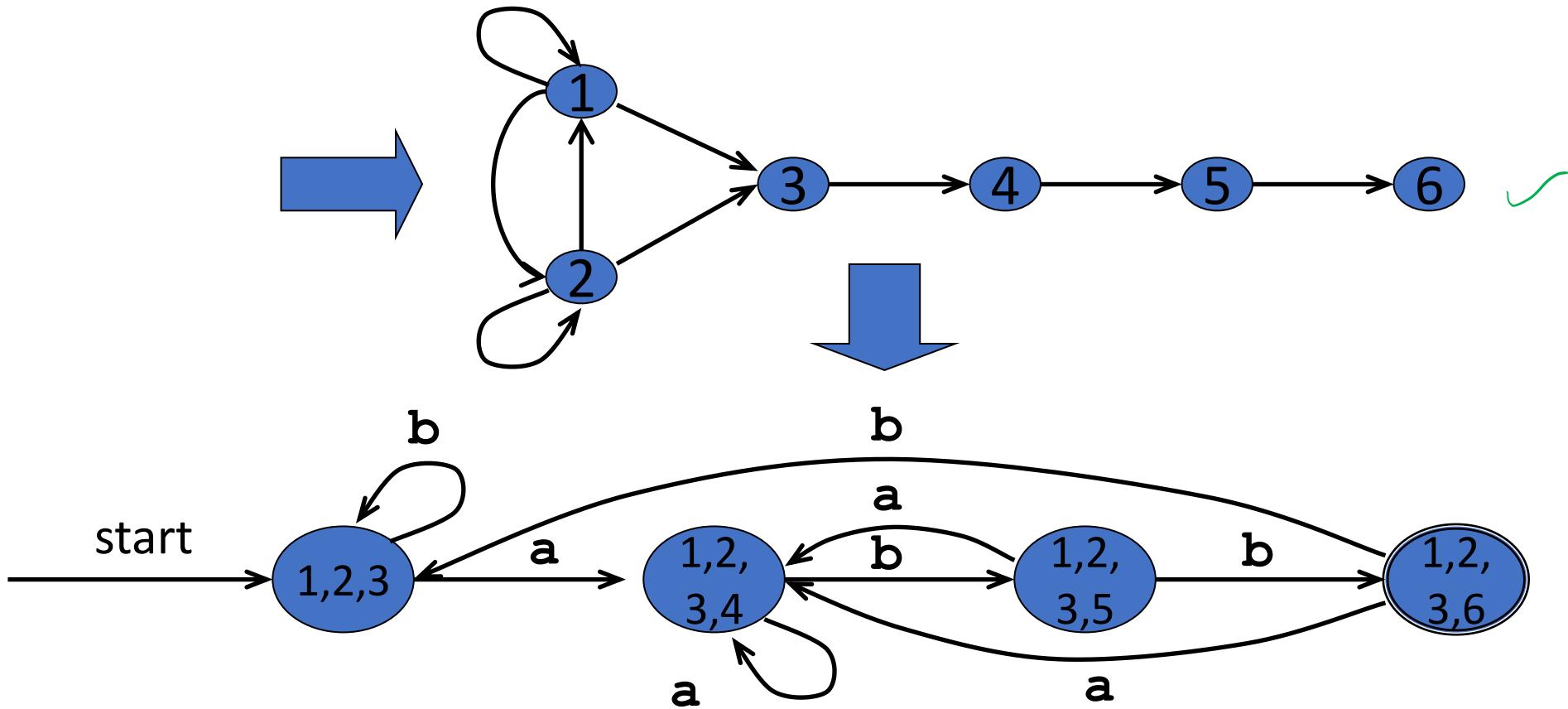
end if

$D\text{tran}[T,a] := U$

end do

end do

From Regular Expression to DFA Directly: Example



Minimized DFA - Table filling minimization algorithm

- Table filling minimization algorithm
 - Construct DFA and then use a procedure to eliminate redundant state
- Construct the DFA directly from RE by using a new algorithm

Basic Idea

- Find all groups of states that can be distinguished by some input string.
- At beginning of the process, we assume two distinguished groups of states:
 - the group of non-accepting states
 - the group of accepting states..
- Then we use the method of partition of equivalent class on input string to partition the existed groups into smaller groups

Minimization Algorithm

- Input: A DFA $M = \{S, \Sigma, \text{move}, s_0, F\}$
- Output: A DFA M' accepting the same language as M and having as few states as possible.

Minimization Algorithm

1. Construct an initial partition Π of the set of states with two groups: the accepting states F and the non-accepting states $S-F$. $\Pi_0 = \{I_0^1, I_0^2\}$
2. For each group I of Π_i , partition I into subgroups such that two states s and t of I are in the same subgroup if and only if for all input symbols a , states s and t have transitions on a to states in the same group of Π_i ; replace I in Π_{i+1} by the set of subgroups formed.
3. If $\Pi_{i+1} = \Pi_i$, let $\Pi_{final} = \Pi_{i+1}$ and continue with step (4). Otherwise, repeat step (2) with Π_{i+1}

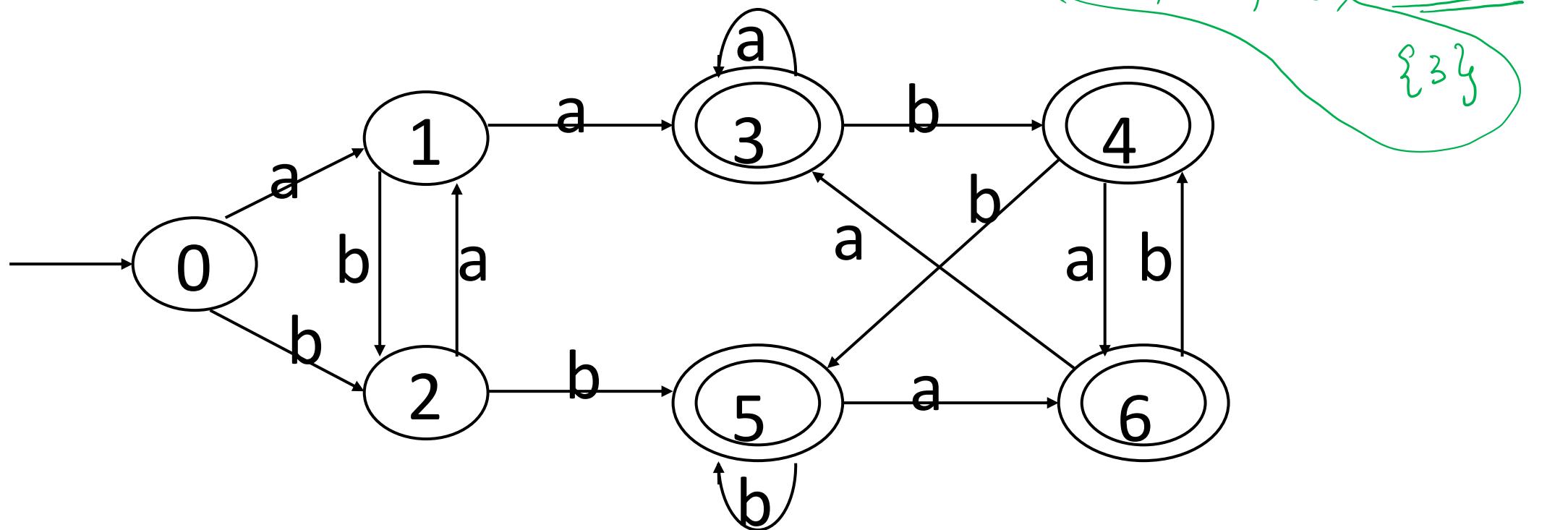
Minimization Algorithm

- Choose one state in each group of the partition Π_{final} as the representative for that group which will be the states of the reduced DFA M' .
- Let s and t be representative states for s 's and t 's group respectively, and suppose on input a there is a transition of M from s to t . Then M' has a transition from s to t on a .

Minimization Algorithm

- If M' has a dead state(a state that is not accepting and that has transitions to itself on all input symbols),then remove it. Also remove any states not reachable from the start state.

Example



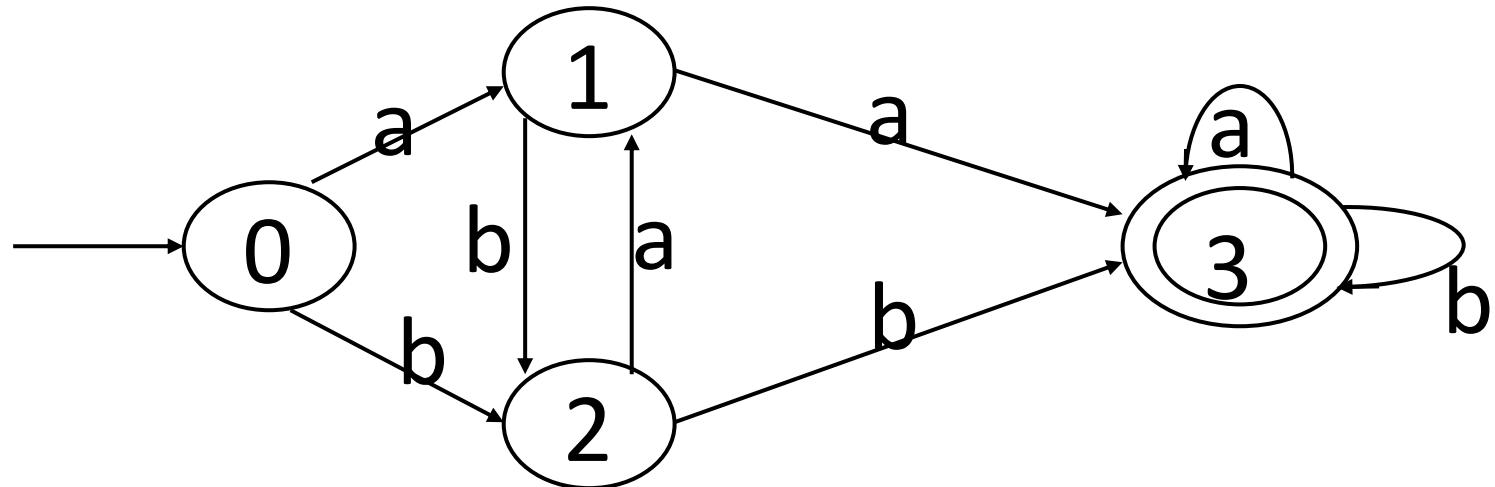
Example

- Initialization: $\Pi_0 = \{\{0,1,2\}, \{3,4,5,6\}\}$
- For Non-accepting states in Π_0 :
 - a: $\text{move}(\{0,2\}, a) = \{1\}$; $\text{move}(\{1\}, a) = \{3\}$. 1,3 do not in the same subgroup of Π_0 .
 - So, $\Pi_1` = \{\{1\}, \{0,2\}, \{3,4,5,6\}\}$
 - b: $\text{move}(\{0\}, b) = \{2\}$; $\text{move}(\{2\}, b) = \{5\}$. 2,5 do not in the same subgroup of $\Pi_1`$.
 - So, $\Pi_1`` = \{\{1\}, \{0\}, \{2\}, \{3,4,5,6\}\}$

Example

- For accepting states in Π_0 :
 - a: $\text{move}(\{3,4,5,6\}, a) = \{3,6\}$, which is the subset of $\{3,4,5,6\}$ in Π_1 “
 - b: $\text{move}(\{3,4,5,6\}, b) = \{4,5\}$, which is the subset of $\{3,4,5,6\}$ in Π_1 “
 - So, $\Pi_1 = \{\{1\}, \{0\}, \{2\}, \{3,4,5,6\}\}$.
- Apply the same step again to Π_1 ,and get Π_2 .
 - $\Pi_2 = \{\{1\}, \{0\}, \{2\}, \{3,4,5,6\}\} = \Pi_1$,
 - So, $\Pi_{\text{final}} = \Pi_1$
- Let state 3 represent the state group $\{3,4,5,6\}$

Minimized DFA



Compiler construction tools

Lexical Analyzer

- scanner generators
- input: source program
- output: lexical analyzer
- task of reading characters from source program and recognizing tokens or basic syntactic components
- maintains a list of reserved words

Lexical Analyzer

- Flex (fast lexical analyzer generator) or LEX – Rule Based programming language
- Example - specifies a scanner which replaces the string “username” with the user’s login name

%%

```
username printf("%s", getlogin());
```

Syntax Analyzer

- parser generators
- input: context-free grammar
- output: syntax analyzer
- the task of the syntax analyzer is to produce a representation of the source program in a form directly representing its syntax structure.

Syntax Analyzer

- Bison (Yacc-compatible parser gen.)
- a general purpose parser generator that converts grammar description for an LALR(1) CFG into a C program

Syntax Analyzer

- Bison grammar example (reverse polish notation)

```
%{  
#define YYSTYPE double  
#include <math.h>  
%}  
%token NUM  
%% /* grammar rules and actions below */  
  
%% C program
```

Semantic Analyzer

- syntax-directed translators
- input: parse tree
- output: routines to generate Intermediate code
- “The role of the semantic analyzer is to derive methods by which the structures constructed by the syntax analyzer may be evaluated or executed.”

- type checker
- two common tactics:
 - ~ flatten the semantic analyzer's parse tree
 - ~ embed semantic analyzer with syntax analyzer (syntax-driven translation)

Intermediate Code Generator

- Automatic code generators
- input: Intermediate code rules
- output: crude target machine program
- “The task of the code generator is to traverse this tree, producing functionally equivalent object code.”
- three address code is one type

Code Optimizer

- Data flow engines
- input: I-code
- output: transformed code
- there is rarely a guarantee that the resulting code is the best possible.

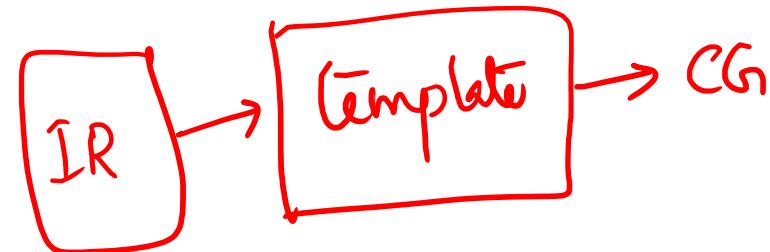
Code Generator

- Automatic code generators
- input: optimized (transformed) I-code
- output: target machine program
- Example $(8 * x) / 2$

Load a, x

Mult a, 8

Div a, 2



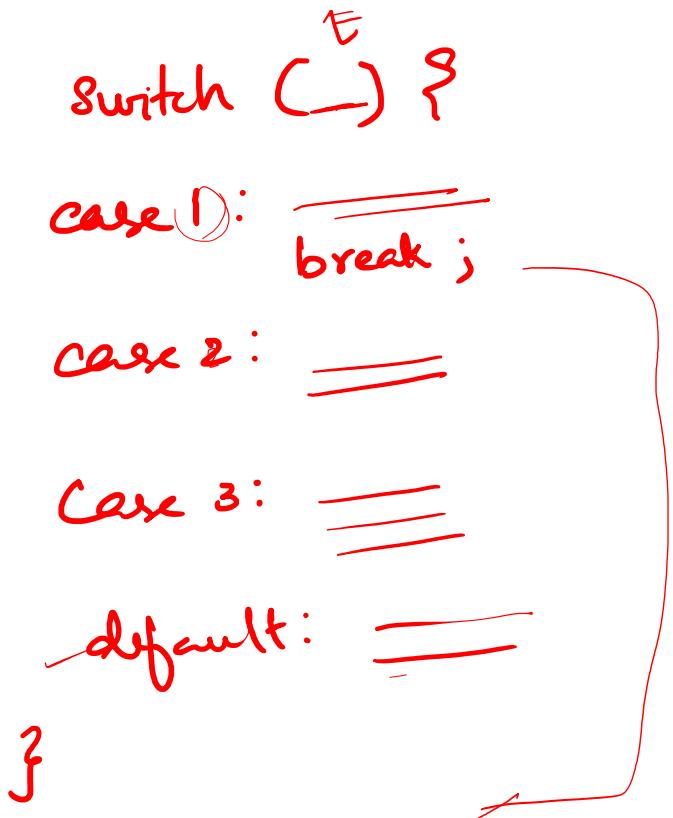
Challenges in Compiler Design

- Language Semantics
- Hardware Platform
- OS and system software
- Error Handling
- Aid in debugging
- Optimization
- Runtime Environment
- Speed of compilation

Language Semantics

- functionality of the programming language has to be supported
- Example
 - Case statements
 - Loop index
 - Break statements

switch (^E) {
 case 1: ====
 case 2: ====
 case 3: ====
 default: ====
}



Hardware Platform

- Hardware platform vary from one machine to another machine that the architecture itself changes
- Code generation strategy for accumulator based machine cannot be similar to a stack based machine
- CISC or RISC instructions set

OS and system software

- Format of file to be executed is depicted by the operating system
- Linking process or the linker tool will combine many object files generated by different compilers into some executable file

Error Handling



- Show appropriate error messages
- Compiler designer has to imagine the probable types of mistakes, and design suitable detection, and recovery mechanism
- Some compilers even go to the extent of modifying source program partially, in order to correct it

10: a =
21: a =
22: a =

10: a = b + c;
11 : d = a * 2;
22 : c = d * a;

Aid in debugging

- Helps in detecting logical errors in the program
- user needs to control the execution of machine language program, but sitting at the source language level
- compiler has to generate extra information regarding the correspondence between source and machine instructions
- Symbol table also needs to be available for to the debugger

Optimization

- Have to identify the set of transformation that may be beneficial for most of the programs in a language
- transformation should be safe
- trade-off between the time spent to optimize a program vs improvement in the execution time
- several levels of optimizations are used
- selecting a debugging mode or debugging option may disable any optimizations that disturbs the correspondence between the source program and object code

Runtime Environment

- deals with creating space for parameters and local variables
- Static memory locations may be used
- Stack frames are used to support recursion

Speed of compilation

- initial phase of program development contains lots of bugs, hence quick compilation may be the objective rather than optimized code
- towards the final stages, execution efficiency becomes the prime concern, more compilation time may be afforded to optimize the machine code

Parser

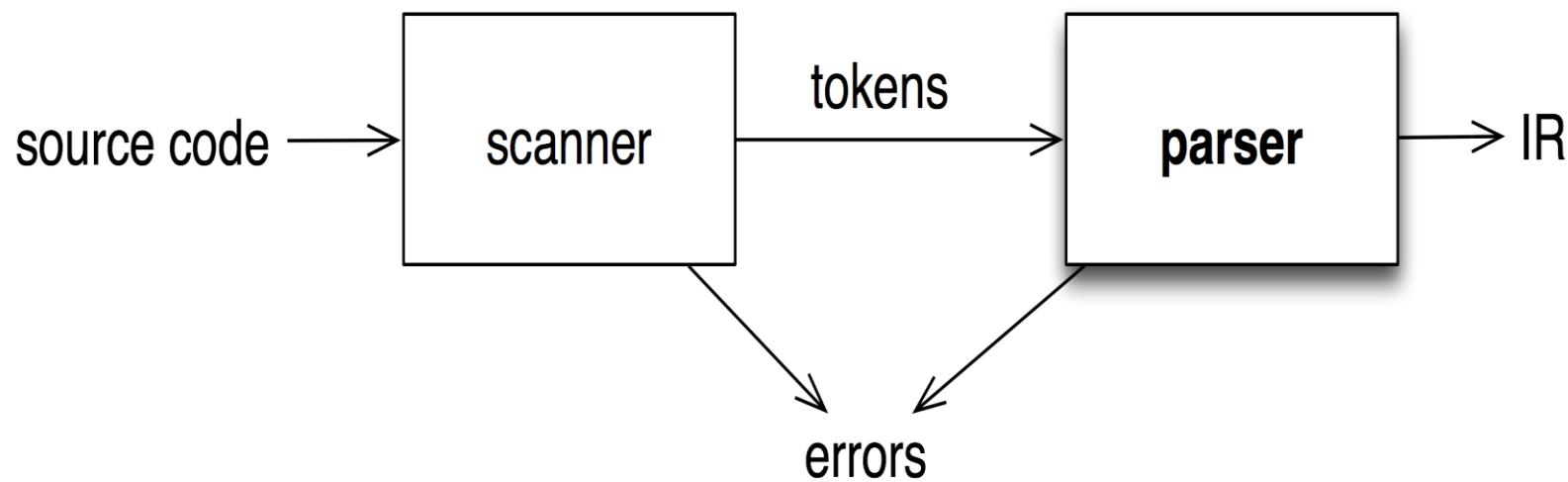
Second Phase of the compiler

- Parser – Typically integrated with the lexical phase of the compiler
- Top Down Parser
- Bottom Up Parser

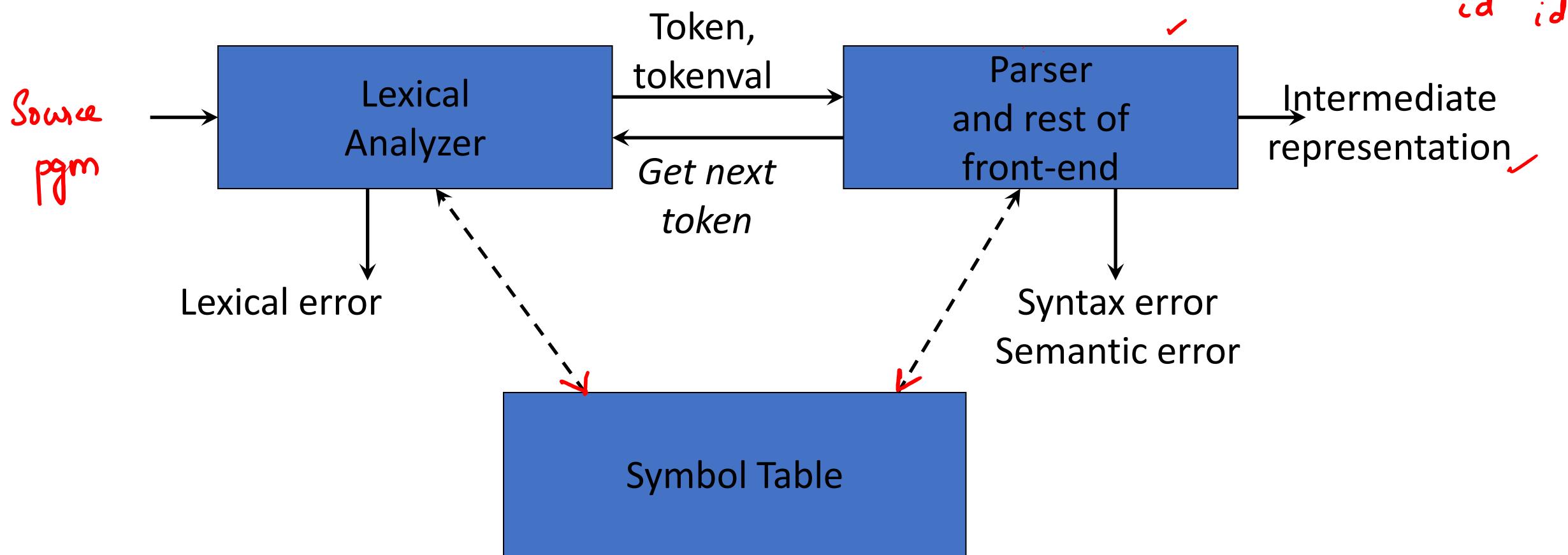
Functions of the Parser

- Validate the syntax of the programming language
- Points out errors in the statements

Role of the Parser



Role of the Parser



General Types of Parsers

- Universal Parsers
 - Cocke- Younger-Kasami
 - Earley's Algorithm
- Top-Down Parsers
- Bottom Up Parsers

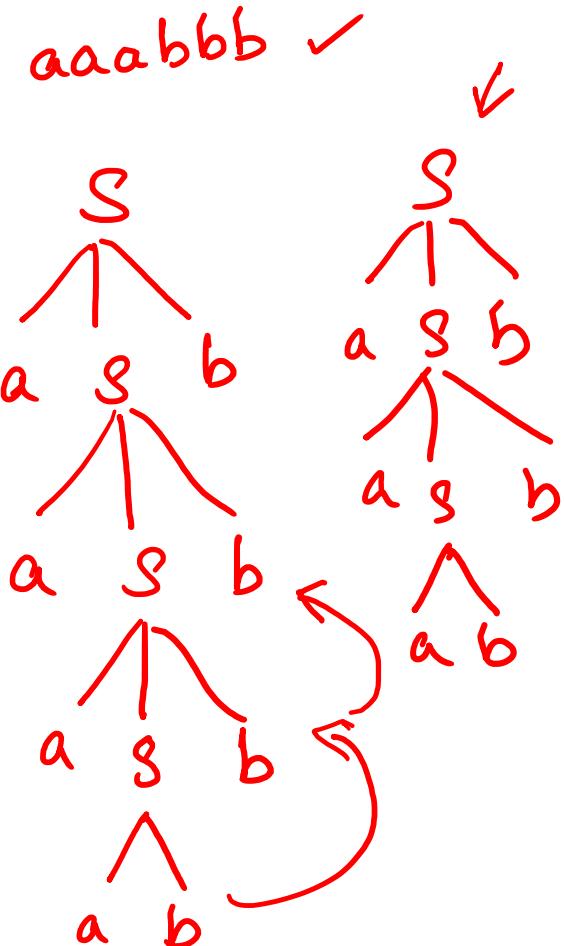
Universal Parsers

- Can parse any Grammar
- Use in NLP
- But too inefficient in Compilers

Top Down Parsers

- Build the parse trees from the top to the bottom
- Recursive Descent parsers – requires backtracking
- LL Parsers – No Backtracking

$S \rightarrow aSb \mid ab$



Example

Consider the Grammar

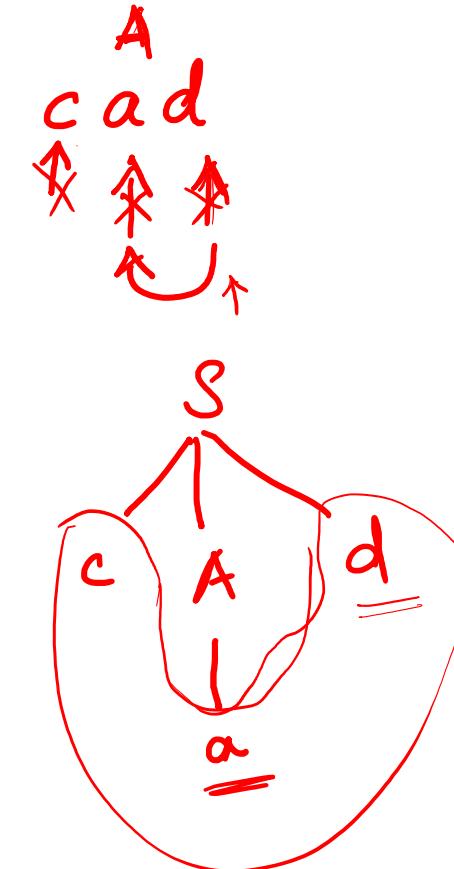
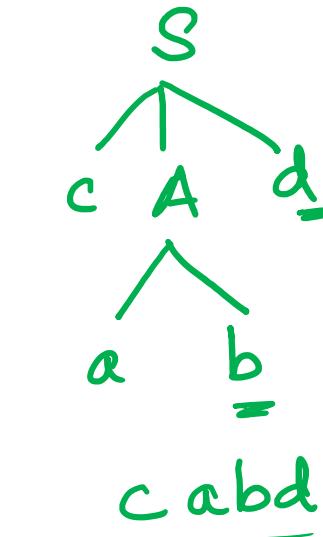
$$S \rightarrow c \underline{A} d$$

$$A \rightarrow ab \mid a \underline{}$$

Let the input be "cad"

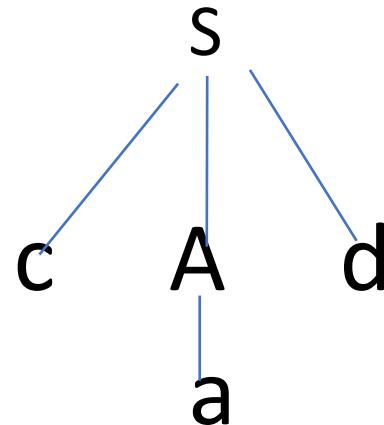
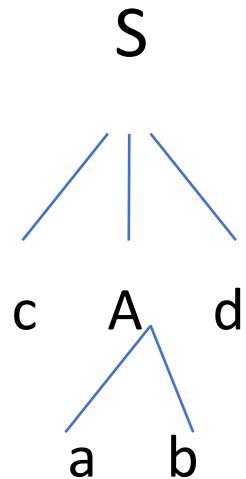
$$\begin{aligned} S &\rightarrow c \overset{①}{A} d \mid d C \\ A &\rightarrow \overset{②}{ab} \mid C \\ C &\rightarrow c \mid \epsilon \end{aligned}$$

cabd



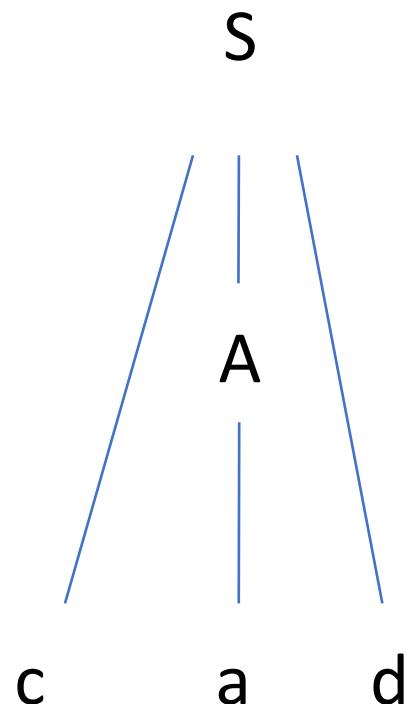
Parsing (Recursive Descent)

Expand A using **the first alternative** $A \rightarrow ab$



Bottom up Parsers

- Start from the bottom and work up to the root for parsing a string
- LR parsers are bottom up parsers



Parsing

- Both Top Down and Bottom up parsers parse the string based on a viable-prefix property
- This property states that before the string is fully processed, if there is an error, the parser will identify it and recovers

Context Free Grammars - CFG

- Programming language constructs are defined using context free grammar
- For example

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

Expression grammar involving the operators, +, *, ()

*id + id * id*

Context Free Grammars

- Defined formally as (V, T, P, S)

V – Variables / Non-terminals

T – Terminals that constitute the string

P – Set of Productions that has a LHS and RHS

S – Special Symbol, subset of V

$$NT \rightarrow \alpha$$

$$A \rightarrow A\alpha$$

$$A \rightarrow B\beta$$

$$A \rightarrow A\alpha \mid B\beta$$

Context Free Grammar

- Example

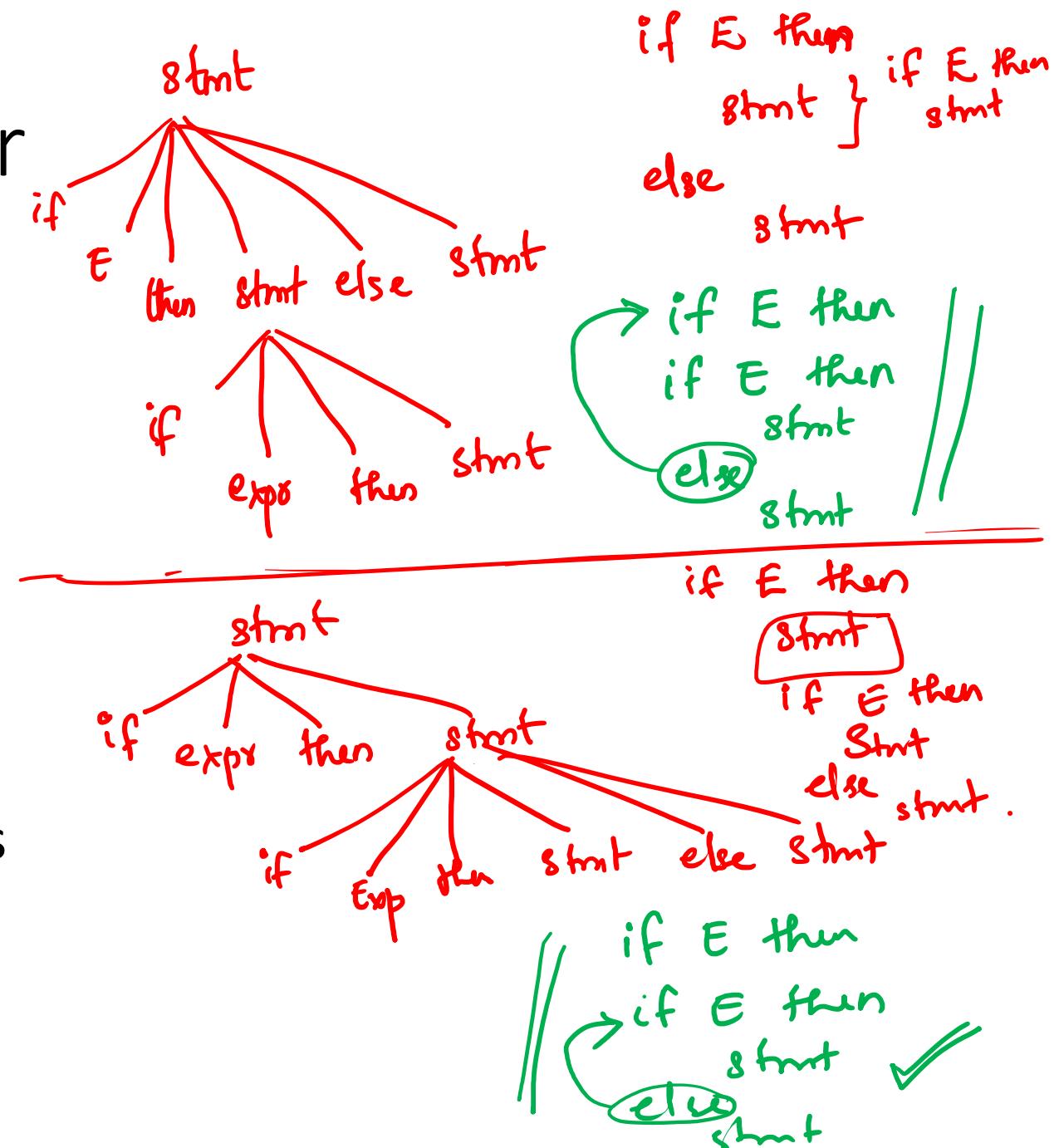
$\text{stmt} \rightarrow \text{if } E \text{ then } \underline{\text{stmt}} \text{ else } \underline{\text{stmt}}$

$\text{stmt} \rightarrow \text{if } E \text{ then } \text{stmt}$

$\text{stmt} \rightarrow a$

$E \rightarrow b$

Here, stmt, E are Non-terminals,
if, then, else, a, b are all terminals



Grammar - notations

- Terminals - $a, b, c, \dots \in T$
 - specific terminals: **0**, **1**, **id**, **+**
- Non-terminals - $A, B, C, \dots \in N$
 - specific non-terminals: *expr*, *term*, *stmt*
- Grammar symbols - $X, Y, Z \in (N \cup T)$
- Strings of terminals
 - $u, v, w, x, y, z \in T^*$
- Strings of grammar symbols
 - $\alpha, \beta, \gamma \in (N \cup T)^*$

Derivation

- The *one-step derivation* is defined by

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

where $A \rightarrow \gamma$ is a production in the grammar

- In addition, we define

- \Rightarrow is *leftmost* \Rightarrow_{lm} if α does not contain a nonterminal
- \Rightarrow is *rightmost* \Rightarrow_{rm} if β does not contain a nonterminal
- Transitive closure \Rightarrow^* (zero or more steps)
- Positive closure \Rightarrow^+ (one or more steps)

Derivation

- The *language generated by G* is defined by
$$L(G) = \{w \mid S \Rightarrow^+ w\}$$

Derivation

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow - E$$

$$E \rightarrow \text{id}$$

$$E \Rightarrow - E \Rightarrow - \text{id}$$

$$E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + \text{id} \Rightarrow_{rm} \text{id} + \text{id}$$

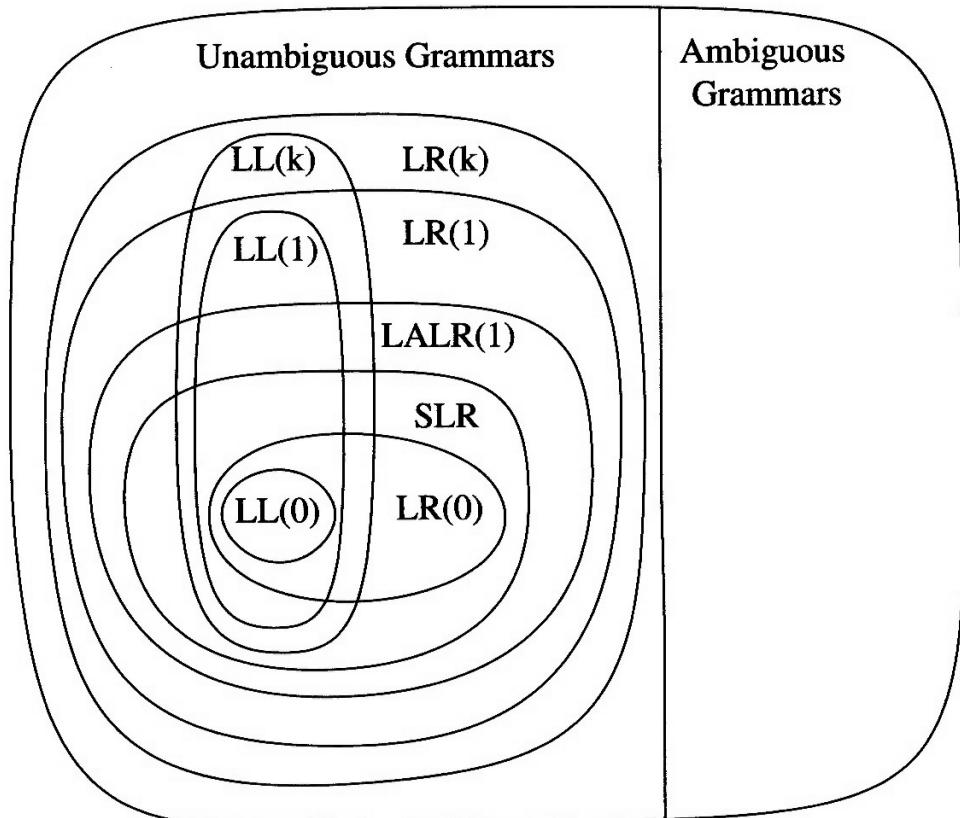
$$E \Rightarrow E^* E \quad E \Rightarrow^+ \underline{\text{id}} * \text{id} + \text{id}$$

$$\begin{aligned} E &\Rightarrow \underset{lm}{\underline{-}} E + E \Rightarrow \text{id} + E \\ &\Rightarrow \text{id} + \text{id} - \end{aligned}$$

Parsers

- Context Free grammars are already defined for all programming constructs
- All strings that are part of the programming language will be based on this construct
- Hence, parsers are designed keeping in mind the CFG

Hierarchy of Grammar Classes



Hierarchy

- **LL(k):**
 - Left-to-right, Leftmost derivation, k tokens lookahead
- **LR(k):**
 - Left-to-right, Rightmost derivation, k tokens lookahead
- **SLR:**
 - Simple LR (uses “follow sets”)
- **LALR:**
 - LookAhead LR (uses “lookahead sets”)

Top Down Parsers

- LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:

$$E \rightarrow T + T$$

$$T \rightarrow (E)$$

$$T \rightarrow - E$$

$$T \rightarrow \mathbf{id}$$

Leftmost derivation:

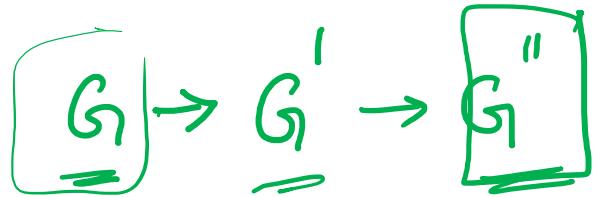
$$E \Rightarrow_{lm} T + T$$

$$\Rightarrow_{lm} \mathbf{id} + T$$

$$\Rightarrow_{lm} \mathbf{id} + \mathbf{id}$$

Top Down Parsers – LL (1) Parsers

- LL parsers cannot handle
 - Left Recursive Grammar
 - Left Factoring

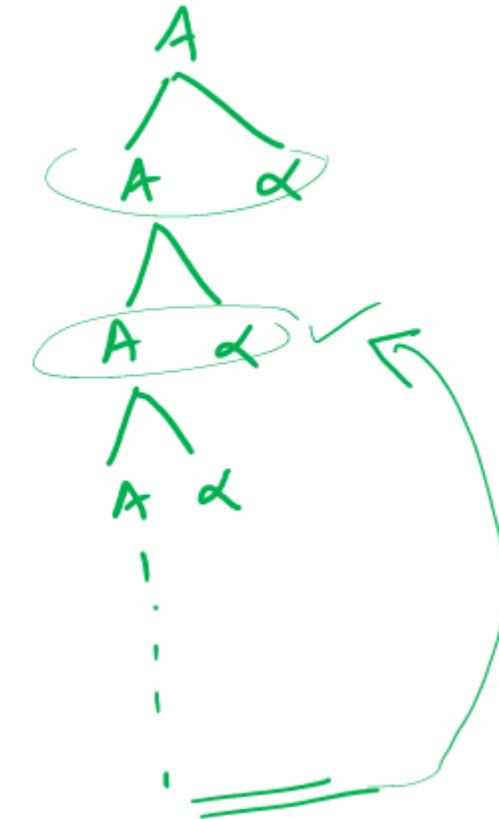


Left Recursive Grammar

- Formally, a grammar is *left recursive* if $\exists A \in NT$ such that \exists a derivation $A \Rightarrow^+ A\alpha$, for some string $\alpha \in (NT \cup T)^+$
- $A \rightarrow A\alpha / \beta / \gamma$

$$\begin{array}{c} \boxed{A \rightarrow A\alpha} \\ A \rightarrow \beta \\ A \rightarrow \gamma \end{array}$$

$$\begin{aligned} A &\Rightarrow A\alpha \\ &\Rightarrow A\alpha\alpha \\ &\Rightarrow A\alpha\alpha\alpha \end{aligned}$$



Left Factor

- When a non-terminal has two or more productions whose right-hand sides start with the same grammar symbols the grammar is said to have left-factor property
- Example $A \rightarrow \underline{\alpha} \beta_1 / \underline{\alpha} \beta_2 / \dots / \alpha \beta_n / \gamma$

$$\begin{aligned} A &\rightarrow \underline{a}B \mid \underline{a}S \mid cb \\ A &\rightarrow \underline{ab}B \underline{C} \mid \underline{ab}B \underline{Sd} \mid cb \end{aligned}$$

Pre-requisites for Top-Down Parser

- Eliminate Left Recursion
- Left Factor the grammar

$$\begin{array}{l} A \xrightarrow{\alpha\beta} \\ A \xrightarrow{\alpha} | \textcircled{B} \beta \\ B \xrightarrow{\gamma} | \epsilon \end{array}$$

Eliminating Left Recursion

Arrange the non-terminals in some order A_1, A_2, \dots, A_n

for $i = 1, \dots, n$ **do**

for $j = 1, \dots, i-1$ **do**

 replace each

 with $A_i \rightarrow A_j \gamma$

 where $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$

$A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$

end

 eliminate the immediate left recursion in A_i

end

Eliminate Left Recursion

- Rewrite every left-recursive production

$$A \rightarrow A\alpha / \beta / \gamma / A\delta$$

- into a right-recursive production:

$$A \rightarrow \beta A_R / \gamma A_R$$

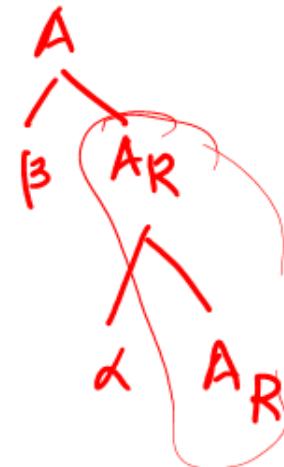
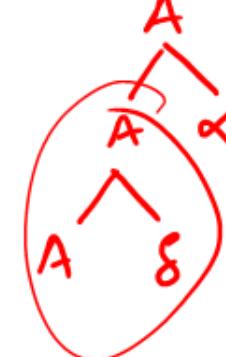
$$A_R \rightarrow \alpha A_R / \delta A_R / \epsilon$$

$$A \rightarrow A\alpha / A\delta / \beta / \gamma$$

$$A \rightarrow \beta A_R / \gamma A_R$$

$$A_R \rightarrow \alpha A_R / \delta A_R / \epsilon$$

$$A \rightarrow \beta / \gamma /$$



Example

- $A \rightarrow BC | a$
- $B \rightarrow CA | Ab$
- $C \rightarrow AB | CC | a$

$C \rightarrow \underline{AB} | CC | a$

$C \rightarrow \underline{BCB} | ab | CC | a$

$C \rightarrow \underline{\underline{CAB_R}} \underline{CB} | abB_R CB | aB | \underline{CC} | a$

① $C \rightarrow abB_R CBC_R | aBC_R | aC_R$

⑤ $C_R \rightarrow AB_R CBC_R | CC_R | \epsilon$

i j
1 1
2 1
3 1 → 2

A - 1 B - 2 C - 3

- ① $A \rightarrow BC | a$ ✓
 $B \rightarrow CA | A b$
 $B \rightarrow CA | \overset{A}{\cancel{BC}} \overset{b}{\cancel{b}} | ab$
- ② $B \rightarrow CAB_R | abB_R$ ✓
 $B_R \rightarrow cbB_R | \epsilon$
- ③ $B \rightarrow CAB_R | abB_R$ ✓
 $B_R \rightarrow cbB_R | \epsilon$

- $i = 1$: nothing to do

$i = 2, j = 1$: $B \rightarrow CA \mid \underline{A} \mathbf{b}$

$\Rightarrow B \rightarrow CA \mid \underline{B} C \mathbf{b} \mid \underline{\mathbf{a}} \mathbf{b}$

$\Rightarrow_{(\text{imm})} B \rightarrow CA B_R \mid \mathbf{a} \mathbf{b} B_R$

$B_R \rightarrow C \mathbf{b} B_R \mid \varepsilon$

$i = 3, j = 1$: $C \rightarrow \underline{A} B \mid CC \mid \mathbf{a}$

$\Rightarrow C \rightarrow \underline{B} C B \mid \underline{\mathbf{a}} B \mid CC \mid \mathbf{a}$

- $i = 3, j = 2$: $C \rightarrow \underline{B} C B \mid a B \mid C C \mid a$
 $\Rightarrow C \rightarrow \underline{C A B_R} C B \mid \underline{a b B_R} C B \mid a B \mid C C \mid a$
 $\Rightarrow_{(imm)} C \rightarrow a b B_R C B C_R \mid a B C_R \mid a C_R$
 $C_R \rightarrow A B_R C B C_R \mid C C_R \mid \varepsilon$

Example - Expression Grammar

G.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \end{aligned}$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid id$$

Modified Grammar

G'

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

Left Factoring

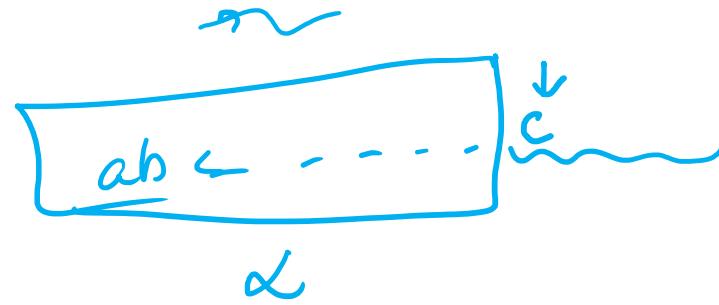
- Replace productions

$$A \rightarrow \alpha \beta_1 / \alpha \beta_2 / \dots / \alpha \beta_n / \gamma$$

with

$$A \rightarrow \underline{\alpha} A_R | \gamma$$

$$A_R \rightarrow \underline{\beta_1} / \beta_2 / \dots / \beta_n$$



Left Factoring

METHOD: For each nonterminal A , find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A -productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α , by

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. \square

Left Factoring - Example

$s \rightarrow \underline{iCtS} \mid \underline{iCtSeS} \mid a$
 $c \rightarrow b$

$s \rightarrow$
 $es \mid \epsilon$

Left Factoring

- $S \rightarrow i\underset{\text{CT}}{\underline{CTSS'}} \mid a$
- $S' \rightarrow eS \mid \epsilon$
- $C \rightarrow b$

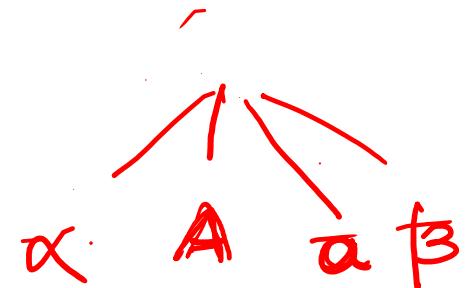
LL (1) Parser – Predictive parser

- L – input is scanned from left to right
- L – left derivation
- (1) – looking at 1 input symbol

Predictive Parser LL (1)

$$A \rightarrow \underline{a}Bb \mid \underline{c}b \mid \underline{B}C$$

- Eliminate left recursion from grammar
- Left factor the grammar
- Compute FIRST and FOLLOW
- Two variants:
 - Recursive (recursive calls)
 - Non-recursive (table-driven)



$\alpha A \underline{a} \beta$

Recursive descent with Recursive calls

- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input
- One procedure is associated with each nonterminal of a grammar
- A simple form of recursive-descent parsing, called predictive parsing, in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal
- The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

```

void S() {
    if 'c'
        match('c')
        call A()
    if 'd' match('d')
        Choose an A-production,  $A \rightarrow X_1 X_2 \dots X_k$ ;
        for ( i = 1 to k ) {
            if (  $X_i$  is a nonterminal )
                call procedure  $X_i()$ ;
            else if (  $X_i$  equals the current input symbol  $a$  )
                advance the input to the next symbol;
            else /* an error has occurred */;
        }
    }
}

```

$$S \rightarrow \underline{c}Ad$$

$$A \rightarrow a \{ cd$$

~~cad~~

下

match(t)

{ if (t == l)
 l++;

else
error;

}

Recursive descent with Recursive calls

$\Rightarrow E + T$

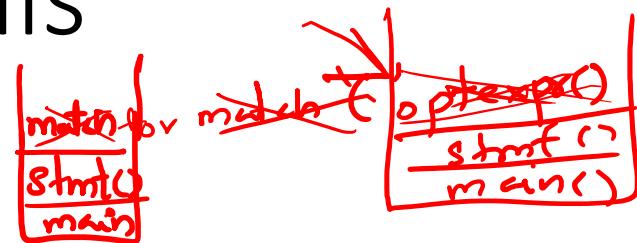
$stmt \rightarrow expr ;$
| $if (expr) stmt$
| $for (optexpr ; optexpr ; optexpr) stmt$
| $other$

$optexpr \rightarrow \epsilon \checkmark$
| $expr \checkmark$

$for (i=0 ; i < n ; i++)$
 $for (; \underbrace{i < n} ; i++)$

Recursive descent with Recursive calls

```
void stmt() {  
    switch ( lookahead ) {  
        case expr:  
            match(expr); match(';',); break;  
        case if:  
            match(if); match('('); match(expr); match(')'); stmt();  
            break;  
        case for:  
            match(for); match('(');  
            → optexpr(); match(';',); optexpr(); match(';',); optexpr();  
                match(')'); stmt(); break;  
        case other:  
            match(other); break;  
        default:  
            report("syntax error");  
    }  
}
```



for(expr ; expr;)
 ↖ ↖ ↑ ↑

Recursive descent with Recursive calls

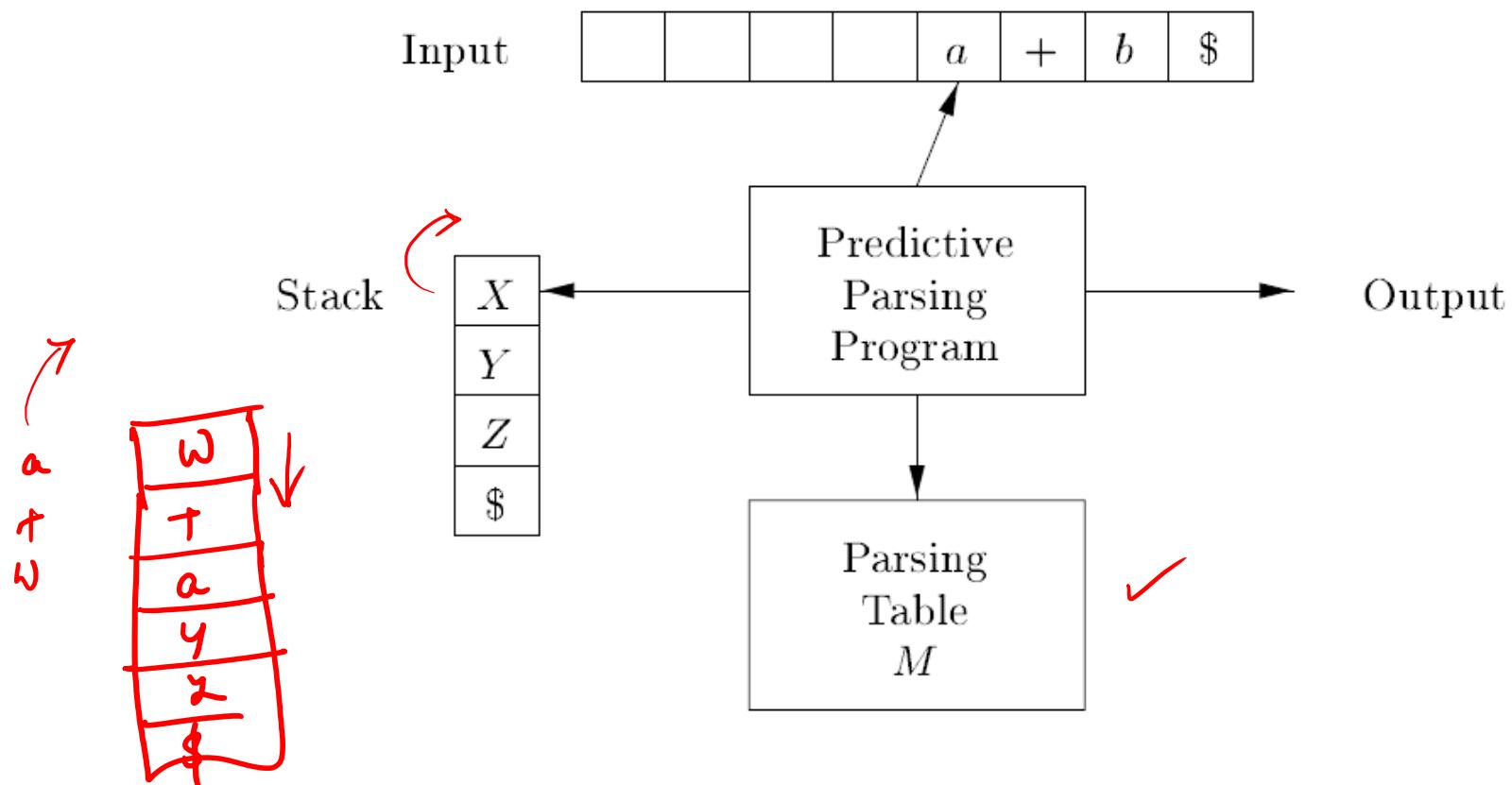
```
void optexpr() {  
    if ( lookahead == expr ) match(expr);  
}
```

```
void match(terminal t) {  
    if ( lookahead == t ) lookahead = nextTerminal;  
    else report("syntax error");  
}
```

Non-Recursive Predictive Parser

Non-Recursive Predictive Parsing

$(x,a) \Rightarrow x \rightarrow \underline{WTa}$



FIRST()

- FIRST function is computed for all terminals and non-terminals
- $\text{FIRST}_{\underline{\alpha}} = \text{the set of terminals that begin all strings derived from } \alpha$

FIRST(a)

FIRST(A)

FIRST(\underline{aA})

FIRST()

- $\text{FIRST}(a) = \{a\}$ if $a \in T$

$$\text{FIRST}(\varepsilon) = \{\varepsilon\}$$

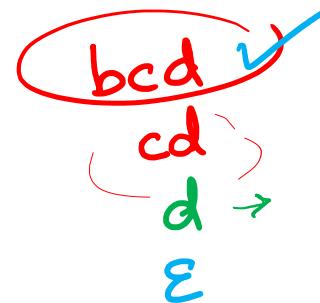
$$\text{FIRST}(A) = \bigcup_{A \rightarrow \alpha} \text{FIRST}(\alpha)$$

for $A \rightarrow \alpha \in P$

$$A \rightarrow \alpha_1 \left| \alpha_2 \right| \alpha_3$$
$$\text{FIRST}(A) = \text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2) \cup \text{FIRST}(\alpha_3)$$

FIRST () – Algorithm

- $\text{FIRST}(X_1 X_2 \dots X_k) =$
if for all $j = 1, \dots, i-1 : \varepsilon \in \text{FIRST}(X_j)$ **then**
 add non- ε in $\text{FIRST}(X_i)$ to $\text{FIRST}(X_1 X_2 \dots X_k)$
if for all $j = 1, \dots, k : \varepsilon \in \text{FIRST}(X_j)$ **then**
 add ε to $\text{FIRST}(X_1 X_2 \dots X_k)$



A \rightarrow $\begin{matrix} x_1 & x_2 & x_3 \\ B & C & D \\ \varepsilon & \varepsilon & \varepsilon \end{matrix}$
B \rightarrow $B | \varepsilon$
C \rightarrow $C | \varepsilon$
D \rightarrow $d | \varepsilon$

$$\begin{aligned}\text{FIRST}(A) &= \text{FIRST}(BCD) \\ &= \{b, \underline{\overline{c}}, \underline{\overline{d}}, \underline{\varepsilon}\}\end{aligned}$$

$$\text{FIRST}(B) = \{b, \varepsilon\}$$

$$\text{FIRST}(C) = \{c, \varepsilon\}$$

$$\text{FIRST}(D) = \{d, \varepsilon\}$$

FOLLOW

- $\text{FOLLOW}(A)$ = the set of terminals that can immediately follow non-terminal A

FOLLOW - Algorithm

- $\text{FOLLOW}(A) =$
 - if** A is the start symbol S **then**
 - add $\$$ to $\text{FOLLOW}(A)$
 - for** all $(B \rightarrow \alpha A \beta) \in P$ **do**
 - add $\text{FIRST}(\beta) \setminus \{\epsilon\}$ to $\text{FOLLOW}(A)$ ✓
 - for** all $(B \rightarrow \alpha A \beta) \in P$ and $\epsilon \in \text{FIRST}(\beta)$ **do**
 - add $\text{FOLLOW}(B)$ to $\text{FOLLOW}(A)$
 - for** all $(B \rightarrow \alpha A) \in P$ **do**
 - add $\text{FOLLOW}(B)$ to $\text{FOLLOW}(A)$

$\Rightarrow \dots c \overset{*}{B} d \dots \dots$
 $\Rightarrow \dots c \alpha A d \dots \dots$
 $\Rightarrow \dots c \alpha A b d \dots \dots$
 $c \alpha A d$
 ϵ

$B \rightarrow \alpha A \underset{\sim}{B}$
 $B \rightarrow \alpha A$

Example

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

$$FO(E) = \{), \$ \}$$

$$FO(T) = FI(E') \cup FO(E) = \{ +,), \$ \} -$$

$$FO(F) = FI(T') \cup FO(T) = \{ *, +,), \$ \} -$$

$$FO(E') = FO(E) = \{), \$ \}$$

$$FO(T') = \{ +,), \$ \}$$

$$FI(+) = \{ + \}$$

$$FI(;) = \{) \}$$

$$FI(E) = FI(T) = \{ C, id \}$$

$$FI(T) = FI(F) = \{ C, id \}$$

$$FI(F) = \{ C, id \}$$

$$FI(E') = \{ +, \underline{\epsilon} \}$$

$$FI(T') = \{ *, \underline{\epsilon} \}$$

$$FI(*) = \{ * \} \quad FI(C) = \{ C \}$$

$$FI(id) = \{ id \}$$

(id + id)

id + id \$

FIRST

- $\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F)$
 $= \{(, \text{id}\}$
- $\text{FIRST}(E') = \{ +, \varepsilon\}$
- $\text{FIRST}(T') = \{ *, \varepsilon\}$

FOLLOW

- $\text{FOLLOW}(E) = \text{FIRST}(\text{'}') \cup \{\$\}$
- $\text{FOLLOW}(T) = \text{FIRST}(E') \cup \text{FOLLOW}(E)$
 $= \{+, \$,)\}$
- $\text{FOLLOW}(F) = \{ *, +, \$,)\}$
 $\text{FIRST}(T') \cup \text{FOLLOW}(T)$

- $\text{FOLLOW}(E') = \text{FOLLOW}(E)$

$$= \{\$,)\}$$

- $\text{FOLLOW}(T') = \text{FOLLOW}(T)$

$$= \{\$, +,)\}$$

Another Example

- Ambiguous grammar

$$\begin{aligned} S &\rightarrow i \underset{A}{C} t S S' \mid a \\ S' &\rightarrow e S \mid \epsilon \\ C &\rightarrow b \end{aligned}$$

$$FO(S) = \{\$, e\}$$

$$FO(S') = \{\$\}$$

$$FO(C) = \{t\}$$

$$FI(C) = \{b\}$$

$$FI(S') = \{e, \epsilon\}$$

$$FI(S) = \{i, a\}$$

$$R \rightarrow (T) \mid c$$

$$T \rightarrow T, R \mid R$$

$$R \rightarrow (T) \mid c$$

$$\begin{aligned} T &\rightarrow RT' \mid \\ T' &\rightarrow , RT' \mid \epsilon \end{aligned}$$

$$FI(R) = \{ \{, c \}$$

$$FI(T) = \{c, \{ \}$$

$$FI(T') = \{ \epsilon, , \}$$

$$FO(R) = \{ \$, ,) \}$$

$$FO(T) = \{) \}$$

$$FO(T') = \{) \}$$

- $\text{First}(S) = \{i, a\}$
- $\text{First}(S') = \{e, \epsilon\}$
- $\text{First}(C) = \{b\}$
- $\text{Follow}(S) = \{\$, e\}$
- $\text{Follow}(S') = \{\$, e\}$

Predictive Parsing Table

- Row for each non-terminal
- Column for each terminal symbol
 - Table[NT, symbol] = Production that matches the [NT, symbol]
 - if First(NT) has ϵ , then add production
 $NT \rightarrow \epsilon$ in all [NT, a] for all 'a' in FOLLOW(NT)

Parsing Table

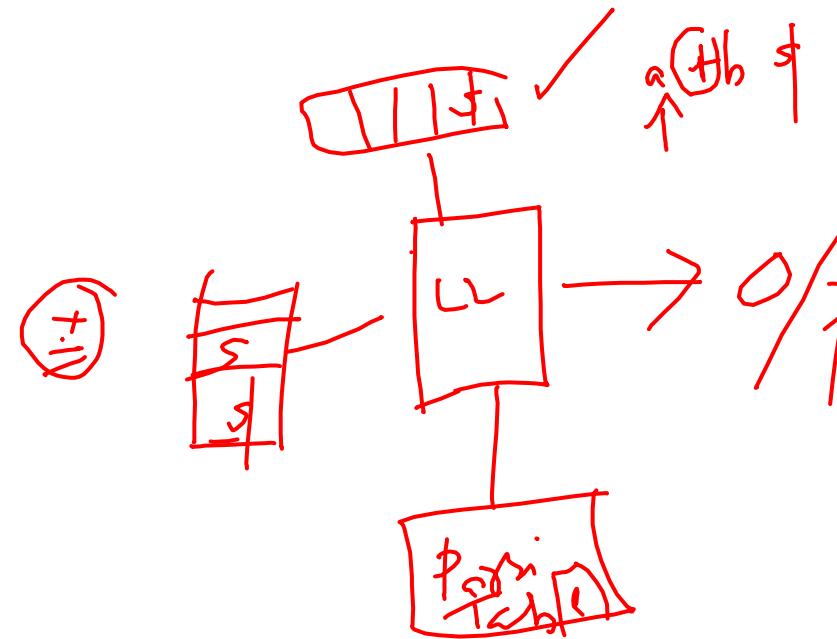
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow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

	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow i C t S S'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow e S$			$S' \rightarrow \epsilon$
C		$C \rightarrow b$				

Parsing action

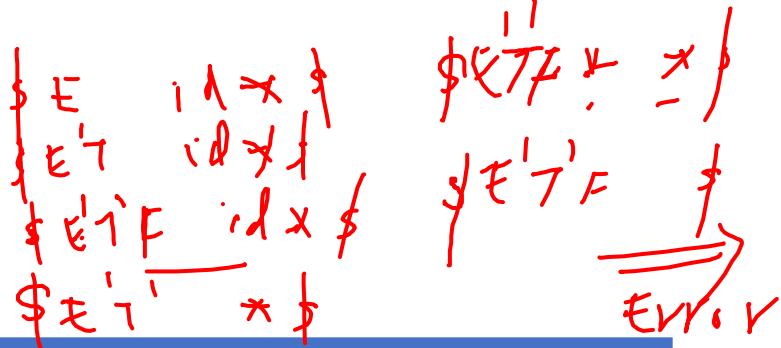
- push(\$)
push(S)
 $a := \text{lookahead}$
- repeat

```
X := pop()
if X is a terminal or X = $ then
    match(X) // move to next token, a := lookahead
else if M[X,a] =  $X \rightarrow Y_1 Y_2 \dots Y_k$  then
    push( $Y_k, Y_{k-1}, \dots, Y_2, Y_1$ ) // such that  $Y_1$  is on top
    produce output and/or invoke actions
else
    error()
endif
until X = $
```



Parsing action Example

Stack	Input String	Action
\$E	id + id* id \$	[E, id]
\$E'T	id + id *id \$	[T, id]
\$E'T'F	id + id *id \$	[F, id]
\$E'T'id	id + id *id \$	id, id -> pop stack and move input
\$E'T'	+ id *id\$	[T', +] -> replace with ε
\$E'	+ id *id\$	[E', +]
\$E'T+	+ id *id\$	+, + → pop stack and move
\$E'T	id * id \$	[T, id]
\$E'T'F	id *id\$	[F, id]



Stack	Input String	Action
\$E'T' <u>id</u>	<u>id</u> *id\$	id, id → pop
\$E'T'	*id \$	[T', *]
\$E'T' <u>F*</u>	*id \$	*, * → pop, and move
\$E'T'F	id\$	[F, id]
\$E'T'id	<u>id</u> \$	id, id → pop
\$E'T'	\$	T', \$ -> replace with ε
\$E'	\$	E', \$ -> replace with ε
\$	\$	Accept ✓

Error Recovery in LL (1) parser

- *Panic mode*
 - Discard input until a token in a set of designated synchronizing tokens is found
- *Phrase-level recovery*
 - Perform local correction on the input to repair the error
- *Error productions*
 - Augment grammar with productions for erroneous constructs
- *Global correction*
 - Choose a minimal sequence of changes to obtain a global least-cost correction

Error Recovery

- Panic Mode
 - Add synchronizing actions to undefined entries based on FOLLOW
- Phrase Mode
 - Change input stream by inserting missing +, *, (, or)
For example: **id id** is changed into **id * id or id + id**

Error Recovery

- Error Production
 - Add productions that will take care of incorrect input combinations

Error Recovery

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

LL (1)

- A grammar G is LL(1) if for each collections of productions

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

for nonterminal A the following holds:

1. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$ for all $i \neq j$
2. if $\alpha_i \Rightarrow^* \varepsilon$ then
 - 2.a. $\alpha_j \Rightarrow^* \varepsilon$ for all $i \neq j$
 - 2.b. $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$
for all $i \neq j$

If then Grammar

- The if then grammar has multiple entries in the parsing table.
- So, confusion on which production to apply
- Ambiguous grammar hence not LL (1)

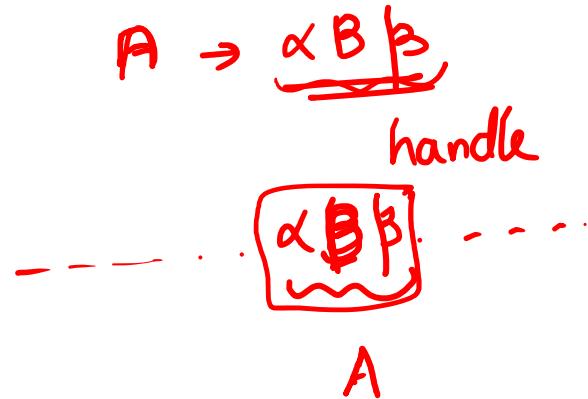
Bottom-up Parser

Bottom-up Parser

- LR methods (Left-to-right, Rightmost derivation)
 - SLR
 - Canonical LR (CALR)
 - Look Ahead LR (LALR)
- Other special cases:
 - Shift-reduce parsing
 - Operator-precedence parsing

Bottom-up parser

- Bottom-up parsers build a derivation by working from the input back toward the start symbol
 - Builds parse tree from leaves to root
 - Builds reverse rightmost derivation



Handle

- Since Bottom-up parsers match the RHS of production with LHS, a concept called ‘handle’ is defined
- A *handle* is a substring of grammar symbols in a *right-sentential form* that matches a right-hand side of a production
- A handle’s reduction to the non-terminal on the LHS represents one step along the reverse of a rightmost derivation
- This sub-string is a handle

Handle - Example

- Expression Grammar Handles
 - id
 - $E * E$
 - (E)

Shift Reduce Parser

- Simplest of the Bottom-up Parsers
- *Shift* input symbols until a handle is found.
- *Reduce* the substring to the non-terminal on the LHS of the corresponding production.

Shift Reduce Parser

- A shift-reduce parser has 4 actions:
 - *Shift* the next input symbol is shifted onto the stack
 - *Reduce* the handle that is at top of stack
 - pop handle
 - push appropriate LHS symbol
 - *Accept* and stop parsing & report success
 - *Error* recovery routine is called

Acceptance

- When the stack has the start symbol and the input is exhausted, the shift reduce parser goes to an accepting state

Consider a grammar

- 1. $E \rightarrow E + E$
- 2. $E \rightarrow E * E$
- 3. $E \rightarrow \mathbf{id}$

Stack	Action	Input
\$	shift	id + id * id \$
\$ <u>id</u>	Reduce by rule 3	+id*id \$
\$ E	shift	
\$ E+	Shift	id * id \$
\$ E + <u>id</u>	Reduce by rule 3	* id \$
\$ <u>E + E</u>	Reduce by Rule 1	* id \$
\$ E	Shift	* id \$
\$ E *	Shift	id \$

Parsing action

$$\left\{ \begin{array}{l} E \rightarrow E+T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid id \end{array} \right.$$

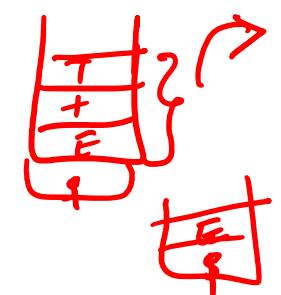
Stack	Action	Input
\$ E *	Shift	id \$
\$ E * <u>id</u>	Reduce by rule 3	\$
\$ <u>E</u> * E	Reduce by rule 2	\$
\$ E	Accept	\$

$\$$
 $\$ id$
 $\$ F$
 $\$ T$
 $\$ E$
 $\$ E +$
 $\$ E + id$

$\frac{id + id * id \$}{+ id * id \$}$
 $+ id * id \$$
 $* id \$$
 $id \$$

$\$ E + F$
 $\$ E + T =$
 $\$ E + T$
 $\$ E * T$
 $\$ E * id$
 $\$ E * F$

$* id \$$
 $id \$$
 $\$$
 Error



Conflicts

- Shift-reduce and reduce-reduce conflicts are caused by
 - The limitations of the parsing method (even when the grammar is unambiguous)
 - Ambiguity of the grammar

SLR Parsers, LR (0) items

Bottom-up Parsers

- Simple Shift-reduce parsers has lot of Shift/Reduce conflicts
- Operator precedence parsers is for a small class of grammars
- Go for LR parsers

LR Parsers

- LR(1) parsers recognize the languages in which one symbol of look-ahead is sufficient to decide whether to shift or reduce
 - L : for left-to-right scan of the input
 - R : for reverse rightmost derivation
 - 1: for one symbol of look-ahead

LR Parsers

- Read input, one token at a time
- Use stack to keep track of current state
 - The state at the top of the stack summarizes the information below.
 - The stack contains information about what has been parsed so far.

LR Parsers

- Use parsing table to determine action based on current state and look-ahead symbol.
- Parsing table construction takes into account the shift, reduce, accept or error action

LR Parsers

- SLR
 - Simple LR parsing
 - Easy to implement, but not powerful
 - Uses LR(0) items
- Canonical LR
 - Larger parser but powerful
 - Uses LR(1) items

- LALR
 - Condensed version of canonical LR
 - May introduce conflicts
 - Uses LR(1) items

SLR Parsers - Handle

- As a SLR parser processes the input, it must identify all possible handles.
- For example, consider the usual expression grammar and the input string
 $a + b.$

SLR Parsers

- If the parser has processed ‘a’ and reduced it to E. Then, the current state can be represented by $E \bullet +E$ where • means
 - E has already been parsed and
 - $+E$ is a potential suffix, which, if determined, yields to a successful parse.

$\text{E} \xrightarrow{\quad} \bullet \text{E} + \text{E}$

$\text{E} \cdot + \underline{\text{E}}$

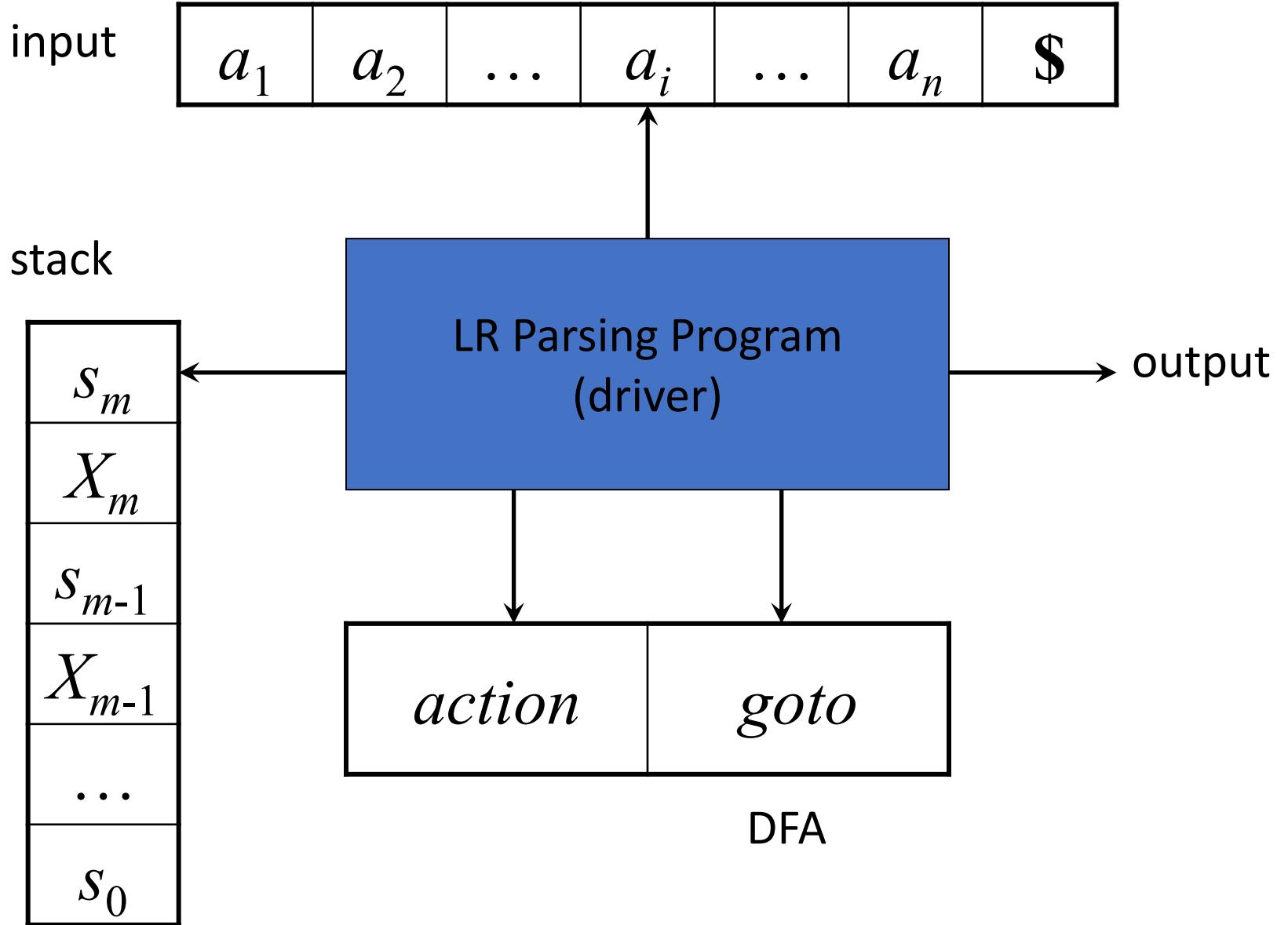
$\text{E} + \text{E} \cdot$

SLR parsers

- Our ultimate aim is to finally reach state $E+E\bullet$, which corresponds to an actual handle yielding to the reduction $E \rightarrow E+E$

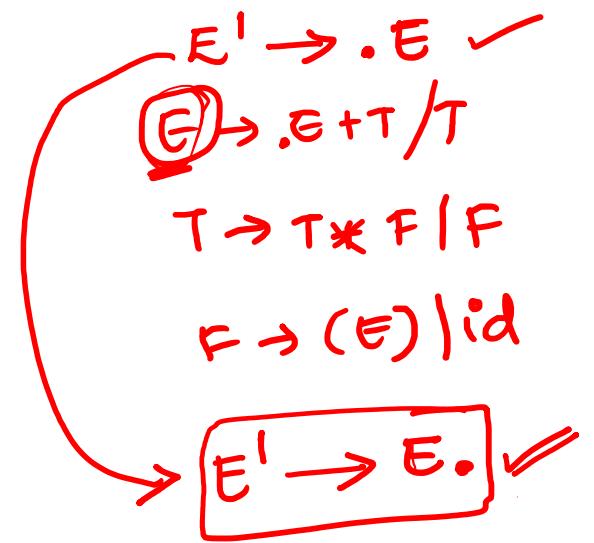
SLR Parsers

- LR parsing works by building an automata where each state represents what has been parsed so far and what we intend to parse after looking at the current input symbol. This is indicated by productions having a “.” These productions are referred to as items.
- Items that has the “.” at the end leads to the reduction by that production



SLR (1) Parser

- Form the augmented grammar
- Construction of LR(0) items
- Construct the follow() for all the non-terminals which requires construction of first() for all the terminals and non-terminals



SLR(1) parser

- Using this and the follow() of the grammar, construct the parsing table
- Using the parsing table, a stack and an input parse the input

LR (0) items \Rightarrow Itemsets

- An $LR(0)$ item of a grammar G is a production of G with a \bullet at some position of the right-hand side
- Thus, a production

$$A \rightarrow XYZ$$

has four items:

$$[A \rightarrow \bullet XYZ] = A \rightarrow \cancel{XYZ}$$

$$[A \rightarrow X \bullet Y Z]$$

$$[A \rightarrow X Y \bullet Z]$$

$$[A \rightarrow X Y Z \bullet]$$

$$A \rightarrow X \cdot \boxed{Y} Z$$

$$A \rightarrow X Y \cdot Z$$

- that production $A \rightarrow \epsilon$ has one item $[A \rightarrow \bullet]$

LR (0) items

- The grammar is augmented with a new start symbol S' and production $S' \rightarrow S$
- Initially, set $C = \text{closure}(\{[S' \rightarrow \bullet S]\})$
- For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that $\text{goto}(I, X) \notin C$ and $\text{goto}(I, X) \neq \emptyset$,
 - add the set of items $\text{goto}(I, X)$ to C
- Repeat until no more sets can be added to C

Closure (I)

- Start with $\text{closure}(I) = I$
- If $[A \rightarrow \alpha \bullet B\beta] \in \text{closure}(I)$ then for each production $B \rightarrow \gamma$ in the grammar, add the item $[B \rightarrow \bullet \gamma]$ to $\text{closure}(I)$ if it is not already there
- Repeat 2 until no new items can be added to $\text{closure}(I)$

$$A \rightarrow a \bullet B b$$

$$B \rightarrow \bullet C D$$

$$B \rightarrow \bullet d D$$

$$C \rightarrow \bullet \gamma$$

$$\begin{aligned} B &\rightarrow c D \mid d D \\ C &\rightarrow \gamma \end{aligned}$$

Goto (I , X)

- For each item $[A \rightarrow \alpha \bullet X\beta] \in I$, add the set of items $\text{closure}(\{[A \rightarrow \alpha X \bullet \beta]\})$ to $\text{goto}(I, X)$ if not already there
- Repeat until no more items can be added to $\text{goto}(I, X)$
- Intuitively, $\text{goto}(I, X)$ is the set of items that are valid for the viable prefix γX when I is the set of items that are valid for γ

Augmented Grammar

$$E' \rightarrow E$$

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Augmented Grammar

1. $E' \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow T * F$
5. $T \rightarrow F$
6. $F \rightarrow (E)$
7. $F \rightarrow id$

$$g_0 : \begin{array}{l} E' \rightarrow .E \\ E \rightarrow .E + T \\ E \rightarrow .T \\ T \rightarrow .T * F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .id \end{array}$$

$g_1 : \text{Goto}(g_0, E)$

$E' \rightarrow E.$

$E \rightarrow E. + T$

$g_2 : \text{Goto}(g_0, T)$
 $E \rightarrow T.$ $\text{Goto}(g_4, T)$

$T \rightarrow T. * F$

$g_3 : \text{Goto}(g_0, F)$

$T \rightarrow F.$ $\text{Goto}(g_4, F)$
 $\text{Goto}(g_6, F)$

$g_5 : \text{Goto}(g_0, id)$
 $F \rightarrow id.$ $\text{Goto}(g_4, id)$
 $\text{Goto}(g_6, id)$
 $\text{Goto}(g_7, id)$

$g_6 : \text{Goto}(g_0, ())$
 $F \rightarrow (- E)$ $\text{Goto}(g_4, C)$
 $E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$g_7 : \text{Goto}(g_1, +)$

$E \rightarrow E + .T$ $\text{Goto}(g_8, +)$

$T \rightarrow .T * F$

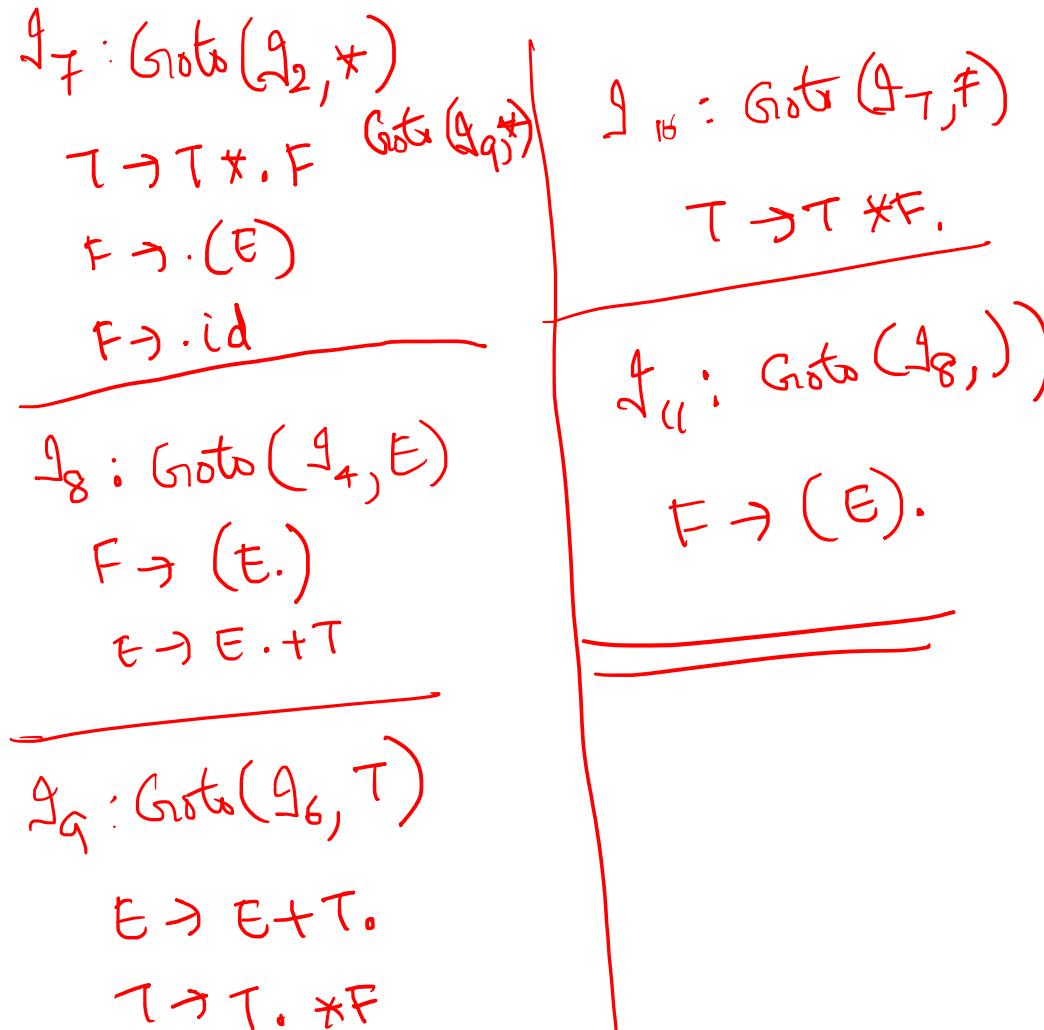
$T \rightarrow .F$

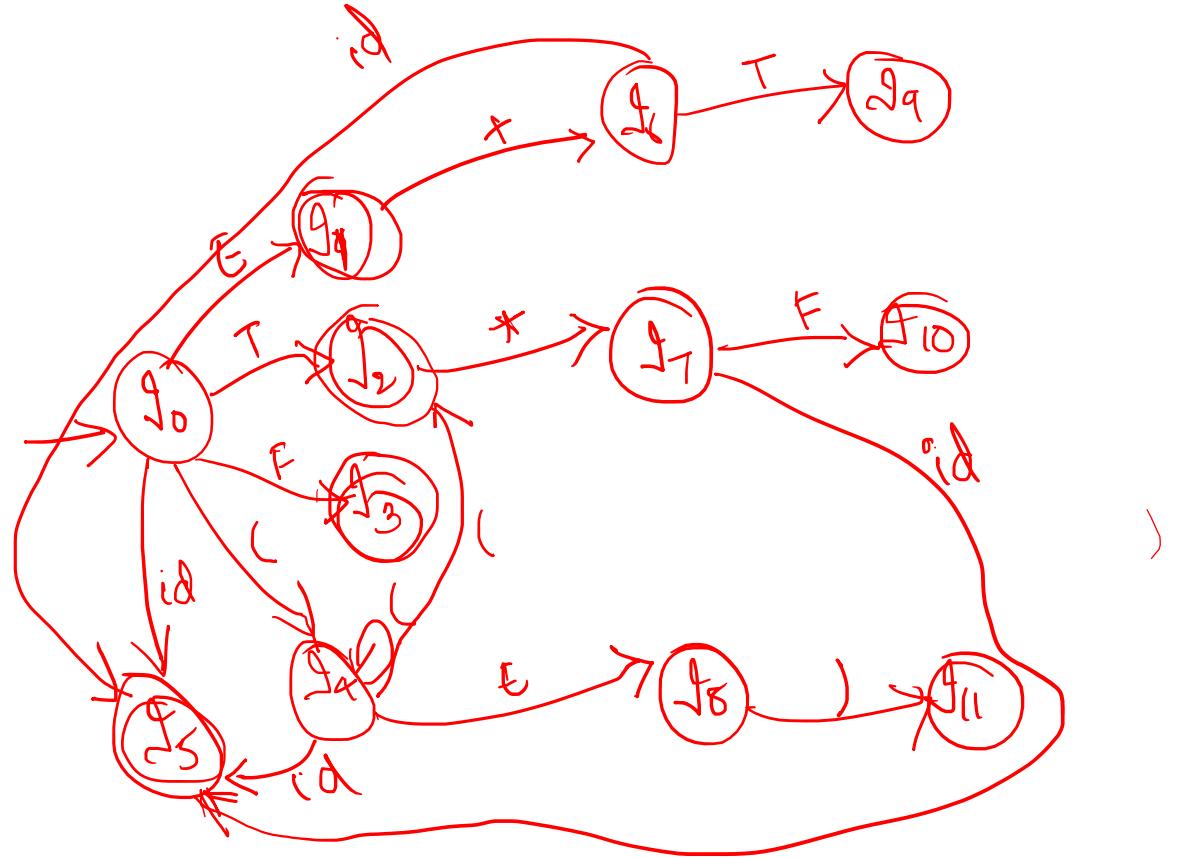
$F \rightarrow .(E)$

$F \rightarrow .id$

I_0

- $E' \rightarrow .E$
- $E \rightarrow .E + T$
- $E \rightarrow .T$
- $T \rightarrow .T * F$
- $T \rightarrow .F$
- $F \rightarrow .(E)$
- $F \rightarrow .id$





Items

- I_0
- $E' \rightarrow .E$
- $E \rightarrow .E + T$
- $E \rightarrow .T$
- $T \rightarrow .T * F$
- $T \rightarrow .F$
- $F \rightarrow .(E)$
- $F \rightarrow .id$
- $I_1 = \text{Goto}(I_0, E)$
- $E' \rightarrow E.$
- $E \rightarrow E. + T$
- $I_2 = \text{Goto}(I_0, T), \text{Goto}(I_4, T),$
 $E \rightarrow T.$
- $T \rightarrow T.*F$

- $I_3 = \text{Goto}(I_0, F), \text{Goto}(I_4, F), \text{Goto}(I_6, F)$
- $T \rightarrow F.$
- $I_5 = \text{Goto}(I_0, \text{id}), \text{Goto}(I_4, \text{id}), \text{Goto}(I_6, \text{id}), \text{Goto}(I_7, \text{id})$
 $F \rightarrow \text{id}.$
- $I_4 = \text{Goto}(I_0, (), \text{Goto}(I_4, (), \text{Goto}(I_6, (), \text{Goto}(I_7, ()$
- $F \rightarrow (.E)$
- $E \rightarrow .E + T$
- $E \rightarrow .T$
- $T \rightarrow .T * F$
- $T \rightarrow .F$
- $F \rightarrow .(E)$
- $F \rightarrow .\text{id}$

Items

$I_6 = \text{Goto}(I_1, +), \text{Goto}(I_8, +),$

$E \rightarrow E + . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_8 = \text{Goto}(I_4, E)$

$F \rightarrow (E.)$

$E \rightarrow E . + T$

$I_9 = \text{Goto}(I_6, T)$

$E \rightarrow E + T.$

$T \rightarrow T . * F$

$I_7 : \text{Goto}(I_2, *), \text{Goto}(I_9, *)$

$T \rightarrow T * . F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_{10} : \text{Goto}(I_7, F)$

$T \rightarrow T * F.$

$I_{11} : \text{Goto}(I_8,))$

$F \rightarrow (E).$

SLR Parsing Table

- Input: Augmented Grammar G'
- Output: SLR parsing table with functions, shift, reduce and accept
- Parsing table is between items and Terminals and non-terminals
- The non-terminals correspond to the `goto()` of the items set
- The terminals have the parsing table corresponding to the action – shift / reduce/accept

SLR Parsing Table

- Augment the grammar with $S' \rightarrow S$
- Construct the set $C = \{I_0, I_1, \dots, I_n\}$ of $LR(0)$ items
- If $[A \rightarrow \alpha \bullet a\beta] \in I_i$ and $goto(I_i, a) = I_j$ then set $action[i, a] = \text{shift } j$, where a is a terminal
- If $[A \rightarrow \alpha \bullet] \in I_i$ then set $action[i, a] = \text{reduce } A \rightarrow \alpha$ for all $a \in FOLLOW(A)$ where $A \neq S'$

SLR parsing table

- If $[S' \rightarrow S \bullet]$ is in I_i , then set $action[i, \$] = \text{accept}$
- If $goto(I_i, A) = I_j$ then set $goto[i, A] = j$
- Repeat for all the items until no more entries added
- The initial state i is the I_i holding item $[S' \rightarrow \bullet S]$
- All other entries are error

Grammar

- $E' \rightarrow E$
- 1 • $E \rightarrow E + T$
- 2 • $E \rightarrow T$
- 3 • $T \rightarrow T * F$
- 4 • $T \rightarrow F$
- 5 • $F \rightarrow (E)$
- 6 • $F \rightarrow id$

Follow

- $\text{Follow}(E) = \{\$, +,)\}$
- $\text{Follow}(T) = \{\$, +, *,)\}$
- $\text{Follow}(F) = \{\$, +, *,)\}$

- s_i means shift state i
- r_j means reduce by production numbered j
- Blank means error

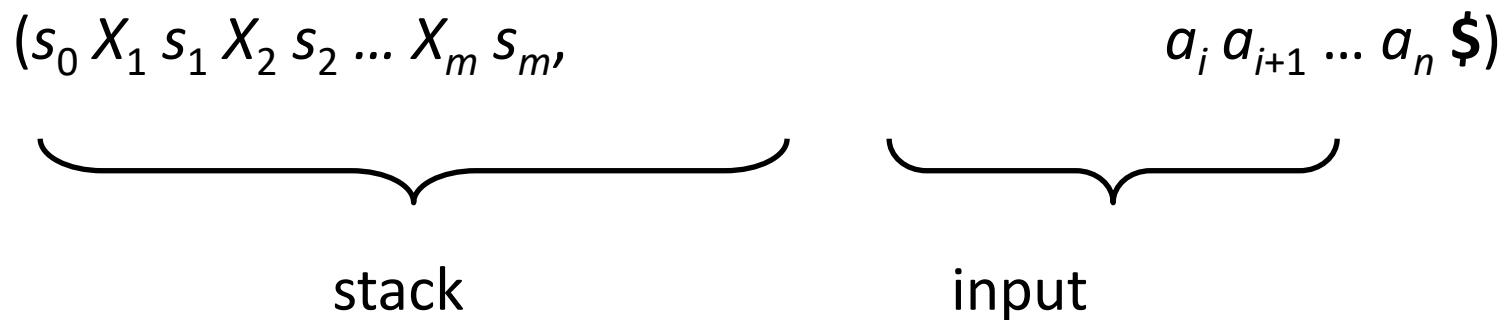
Shift, Accept, Reduce

State	Action							Goto		
	id	+	*	()	\$	E	T	F	
0	s5				s4		1	2	3	
1		s6				accept				
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	

Shift, Accept and Reduce

State	Action							Goto		
	id	+	*	()	\$	E	T	F	
7	s5				s4					10
8			s6			s11				
9		r1	s7			r1	r1			
10		r3	r3		r3	r3				11
11		r <u>5</u>	r5		r5	r5				

SLR Parsing



Parsing action

- If $action[s_m, a_i] = \text{shift } s$, then push a_i , push s , and advance input:
 $(s_0 X_1 s_1 X_2 s_2 \dots X_m \underline{s_m} a_i s, \ a_{i+1} \dots a_n \$)$
- If $action[s_m, a_i] = \text{reduce } A \rightarrow \beta$ and $goto[s_{m-r}, A] = s$ with $r=|\beta|$ then
pop $2r$ symbols, push A , and push s :
 $(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} \underline{s_{m-r}} \underline{A} s, \ \underline{a_i} \underline{a_{i+1}} \dots a_n \$)$

$$E \rightarrow \boxed{E + T}$$

2 x 3

$s_0 \dots - \underline{s_{m-r}}$ $\boxed{s_m}$,

$s_0 \dots - \underline{s_{m-r}} A s$

$\underline{a_i} \ a_{i+1} \dots$

a_i

- If $\text{action}[s_m, a_i] = \text{accept}$, then stop
- If $\text{action}[s_m, a_i] = \text{error}$, then attempt recovery

Parsing algorithm

- Set input to point to the first symbol of w\$
- Repeat
 - Let s be the state on the top of the stack
 - Let a be the symbol pointed to by ip
 - If action [s, a] = shift s' then
 - Push a then s' on top of the stack
 - Move input to the next input symbol
 - Else if action [s, a] = reduce A → β then
 - Pop 2 * | β | symbols off the stack
 - Let s' be the state now on the top of the stack
 - Push A then goto [s', A] on top of the stack
 - Output the production A → β
 - Else if action[s, a] = accept then return;
 - Else error()

Parsing action

Stack	Input	Action
0	id * id + id \$	[0, id] → s5 , shift
0 id 5	* id + id \$	[5, *] → r6, pop 2 symbols, Goto[0, F] → 4
0 F 3	* id + id \$	[3, *] → r4, pop 2 symbols, Goto[0, T] → 2
0 T 2	* id + id \$	[2, *] → s7, shift
0 T 2 * 7	id + id \$	[7, id] → s5, shift
0 T 2 * 7 id 5	+ id \$	[5, +] → r6, pop 2 symbols, Goto[7, F] → 10
0 T 2	+ id \$	[2, +] → r2, pop 2 symbols and goto [0 , E] → 1

Parsing action

Stack	Input	Action
0 E 1	+ id \$	[1, +] → s6, shift
0 E 1 + 6	id \$	[6, id] → s5, shift
0 E 1 + 6 id 5	\$	[5, \$] → r6, pop 2 symbols, goto [6, F] → 3
0 E 1 + 6 F 3	\$	[3 , \$] → r4, pop 2 symbols, goto [6, T] → 9
0 E 1 + 6 T 9	\$	[9, \$] → r1, pop 6 symbols, goto [0, E] → 1
0 E 1	\$	[1, \$] → accept, hence successful parsing

Problems with SLR grammar

- Every SLR grammar is unambiguous, but **not** every unambiguous grammar is SLR
- Consider for example the unambiguous grammar

Example

$$\begin{aligned} id &= *id \\ *id &= *id \end{aligned}$$

- $S \rightarrow L = R$
- $S \rightarrow R$
- $L \rightarrow *R$
- $L \rightarrow id$
- $R \rightarrow L$

Items set

- I_0 :
 $S' \rightarrow \bullet S$
 1. $S \rightarrow \bullet L = R$
 2. $S \rightarrow \bullet R$
 3. $L \rightarrow \bullet^* R$
 4. $L \rightarrow \bullet \text{id}$
 5. $R \rightarrow \bullet L$
- I_1 : (I_0, S)
 $S' \rightarrow S \bullet$
- I_2 : (I_0, L)
 $S \rightarrow L \bullet = R$
 $R \rightarrow L \bullet$
- I_3 : (I_0, R)
 $S \rightarrow R \bullet$

Items set

- $I_4: (I_0, *) (I_4, *) (I_6, *)$
 $L \rightarrow * \bullet R$
 $R \rightarrow \bullet L$
 $L \rightarrow \bullet^* R$
 $L \rightarrow \bullet \mathbf{id}$
- $I_5: (I_0, \mathbf{id}) (I_4, \mathbf{id}) (I_6, \mathbf{id})$
 $L \rightarrow \mathbf{id} \bullet$
- $I_9: (I_6, R)$
 $S \rightarrow L = R \bullet$
- $I_6: (I_2, =)$
 $S \rightarrow L = \bullet R$
 $R \rightarrow \bullet L$
 $L \rightarrow \bullet^* R$
 $L \rightarrow \bullet \mathbf{id}$
- $I_7: (I_4, R)$
 $L \rightarrow * R \bullet$
- $I_8: (I_4, L) (I_6, L)$
 $R \rightarrow L \bullet$

- $\text{Follow}(S) = \{ \$ \}$
- $\text{Follow}(L) = \{ =, \$ \}$
- $\text{Follow}(R) = \{ \$, = \}$

State	Action				Goto		
	id	=	*	\$	S	L	R
0	s5		s4		1	2	3
1				accept			
2			s6 / r5		r5		
3				r2			
4	s5		s4			8	7
5		r4		r4			
6	s5		s4			8	9

State	Action				Goto		
	id	=	*	\$	S	L	R
7		r3		r3			
8		r5		r5			
9				r1			

0
0 id5

0 L2

0 R3

$$\begin{aligned} id &= *id \$ \\ &= *id \$ \\ &= *id \$ \\ &= *id \$ \end{aligned}$$

r4 L → id
r5 R → L

Reduce → Error

shift

0 L2 = 6

* id \$

0 L2 = 6 * 4

id \$

0 L2 = 6 * 4 id5

\$ L → id

0 L2 = 6 * 4 L8

\$ R → L

0 L2 = 6 * 4 RT

\$ R3

0 L2 = 6 L8

\$ R → L

0 L2 = 6 R9

\$ R1

0 S1

\$ accept

Conflict

- Shift / reduce conflict arises
- Because the grammar is not SLR(1)
- Follow information alone is not sufficient
- Hence, powerful parser is required

Summary

- Learnt to parse the SLR(1) grammar using the SLR(1) parsing algorithm
- Some grammar results in Shift / Reduce conflict

CALR Parsing

Conflict in SLR parsers

- Shift / reduce conflict arises
- Follow information alone is not sufficient to decide when to reduce.
- Hence, powerful parser is required

Conflicts in SLR parsers

- In SLR, if there is a production of the form $A \rightarrow \alpha\cdot$, then a reduce action takes place based on $\text{follow}(A)$
- There would be situations, where when state i appears on the TOS, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by terminal 'a' in a right sentential form
- Hence, the reduction $A \rightarrow \alpha$ would be invalid on input 'a'

CALR parsers motivation

- If it is possible to do more in the states that allow us to rule out some of the invalid reduction, introduce more states
- Introduce exactly which input symbols to follow a particular non-terminal

CALR parsers

- Construct LR(1) items
- Use these items to construct the CALR parsing table involving action and goto
- Use this table, along with input string and stack to parse the string

CALR motivation

- Extra symbol is incorporated in the items to include a terminal symbol as a second component
- $A \rightarrow [\alpha \cdot \beta, a]$ where $A \rightarrow \alpha\beta$ is a production and 'a' is a terminal or the right end marker \$ - LR(1) item

$\underline{A \rightarrow \alpha \cdot \beta}, \underline{a}$

↓
terminal and \$

\$ a

LR(1) item

- 1 – refers to the length of the second component – lookahead of the item
- Lookahead has no effect in $A \rightarrow [\alpha .\beta , a]$ where β is not ϵ , but $A \rightarrow [\alpha . , a]$ calls for a reduction $A \rightarrow \alpha$ if the next input symbol is ‘a’, ‘a’ will be subset of $\text{follow}(A)$

LR(1) item

- $A \rightarrow [\alpha .\beta, a]$ is a valid item for a viable prefix γ if there is a derivation $S \Rightarrow \delta A w \Rightarrow \delta \alpha \beta w$ where $\gamma = \delta \alpha$ and either 'a' is the first symbol of 'w' or 'w' is ϵ and 'a' is $\$$

LR(1) item algorithm

- Closure (I)

{repeat for each item $[A \xrightarrow{\underline{\alpha}} \alpha \cdot B\beta, a]$ in I ,

for each production $B \rightarrow \gamma$ in G' and

each terminal b in $\text{First}(\underline{\beta}a)$ such that $[B \rightarrow \cdot \underline{\gamma}, b]$ is not in I do

add $[B \rightarrow \cdot \underline{\gamma}, b]$ to set I

until no more items can be added to I

end }

$$G \Rightarrow \underline{G'} \quad s' \rightarrow s$$

$\text{first}(\underline{\beta}a)$

$$A \rightarrow \alpha \cdot B \xrightarrow{\underline{\beta}} (\underline{a})$$

$$B \rightarrow \cdot \gamma, \underline{b}$$

$\text{first}(a)$

$$A \rightarrow \alpha \cdot B, a$$

$$B \rightarrow \cdot \gamma, \underline{a}$$

Goto(I, X)

Begin

 Initialize J to be the empty set

 For each item $[A \rightarrow \alpha.X\beta, a]$ in I such that

 add item $[A \rightarrow \alpha X. \beta, a]$ to set J;

 Return closure(J)

end

Items(G')

Begin C:= closure ({ $S' \rightarrow .S, \$$ });

repeat

 for each set of items I in C

 for each grammar symbol X

 if goto(I,X) is not empty and not in C

 add goto(I, X) to C;

until no more set of items can be added to C

end

Example

1 • $S \rightarrow CC$

2 • $C \rightarrow cC$

3 • $C \rightarrow d$

$\text{FIRST}(C) = \{c, d\}$

• Augmented

• $S' \rightarrow S$

• $S \rightarrow CC$

• $C \rightarrow cC$

• $C \rightarrow d$

LR(1) items

- $I_0 :$
 $S' \xrightarrow{A \rightarrow \alpha} .S, \$$ $F_1(\beta a) = F_1(\$)$
- $S \xrightarrow{A \rightarrow \alpha} .CC, \$$ $F_1(C\$) = \{c, d\}$
- $C \rightarrow .cC, c/d$ (first($C\$$))
- $C \rightarrow .d, c/d$

- $I_1 : \text{goto}(I_0, S)$
 $S' \rightarrow S., \$$
- $I_2 : \text{goto}(I_0, C)$
 $S \rightarrow C.C, \$$ $F_1(\$)$
- $C \rightarrow .cC, \$$
- $C \rightarrow .d, \$$

$C \rightarrow c.C, \$$

$C \rightarrow .cc, \$$

$C \rightarrow .d, \$$

- $I_3 : \text{goto}(I_0, c), \text{goto}(I_3, c),$
 $C \rightarrow \underline{\underline{c.C}}, \underline{c/d}$

$C \rightarrow .cC, c/d$

$C \rightarrow .d, c/d$

- $I_4 : \text{goto}(I_0, d) \text{ goto}(I_3, d)$

$C \rightarrow d., c/d$

- $I_5 : \text{goto}(I_2, C)$
 $S \rightarrow CC., \$$
- $I_6 : \text{goto}(I_2, c) \text{ goto}(I_6, c)$

$\left. \begin{array}{l} C \rightarrow c.C, \$ \\ C \rightarrow .cC, \$ \\ C \rightarrow .d, \$ \end{array} \right\}$

- $I_7 : goto(I_2, d) \quad goto(I_6, d)$

$\checkmark C \rightarrow d., \$$

- $I_8 : goto(I_3, C)$

$C \rightarrow cC., c/d$

- $I_9 : goto(I_6, C)$

$C \rightarrow cC., \$$

Parsing Table

- Construct $C = \{I_0, I_1, I_2 \dots I_n\}$ the collection of LR(1) items for G'
- State I of the parser is from I_i
 - if $[A \rightarrow \alpha.a\beta, b]$ is in I_i and $\text{goto}(I_i, a) = I_j$ set action $[i, a] = \text{shift } j$, where a is a terminal
 - if $[A \rightarrow \alpha . , a]$ is in I_i and $A \neq S'$, then set action $[i, a] = \text{reduce by } A \rightarrow \alpha$
// a conflict here implies the grammar is not CALR grammar
- If $\text{goto}(I_i, A) = I_j$ then $\text{goto}(i, A) = j$
- $[S' \rightarrow .S, \$]$ implies an accept action
- All other entries are error

Parsing table - CALR

Stat	Action			goto	
e	c	d	\$	s	c
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9

State	Action			goto	
	c	d	\$	s	c
7			r3		
8	r2	r2			
9			r2		

Parsing algorithm

- Set input to point to the first symbol of w\$
- Repeat
 - Let s be the state on the top of the stack
 - Let a be the symbol pointed to by ip
 - If action [s, a] = shift s' then
 - Push a then s' on top of the stack
 - Move input to the next input symbol
 - Else if action [s, a] = reduce $A \rightarrow \beta$ then
 - Pop 2 * | β | symbols off the stack
 - Let s' be the state now on the top of the stack
 - Push A then goto $[s', A]$ on top of the stack
 - Output the production $A \rightarrow \beta$
 - Else if action[s, a] = accept then return;
 - Else error()

Parsing with CALR parser

Stack	Input	Action
0	ccdd\$	[0, c] – shift 3
0 c 3	c d d \$	[3, c] – shift 3
0 c 3 c 3	d d \$	[3, d] – shift 4
0 c 3 c 3 d 4	d \$	[4, d] – reduce 3, pop 2 symbols from stack, push C, goto(3, C) = 8 $c \rightarrow d$
0 c 3 c 3 C 8	d \$	[8, d] – reduce 2, pop 4 symbols from the stack, push C, goto(3, C) = 8 $c \rightarrow cC$
0 c 3 C 8	d \$	[8, d] – reduce 2, pop 4 symbols from the stack, push C, goto(0, C) = 2 $C \rightarrow CC$

0C2

Stack	Input	Action
0 C 2	d \$	[2, d] – shift 7
0 C 2 d <u>7</u>	\$	[7, \$] – reduce 3, pop 2 symbols from the stack, goto(2, C) = 5 <u>C → d</u>
0 C 2 <u>C 5</u>	\$	[5, \$] – reduce 1, pop 4 symbols off the stack, goto(0, S) = 1 <u>S → CC</u>
0 S 1	\$	[1, \$] – accept – successful parsing

Example

- $S' \rightarrow S$
- $S \rightarrow L = R$
- $S \rightarrow R$
- $L \rightarrow *R$
- $L \rightarrow id$
- $R \rightarrow L$

q₀:

$$S' \xrightarrow{\quad} .S^B, \$$$

$$S \xrightarrow{\quad} .L^B = R^A, \$ \quad FI(= R \$) - \{ = \}$$

$$S \xrightarrow{\quad} .R^B, \$$$

$$\left\{ \begin{array}{l} L \xrightarrow{\quad} . *R, = / \$ \\ L \xrightarrow{\quad} . id, = / \$ \end{array} \right.$$

$$R \xrightarrow{\quad} .L^B, \$$$

$$FI(\$) = \$ \quad \left\{ \begin{array}{l} L \xrightarrow{\quad} . *R, \$ \\ L \xrightarrow{\quad} . id, \$ \end{array} \right.$$

Another Example

- I_0
 $[S' \rightarrow \bullet S, \$] \text{ goto}(I_0, S) = I_1$
 $[S \rightarrow \bullet L=R, \$] \text{ goto}(I_0, L) = I_2$
 $[S \rightarrow \bullet R, \$] \text{ goto}(I_0, R) = I_3$
 $[L \rightarrow \bullet *R, =/\$] \text{ goto}(I_0, *) = I_4$
 $[L \rightarrow \bullet \text{id}, =/\$] \text{ goto}(I_0, \text{id}) = I_5$
 $[R \rightarrow \bullet L, \$] \text{ goto}(I_0, L) = I_2$
- $I_1 : \text{ goto}(I_0, S)$
 $[S' \rightarrow S \bullet, \$]$
- $I_2 : \text{ goto}(I_0, L)$
 $[S \rightarrow L \bullet=R, \$] \text{ goto}(I_2, =) = I_6$
 $[R \rightarrow L \bullet, \$]$
- $I_3 : \text{ goto}(I_0, R)$
 $[S \rightarrow R \bullet, \$]$
- $I_4 : \text{ goto}(I_0, *) \text{ goto}(I_4, *)$
 $[L \rightarrow * \bullet R, =/\$] \text{ goto}(I_4, R) = I_7$
 $[R \rightarrow \bullet L, =/\$] \text{ goto}(I_4, L) = I_8$
 $[L \rightarrow \bullet *R, =/\$] \text{ goto}(I_4, *) = I_4$
 $[L \rightarrow \bullet \text{id}, =/\$] \text{ goto}(I_4, \text{id}) = I_5$
- $I_5 : \text{ goto}(I_0, \text{id}) \text{ goto}(I_4, \text{id})$
 $[L \rightarrow \text{id} \bullet, =/\$]$

- $I_6 : \text{goto}(I_2, =)$

$[S \rightarrow L=\bullet R, \$] \text{ goto}(I_6, R) = I_9$
 $[R \rightarrow \bullet L, \$] \text{ goto}(I_6, L) = I_{10}$
 $[L \rightarrow \bullet^* R, \$] \text{ goto}(I_6, *) = I_{11}$
 $[L \rightarrow \bullet \text{id}, \$] \text{ goto}(I_6, \text{id}) = I_{12}$

- $I_7 : \text{goto}(I_4, R)$

$[L \rightarrow *R\bullet, =/\$]$

- $I_8 : \text{goto}(I_4, L)$

$[R \rightarrow L\bullet, =/\$]$

- $I_9 : \text{goto}(I_6, R)$

$[S \rightarrow L=R\bullet, \$]$

- $I_{10} : \text{goto}(I_6, L) \text{ goto}(I_{11}, L)$

$[R \rightarrow L\bullet, \$]$

- $I_{11} : \text{goto}(I_6, *) \text{ goto}(I_{11}, *)$

$[L \rightarrow * \bullet R, \$] \text{ goto}(I_{11}, R) = I_{13}$

$[R \rightarrow \bullet L, \$] \text{ goto}(I_{11}, L) = I_{10}$

$[L \rightarrow \bullet^* R, \$] \text{ goto}(I_{11}, *) = I_{11}$

$[L \rightarrow \bullet \text{id}, \$] \text{ goto}(I_{11}, \text{id}) = I_{12}$

- $I_{12} : \text{goto}(I_6, \text{id}) \text{ goto}(I_{11}, \text{id})$
 $[L \rightarrow \mathbf{id}\bullet, \$]$

- $I_{13} : \text{goto}(I_{11}, R)$
 $[L \rightarrow *R\bullet, \$]$

Parsing Table

State	Action					goto		
	id	*	=	\$	S	L	R	
0	s5	s4			1	2	3	
1				accept				
2			s6	r5				
3				r2				
4	s5	s4				8	7	
5			r4	r4				
6	s12	s11				10	9	

State	Action				Goto		
	id	*	=	\$	S	L	R
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

Summary

- CALR – most powerful parser
- Have so many items and states
- No conflicts

Parser Generator

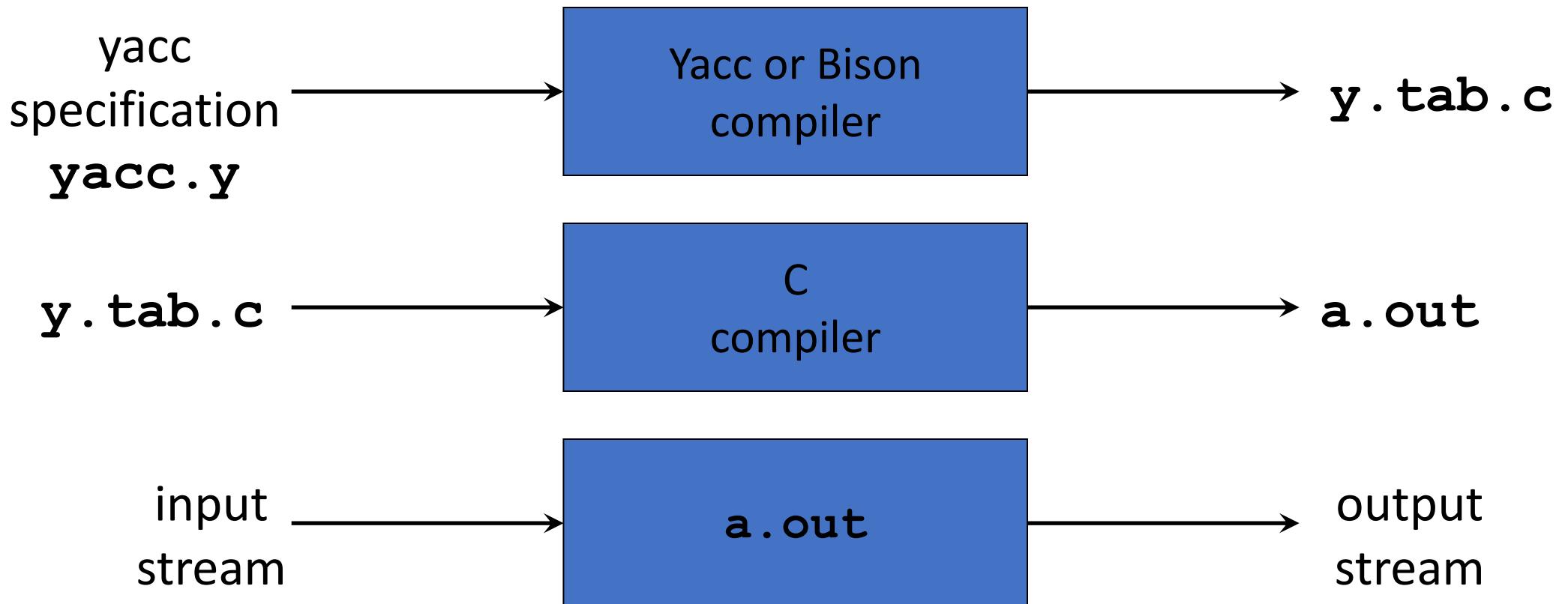
Parser Generator

- There is enough integration between the lexer and the parser
- Lexer has already been implemented using a tool LEX
- Parser could also be done with a tool so as to help speed up

ANTLR, Yacc, and Bison

- *ANTLR* tool generates LL(k) parsers
- *Yacc* (Yet Another Compiler Compiler) generates LALR(1) parsers
- *Bison* (Yacc improved)
- As bottom up parsers are preferred, YACC is preferred and used tool

Creating an LALR(1) Parser with Yacc/Bison



Lex v.s. Yacc

- Lex
 - Lex generates C code for a lexical analyzer, or **scanner**
 - Lex uses patterns that match strings in the input and converts the strings to tokens
- Yacc
 - Yacc generates C code for syntax analyzer, or **parser**.
 - Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree.

Yacc Specification

- A *yacc specification* consists of three parts:
yacc declarations, and C declarations in % { % }
%%
translation rules
%%
user-defined auxiliary procedures
- *Translation rules* are grammar productions and actions:
production₁{ semantic action₁ }
production₂{ semantic action₂ }
...
production_n{ semantic action_n }

Writing a Grammar in Yacc

$$E \rightarrow E + T \mid T$$
$$E : E^+ \mid T^+$$

- Productions in Yacc are of the form

Nonterminal : tokens/nonterminals { *action* }

| tokens/nonterminals { *action* }
...

;

Grammar in YACC

- Tokens that are single characters can be used directly within productions, e.g. '+'
- Named tokens must be declared first in the declaration part using
%token TokenName

Synthesized Attributes

- Semantic actions may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:

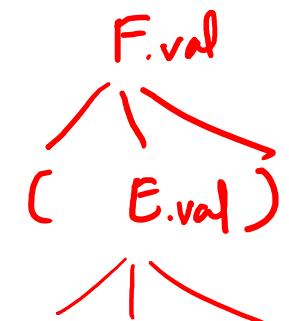
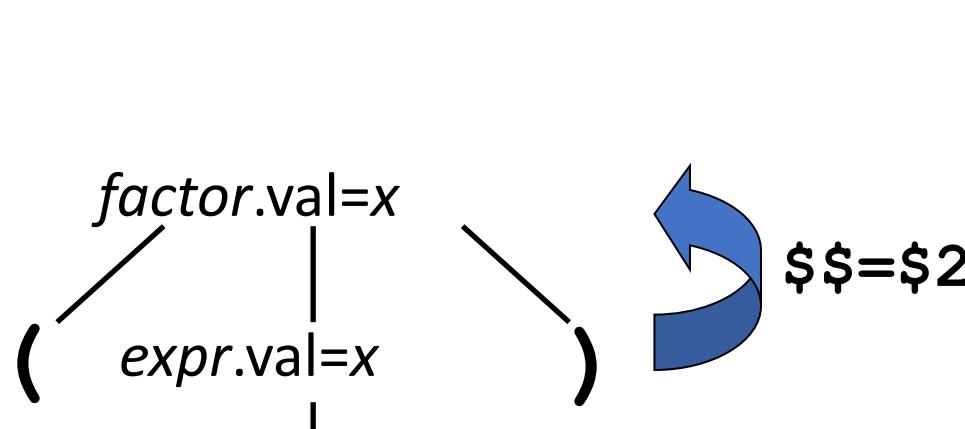
$X : Y_1 Y_2 Y_3 \dots Y_n \quad \{ \text{action} \}$

- $\$\$$ refers to the value of the attribute of X
- $\$_i$ refers to the value of the attribute of Y_i

- For example

factor : ' (' expr ') ' { \$\$=\$2; }

F → C E



yyparse()

- Called once from main() [*user-supplied*]
- Repeatedly calls yylex() until done:
 - On syntax error, calls yyerror() [*user-supplied*]
 - Returns 0 if all of the input was processed;
 - Returns 1 if syntax error

Example:

```
int main() { return yyparse(); }
```

Definitions

- Information about tokens:
 - token names:
 - declared using '**%token**'
 - single-character tokens don't have to be declared
 - any name not declared as a token is assumed to be a nonterminal.
 - start symbol of grammar, using '**%start**' [optional]
 - operator info:
 - precedence, associativity
 - stuff to be copied verbatim into the output (e.g., declarations, **#includes**): enclosed in **%{ ... %}**

Definitions Section

```
% {  
#include <stdio.h>  
#include <stdlib.h>  
% }  
%token ID NUM  
%start expr
```

YACC Rules

Grammar production

$A \rightarrow B_1 B_2 \dots B_m$

$A \rightarrow C_1 C_2 \dots C_n$

$A \rightarrow D_1 D_2 \dots D_k$



yacc rule

$A : \underbrace{B_1 B_2 \dots B_m}_{|} \{ \}$
 $A : \underbrace{C_1 C_2 \dots C_n}_{|} \{ \}$
 $A : \underbrace{D_1 D_2 \dots D_k}_{; } \{ \}$

$A \rightarrow B_1 | B_2 | B_3$

- Rule RHS can have arbitrary C code embedded, within { ... }. E.g.:

$A : \underbrace{B_1}_{\text{printf("after B1\n"); x = 0; }} \underbrace{B_2}_{\{ x++; \}} \underbrace{B_3}_{\{ \}} \dots \quad A \rightarrow \underbrace{B_1 B_2 B_3}_{\text{printf("after B1\n"); x = 0; }} \{ \}$

- Left-recursion more efficient than right-recursion:

- $A : A x | \dots$ rather than $A : x A | \dots$

The Position of Rules

```
$$      |   2   3  
expr : expr '+' term    { $$ = $1 + $3; }  
      | term          { $$ = $1; }  
  
;  
      |   2   3  
term : term '*' factor { $$ = $1 * $3; }  
      | factor         { $$ = $1; }  
  
;  
factor : '(' expr ')' { $$ = $2; }  
      | ID  
      | NUM  
;
```

E → E + T | T
T → T * F | F
F → (E) | id | num

Conflicts

- A conflict occurs when the parser has multiple possible actions in some state for a given next token.
- Two kinds of conflicts:
 - *shift-reduce conflict*:
 - The parser can either keep reading more of the input (“shift action”), or it can mimic a derivation step using the input it has read already (“reduce action”).
 - *reduce-reduce conflict*:
 - There is more than one production that can be used for mimicking a derivation step at that point.

Conflicts

- Two types:
 - shift-reduce [default action: *shift*]
 - reduce-reduce [default: *reduce with the first rule listed*]
- Removing conflicts:
 - specify operator precedence, associativity;
 - restructure the grammar
 - use **y.output** to identify reasons for the conflict.

Handling Conflicts

General approach:

- Iterate as necessary:
 1. Use “yacc -v” to generate the file **y.output**.
 2. Examine **y.output** to find parser states with conflicts.
 3. For each such state, examine the items to figure why the conflict is occurring.
 4. Transform the grammar to eliminate the conflict

Handling Conflicts

Reason for conflict	Possible grammar transformation
Ambiguity with operators in expressions	Specify associativity, precedence
Error action	Remove or eliminate offending error action
Semantic action	Remove the offending semantic action
Insufficient lookahead	Expand the nonterminal involved
Other	No idea

Specifying Operator Properties

- Binary operators: **%left, %right, %nonassoc:**

```
%left '+' '-'
```

```
%left '*' '/'
```

```
%right '^'
```

Operators in the same group have
the same precedence



highest precedence

- Unary operators: **%prec**

- Changes the precedence of a rule to be that of the token specified.

E.g.:

```
%left '+' '-'
```

```
%left '*' '/'
```

Expr: expr '+' expr

```
    | '-' expr %prec '*'
```

```
    | ...
```

Error Handling

- The “token” ‘error’ is reserved for error handling:
 - can be used in rules;
 - suggests places where errors might be detected and recovery can occur.

Example:

```
stmt : IF '(' expr ')' stmt  
      | IF '(' error ')' stmt  
      | FOR ...  
      | ...
```

% token ID **error** ✗

Parser Behavior on Errors

- When an error occurs, the parser:
 - pops its stack until it enters a state where the token ‘error’ is legal;
 - then behaves as if it saw the token ‘error’
 - performs the action encountered;
 - resets the lookahead token to the token that caused the error.
 - If no ‘error’ rules specified, processing halts.

Controlling Error Behavior

- Parser remains in error state until three tokens are correctly read in and shifted
 - prevents cascaded error messages;
 - if an error is detected while parser in error state:
 - no error message is given;
 - input token causing the error is deleted.
- To force the parser to believe that an error has been fully recovered from:
yyerrok;
- To clear the token that caused the error:
yyclearin;

Placing ‘error’ tokens

- Close to the start symbol of the grammar:
 - To allow recovery without discarding all input.
- Near terminal symbols:
 - To help permit a small amount of input to be discarded due to an error.
 - Tokens like ‘)’, ‘;’, that follow non-terminals.
- Without introducing conflicts

Error Messages

- On finding an error, the parser calls a function

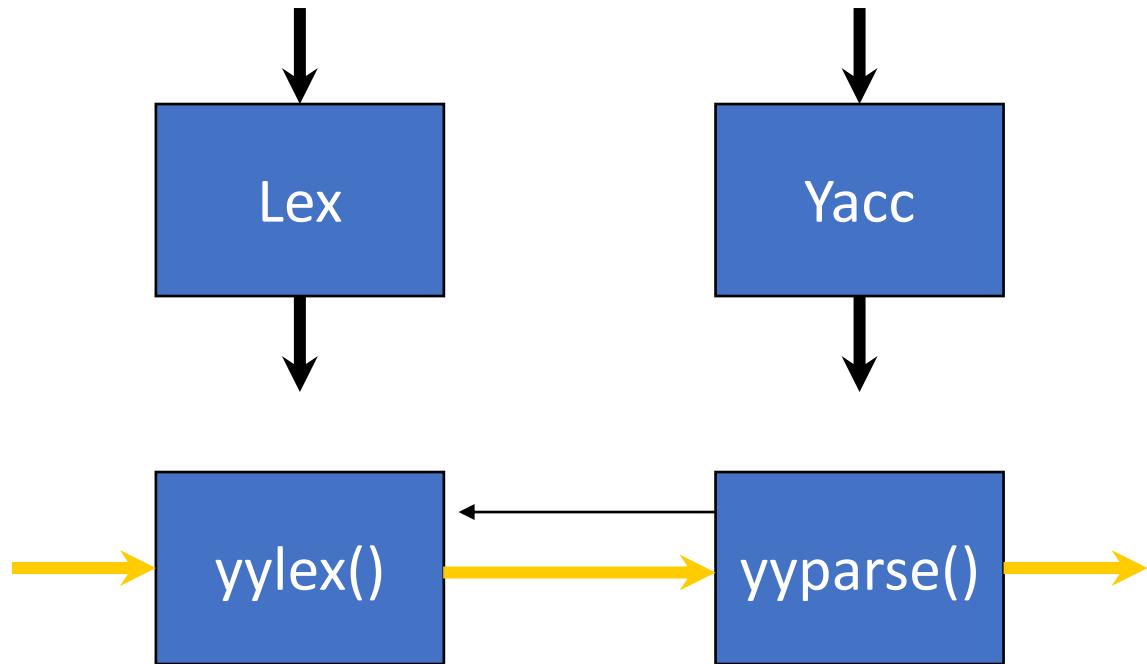
```
void yyerror(char *s) /* s points to an error msg */
```

- user-supplied, prints out error message.

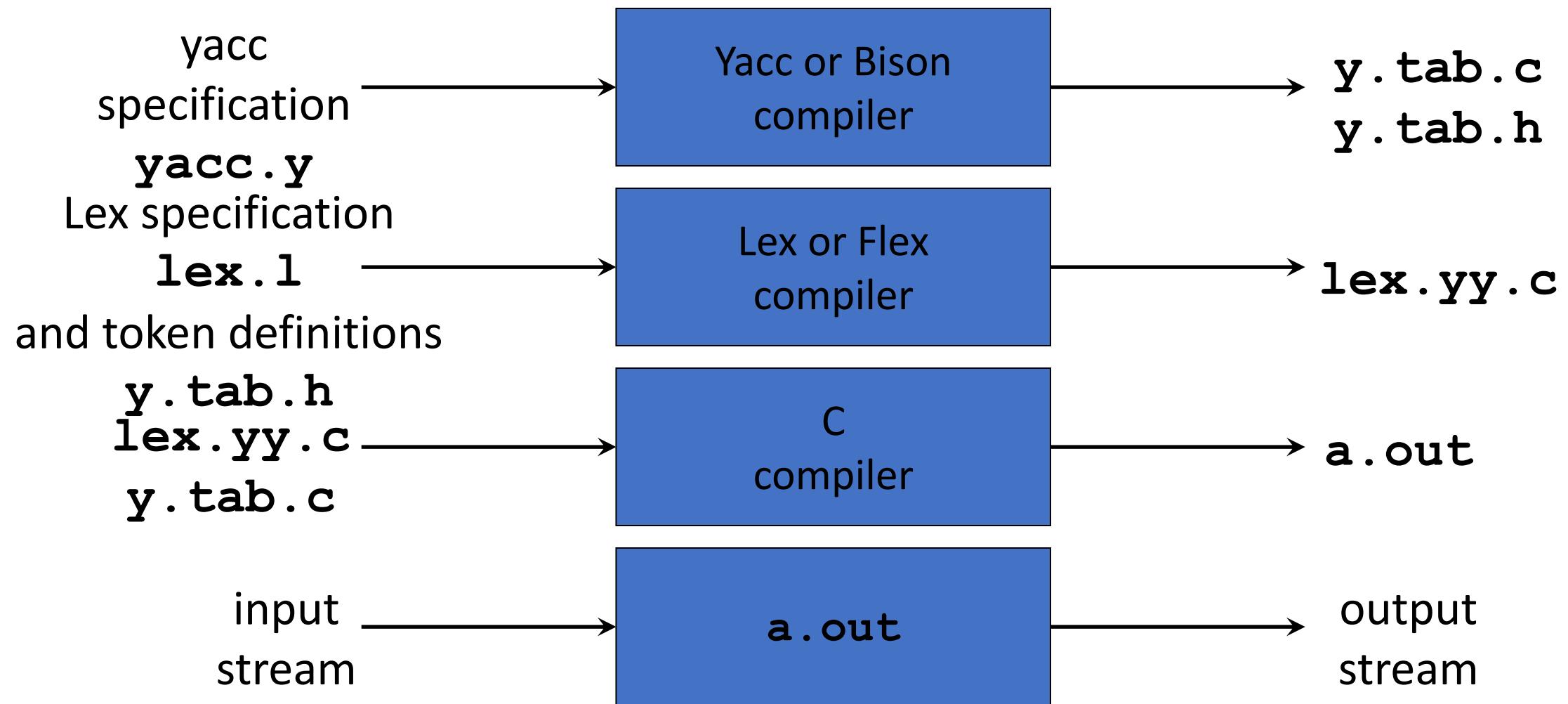
- More informative error messages:

- int yychar: token no. of token causing the error.
- user program keeps track of line numbers, as well as any additional info desired.

Lex with Yacc



Combining Lex/Flex with Yacc/Bison



Summary

- Features of YACC and the need for YACC to perform parsing