Lecture 2
Syntax Analysis

Dr. Wilson Rivera

ICOM 4036: Programming Languages

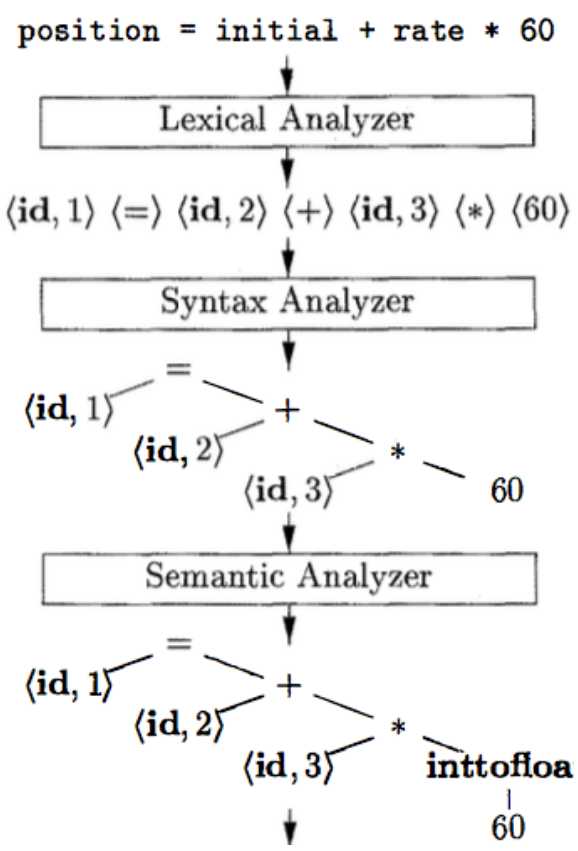
Electrical and Computer Engineering Department
University of Puerto Rico

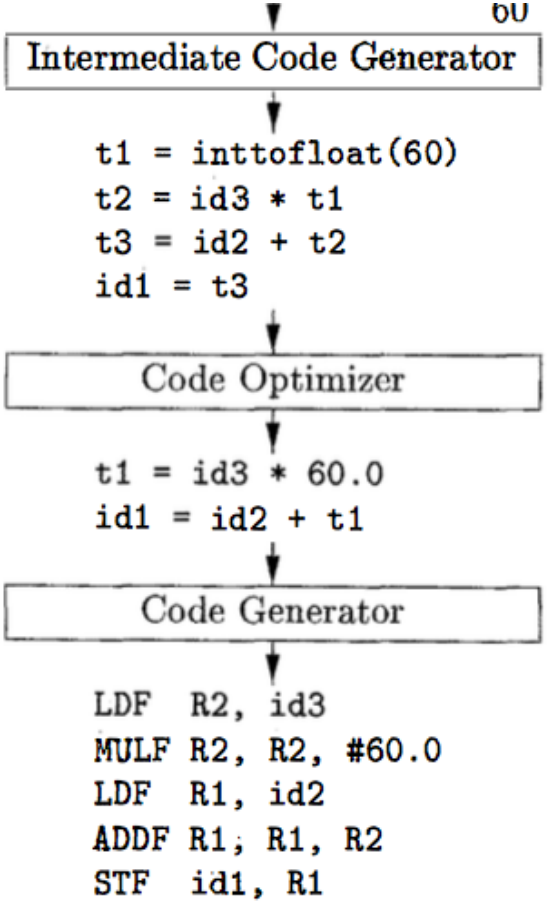
Some slides adapted from Sebesta's textbook

Lecture Outline

- Introduction to syntax analysis
- Regular expressions
- Finite Automata
- Context free grammars
- Lexical Analysis
- Parsing

Introduction



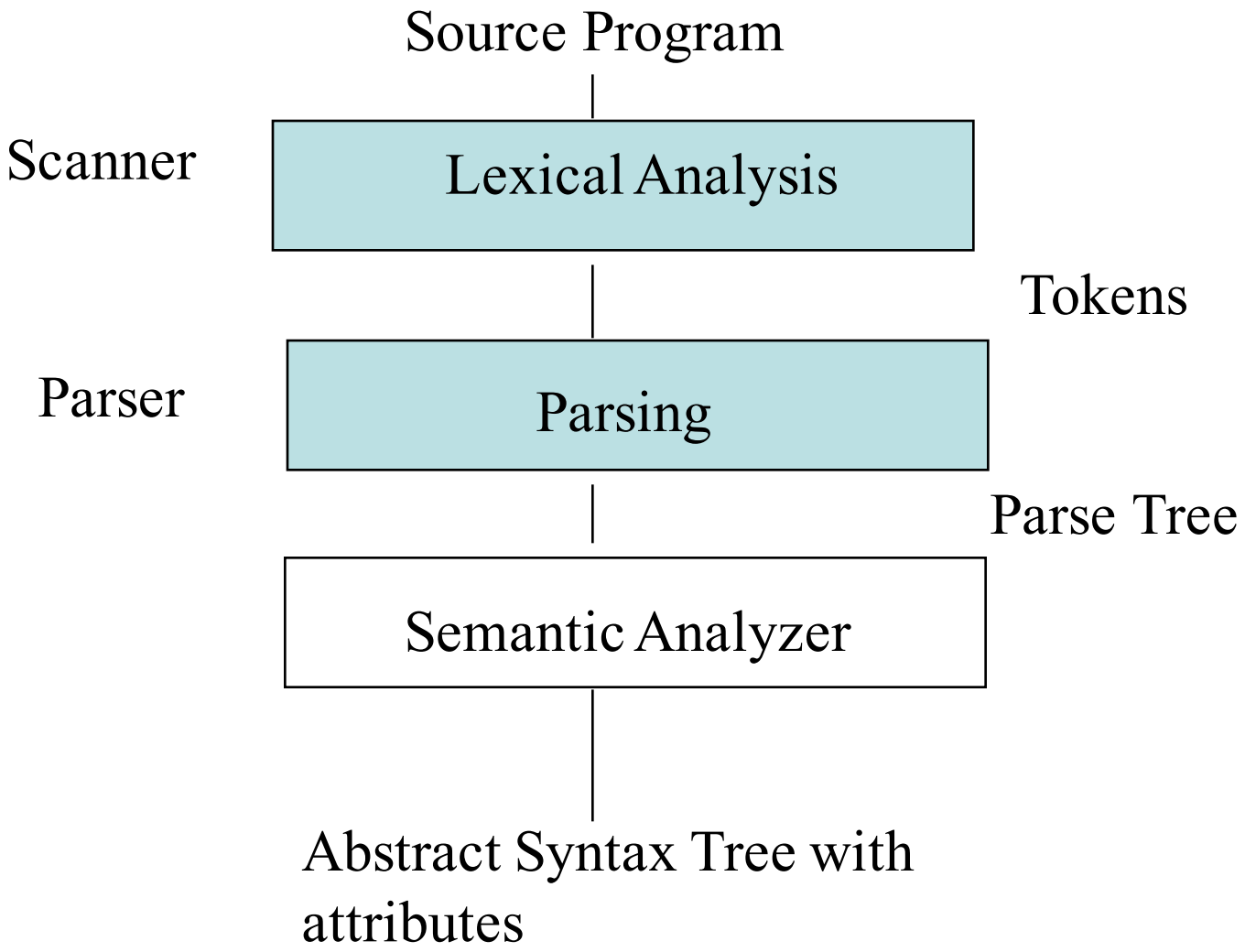


Introduction

- **Syntax**: the form or structure of the expressions, statements, and program units
- **Semantics**: the meaning of the expressions, statements, and program units

- Syntax and semantics provide a language's definition
  - Syntactic specification of a programming language
  - Construction of efficient parsers
  - Error detection
  - Extension language and new constructs

Introduction



Regular Expressions

Regular Expressions: A concise notation for regular sets

(1) Φ denotes the regular set Φ.

(2) Λ denotes the regular set $\{\Lambda\}$.

(3) α denotes the regular set $\{\alpha\}$.

(4) If p and q are regular expressions denoting the regular sets P and Q respectively, then

- (a) (p | q) denotes $P \cup Q$ (Union)
- (b) (pq) denotes P Q (Concatenation)
- (c) (p)* denotes P* (Kleene Closure)

(5) Nothing else is a regular expression.

Notation:

p+ = p* p (non reflexive closure)

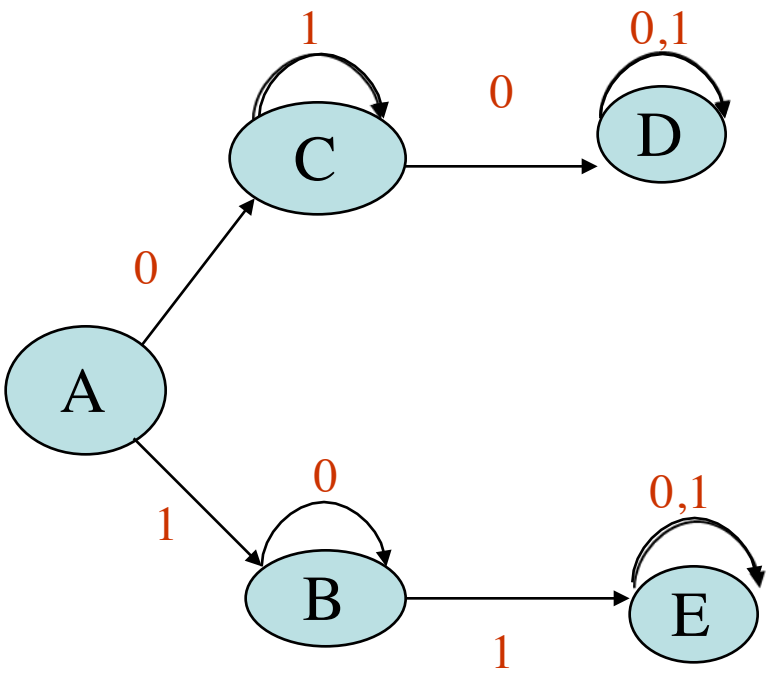
Regular Expressions

RE	Regular language Description
a|b	{a,b}
(a|b)(a|b)	{aa, ab, ba, bb}
a*	{e, a, aa, aaa, …}
a*b	{b,ab, aab, aaab, …}
(a|b)*	{e, a, aa, aaa, …, b, bb, bbb… ab, abb,…. aab, aabb, ….}

Regular Expressions

- Regular Expression

  (1O*1(0|1)*) | (01*0(0|1)*)



Finite Automata

Deterministic Finite Automaton (DFA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

(1) Q is a finite non-empty set of states

(2) Σ is a finite set of input symbols

(3) $q_0 \in Q$ (initial state)

(4) $F \in Q$ (final states)

(5) δ is a partial mapping from $Q \times \Sigma$ to Q

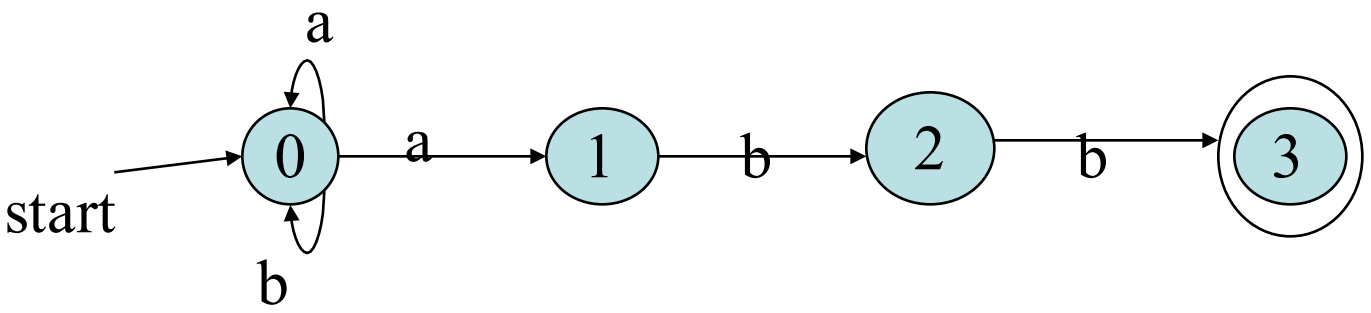
(transition function: move function)

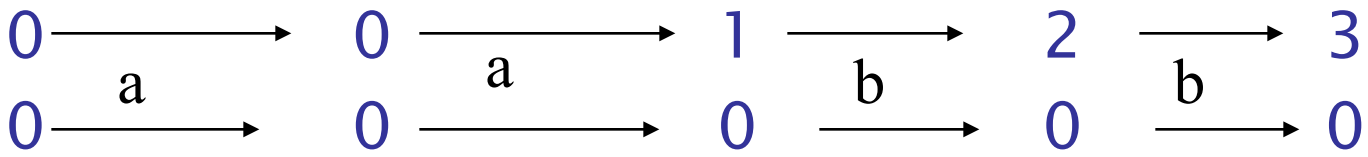
Non-deterministic Finite Automaton (NFA)

- May have a choice of moves, i.e. δ is a mapping from $Q \times \Sigma$ to 2^Q
- Also allows $\in$-transitions, i.e., $\delta(q, \in) \subseteq Q$

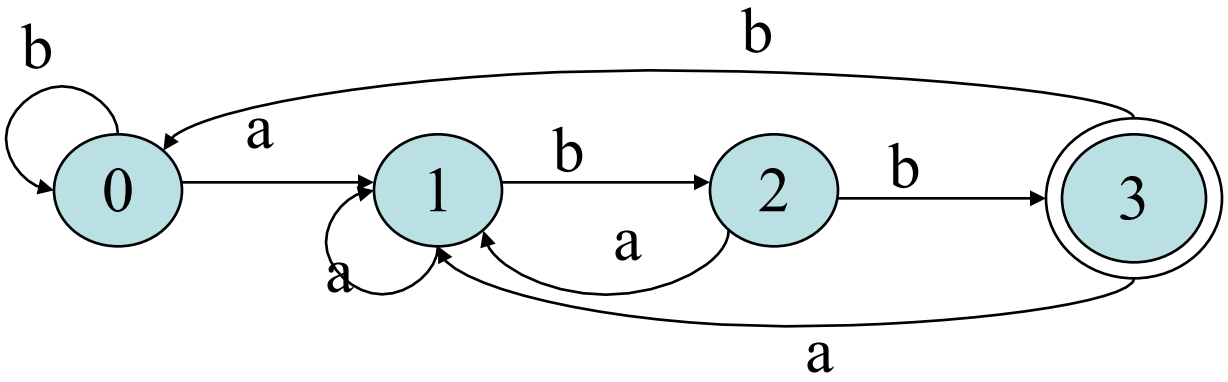
NFA Example

(a|b)*abb = { abb, aabb, babb, aaabb, bbabb,…}





DFA Example



DFA:

1. No state has an e-transition
2. For each state ***S*** and input symbol ***a***, there is at most one edge labeled ***a*** leaving ***S***.

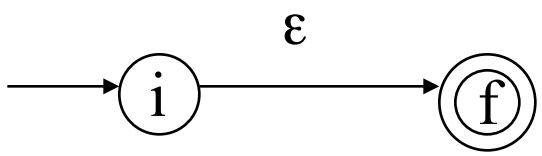
Finite Automata Construction

- For every NFA M_1 there is a DFA M_2 for which $L(M_2) = L(M_1)$
- *Thompson's Construction:*
  - Systematically generate an NFA for a given Regular Expression.
- *Subset construction algorithm*:
  - Converts an NFA to an equivalent DFA
  - *Key:* identify sets of states of NFA that have similar behavior, and make each set a single state of the DFA.

From a Regular Expression to an NFA

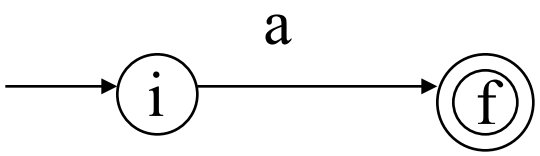
Thomson's Construction

1



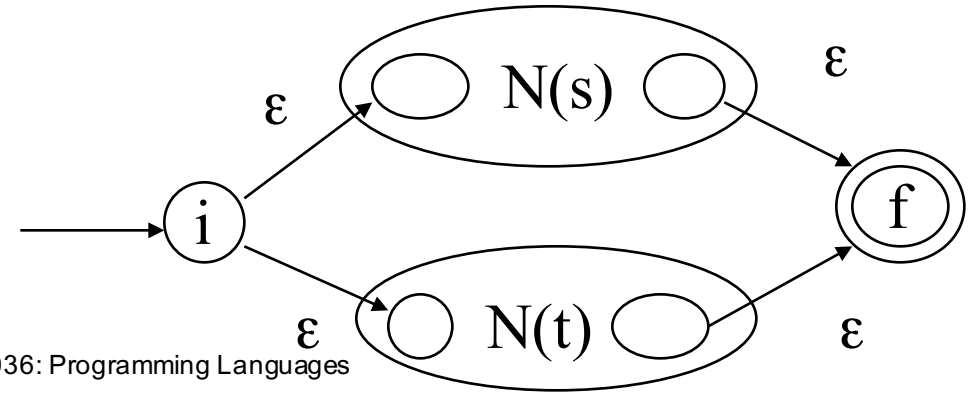
N(ε)

2



N(a), a∈Σ

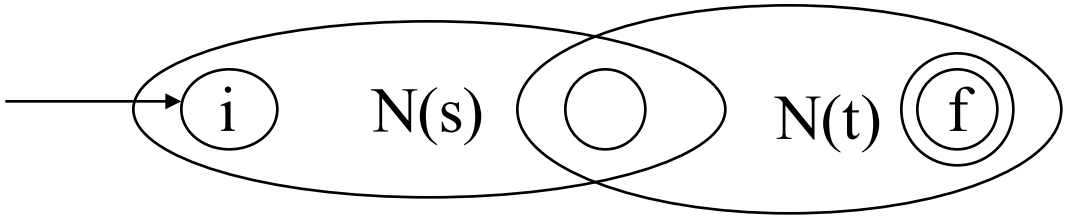
3



N(s|t)

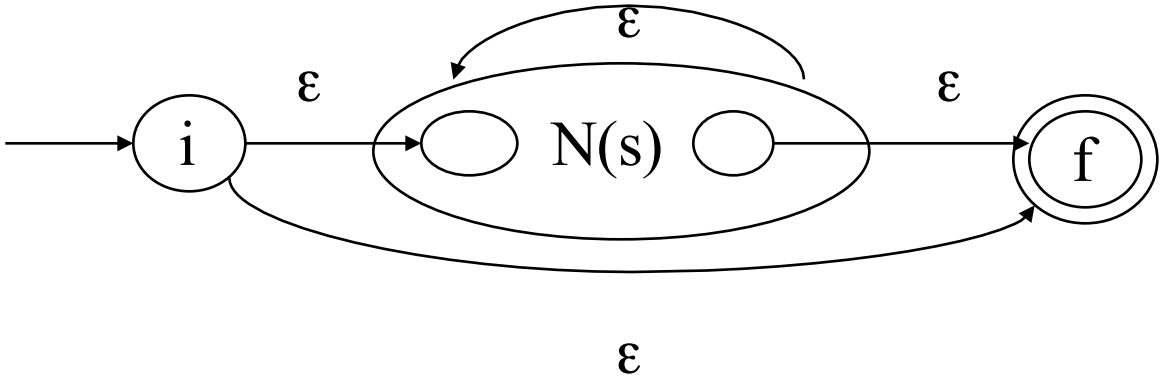
From a Regular Expression to an NFA

4



N(st)

5

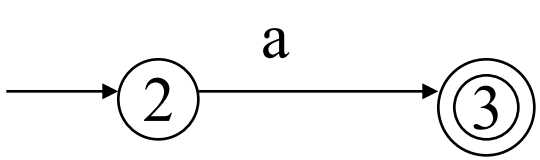


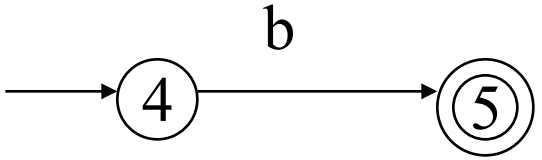
N(s*)

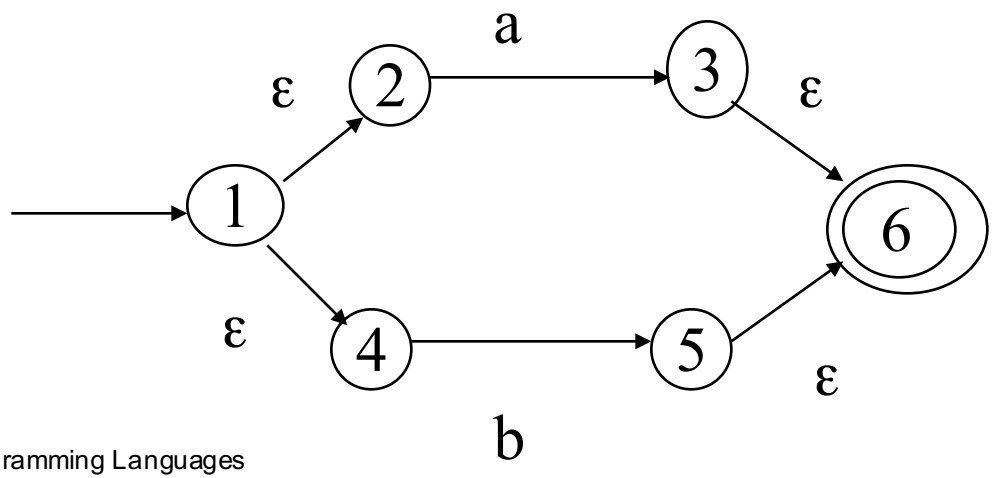
6 Combine single NFAs for complex structures

From a Regular Expression to an NFA

Example : (a|b)*abb

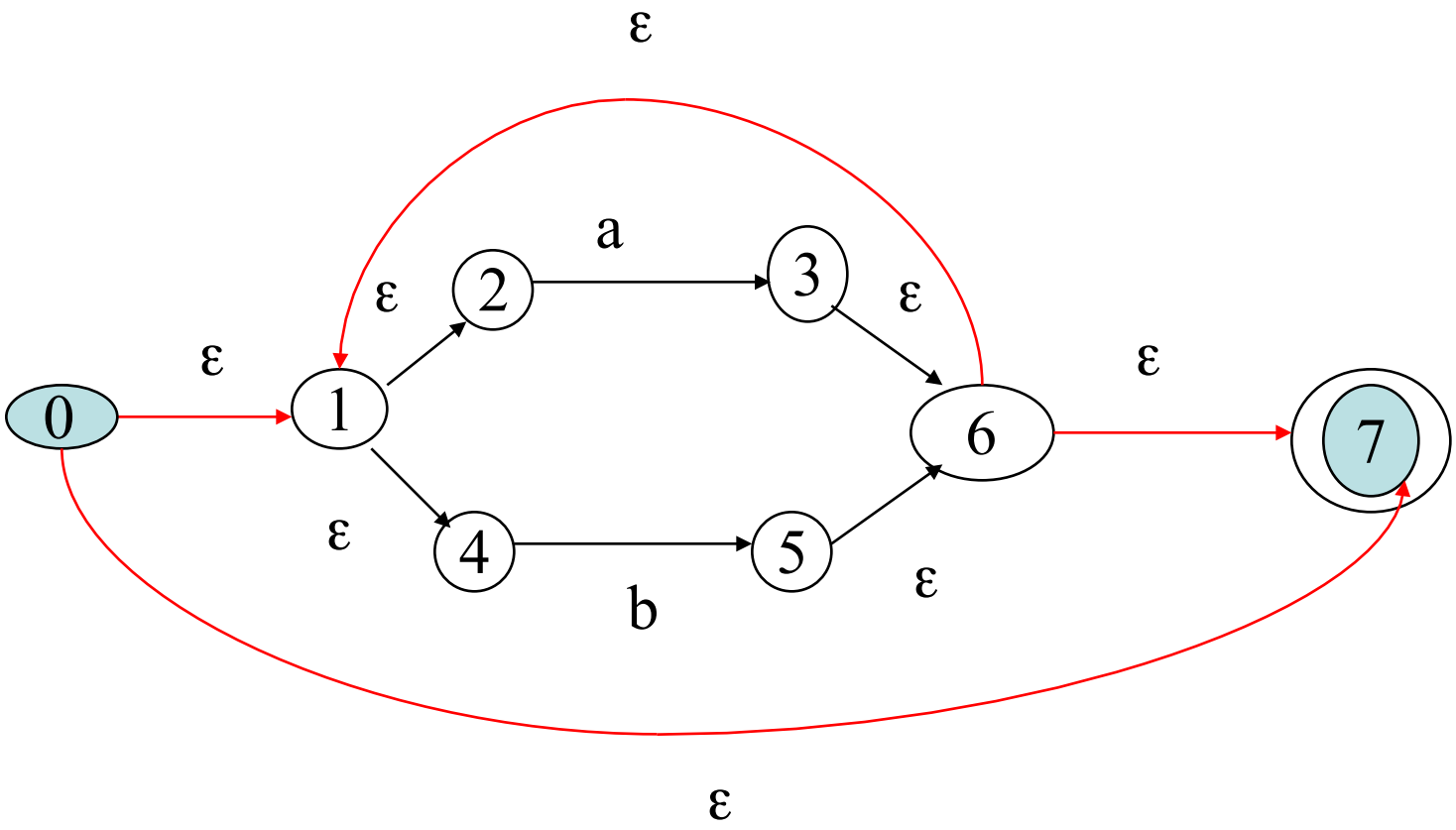






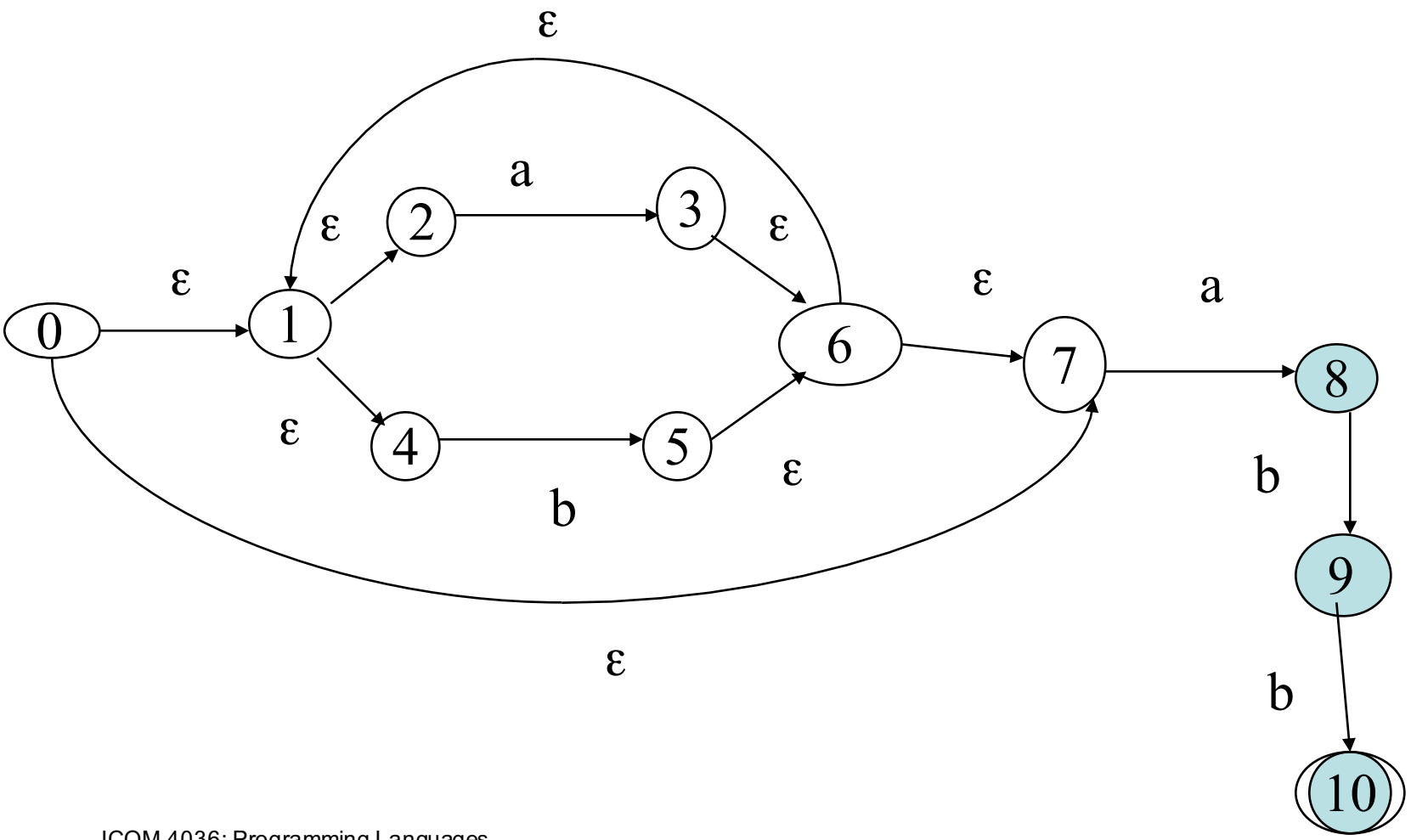
From a Regular Expression to an NFA

Example : (a|b)*abb



From a Regular Expression to an NFA

Example : (a|b)*abb



From an NFA to a DFA

Subset Construction

Operation	Description
ε-closure(s)	Set of NFA states reachable from an NFA state s on e-transitions along
ε-closure(T)	Set of NFA states reachable from some NFA state s in T on e-transitions along
Move(T,a)	Set of NFA states reachable from some NFA state set with a transition on input symbol a

From an NFA to a DFA

Subset Construction

Initially, e-closure (s_0) is the only states in D and it is unmarked

```
while there is an unmarked state T in D do
    mark T;
    for each input symbol a do
        U:= e-closure(move(T,a));
        if U is not in D then
            add U as an unmarked state to D
        Dtran[T,a]:=U;
    end(for)
end(while)
```

From an NFA to a DFA

Example

- e-closure(0)={0,1,2,4,7}=A

- e-closure(move(A,a))=
  e-closure({3,8})={1,2,3,4,6,7,8}=B
  Thus, Dtran[A,a]=B

- e-closure(move(A,b))=
  e-closure({5})={1,2,4,5,6,7}=C
  Thus, Dtran[A,b]=C

From an NFA to a DFA

- e-closure(move(B,a))=
  e-closure({3,8})=B
  Thus, Dtran[B,a]=B

- e-closure(move(B,b))=
  e-closure({5,9})={1,2,4,5,6,7,9}=D
  Thus, Dtran[B,b]=D

From an NFA to a DFA

- e-closure(move(C,a))=
  e-closure({3,8})=B
  Thus, Dtran[C,a]=B

- e-closure(move(C,b))=
  e-closure({5})=C
  Thus, Dtran[C,b]=C

From an NFA to a DFA

- e-closure(move(D,a))=
  e-closure({3,8})=B
  Thus, Dtran[D,a]=B

- e-closure(move(D,b))=
  e-closure({5,10})={1,2,4,5,6,7,10}=E
  Thus, Dtran[D,b]=E

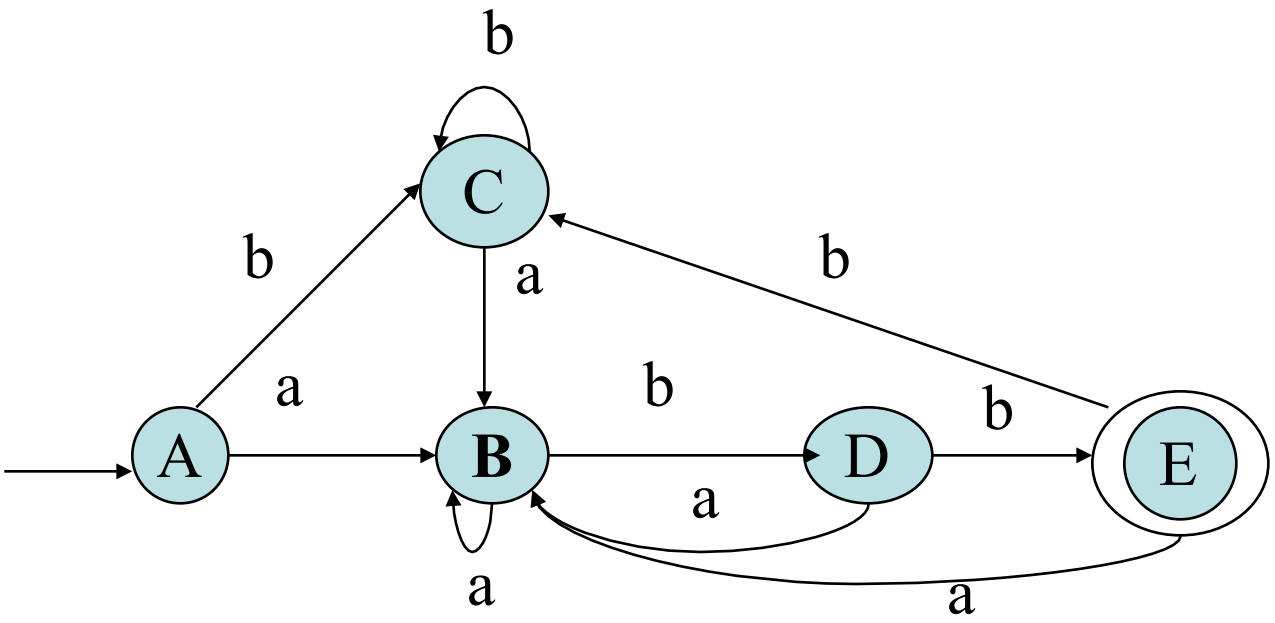
From an NFA to a DFA

- e-closure(move(E,a))=
  e-closure({3,8})=B
  Thus, Dtran[E,a]=B

- e-closure(move(E,b))=
  e-closure({5})=C
  Thus, Dtran[E,b]=C

From an NFA to a DFA

states	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



From a DFA to a minimal DFA

Hopcroft Algorithm

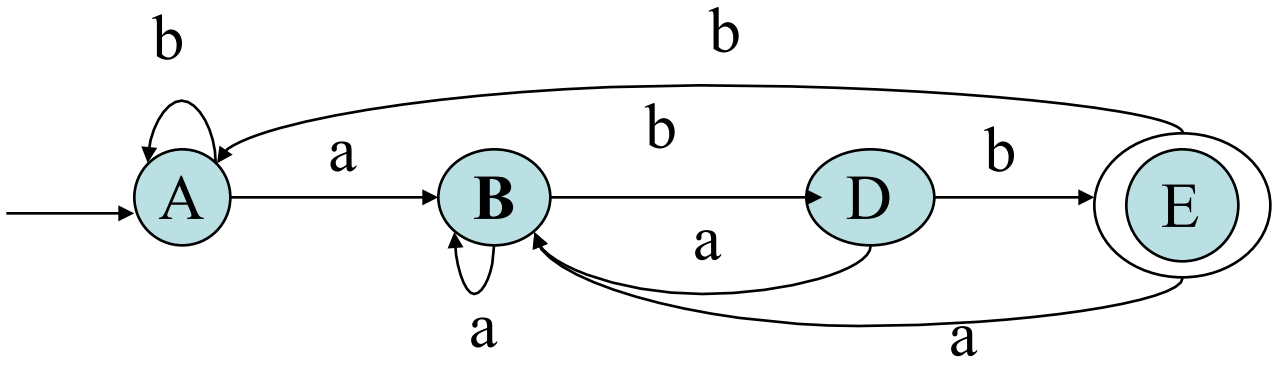
1. Construct an initial partition P of set of states with two groups:
   The accepting states F and the nonaccepting states S-F
2. Apply the following procedure to P to construct a new partition P_{new}
   for each group G of P do
   divide G into subgroups such that two states
   of G are in the same subgroup if and only if for all input symbol a, states s and t have transitions on a to states in the same group of P.
   Replace G in P_{new} by the set of all subgroup formed.
3. If P_{new} =P let = P_{final} =P and go to 4. Otherwise repeat (2) with P= P_{new}.
4. Choose one state in each group of the partition P_{final} as the representative for that group
5. Contruct the new transition table by replacing the states in a group by the respresentative

From a DFA to a minimal DFA

1. P={ (ABCD), (E)}
2. P_new={(ABC), (D), (E)} D→E
3. P_new={(AC), (B), (D), (E)} B→D
4. P_final={(AC), (B), (D), (E)}
5. We choose A as the representative for the subgroup (AC)

From an NFA to a DFA

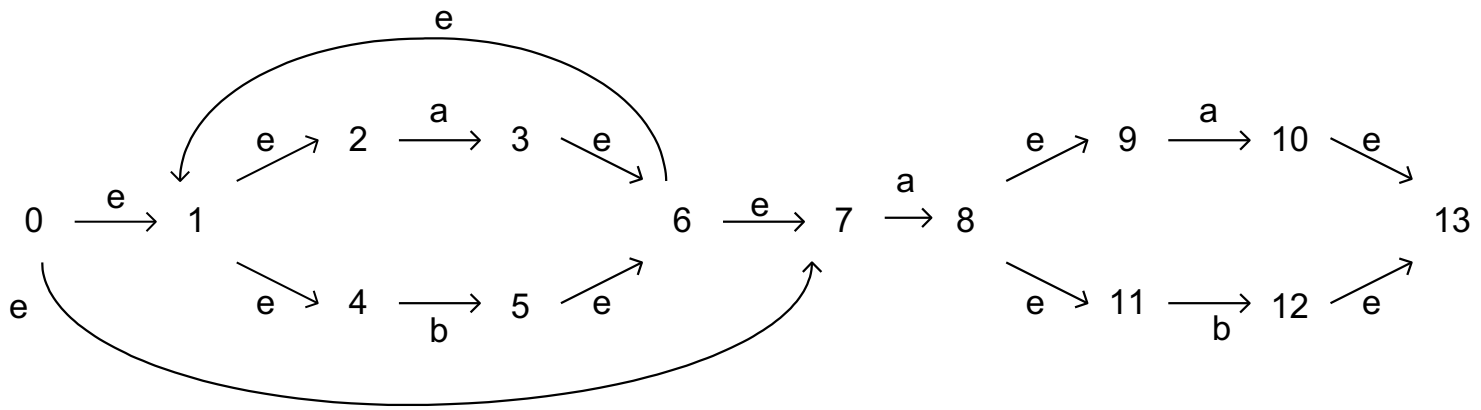
states	a	b
A	B	A
B	B	D
D	B	E
E	B	A



RE to NFA: Example

- Desing a NFA for the following regular expression

(a|b)*a(a|b)



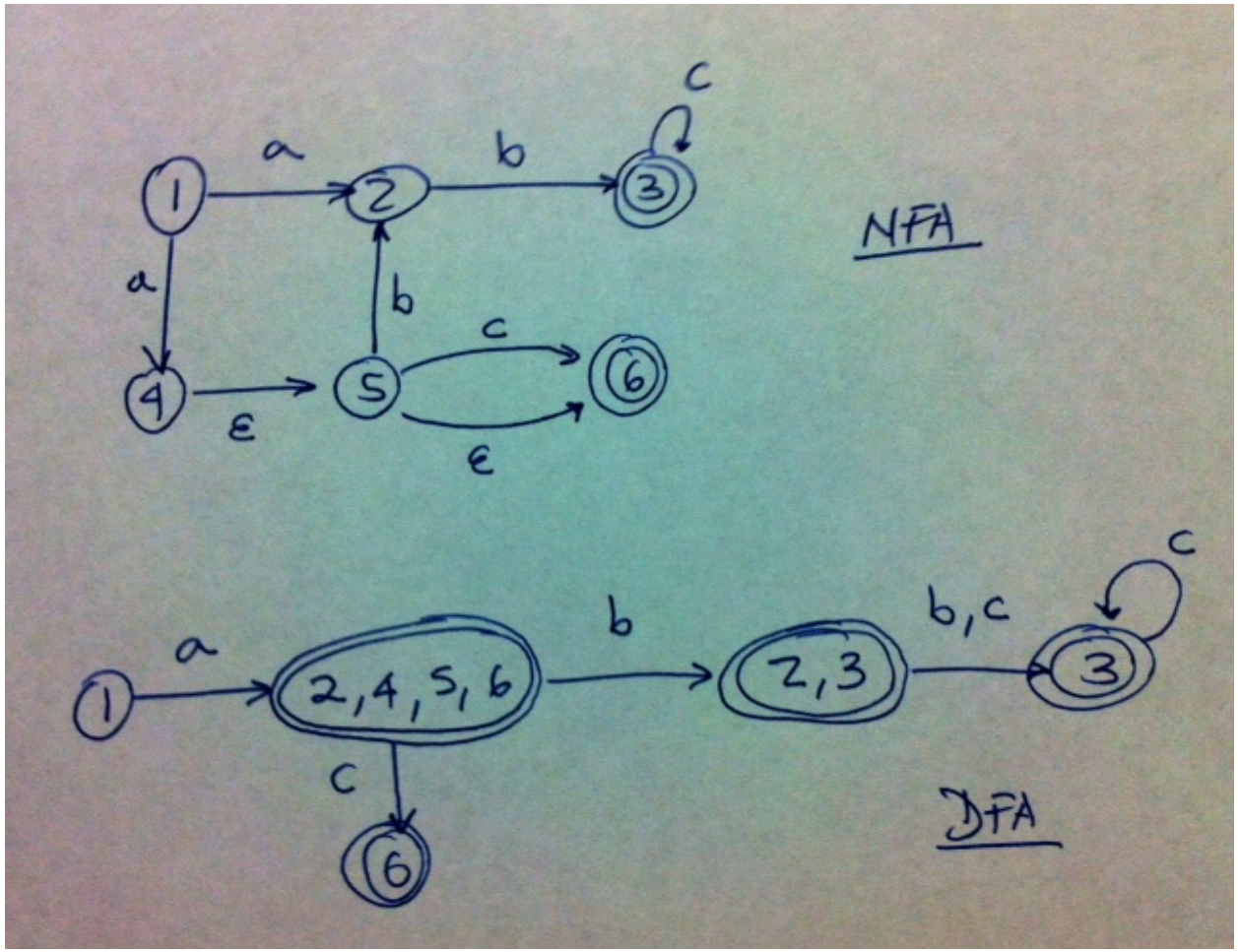
NFA to DFA: Example

- Reduce the NFA to a DFA

- E(0)={0 1 2 4 7}=A
- E(move A a) =E(3 8)= {1 2 3 4 6 7 8 9 11}=B
- E(move A b)=E(5)={1 2 4 5 6 7}=C
- E(move B a)=E(3 8 10)={1 2 3 4 6 7 8 9 10 13} =D
- E(move B b)=E(5 12)={1 2 4 5 6 7 12 13}=E
- E(move C a)=E (3 8) = B
- E(move C b)= E (5) =C

  E(move D a)= E(3 8 10) =D
- E(move D b)= E(5)= C
- E(mov E a)=E(3 8)=B
- E(mov E b)=E(5)=C

NFA to DFA



Regular Expressions (summary)

- REs are commonly used to define search patterns and the lexical structure of programming Languages

- REs are a convenient mechanism for describing languages but they do not give a method for recognizing tokens
  - Construct a NFA for the regular expression.
  - Convert the NFA to an equivalent DFA.
  - Minimize the number of states in the DFA

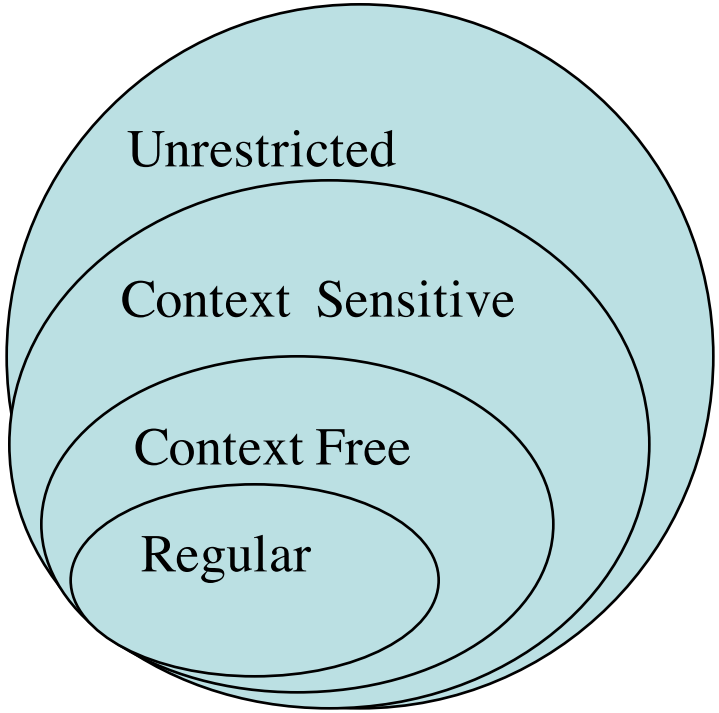
More on Regular Expressions

- $\{a^n b^n : n>0\}$ is not regular since the number of a's controls the number of b's
  - This type of operation is not allowed according to the definition of regular expressions

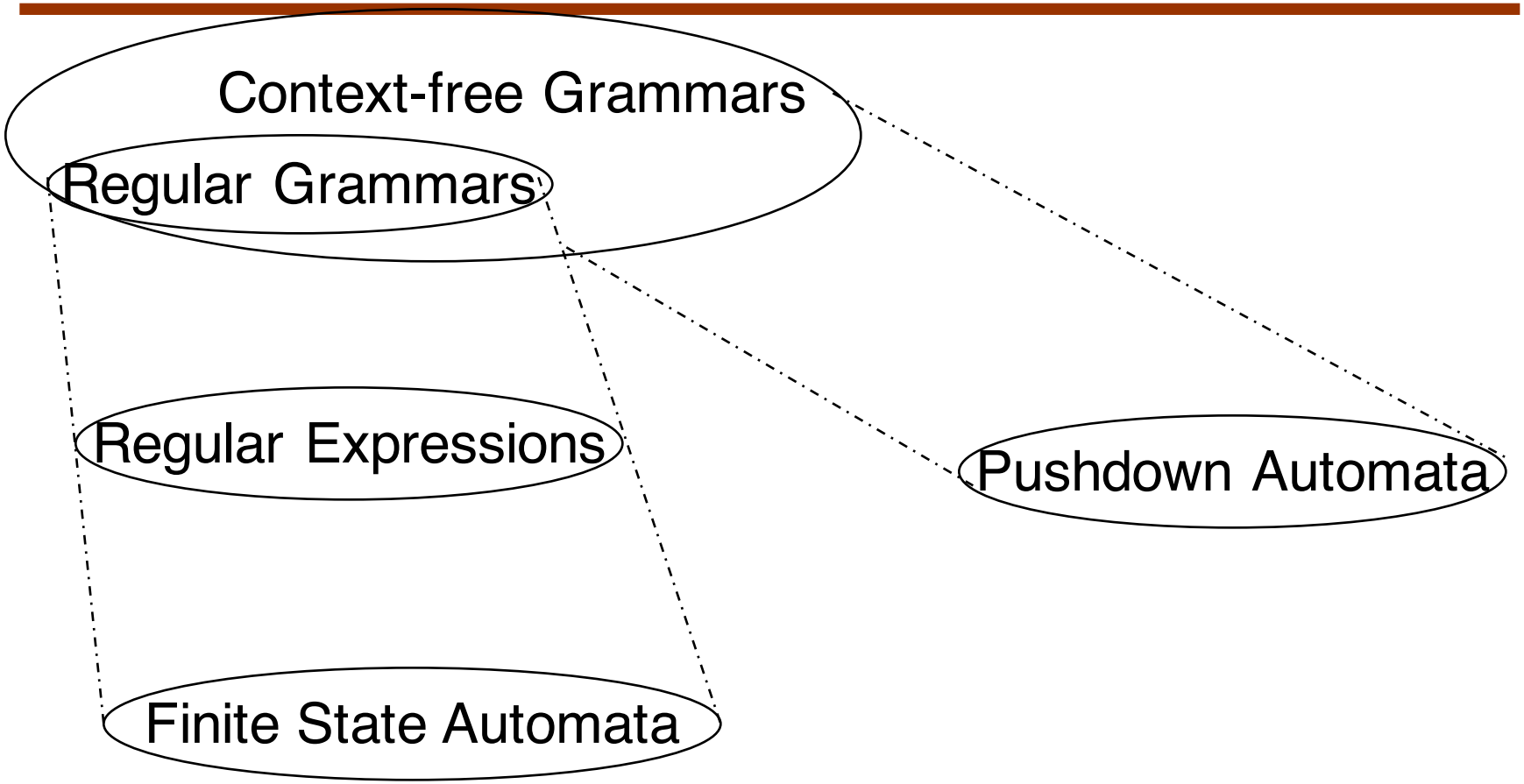
- Many real world REs engines implement features that cannot be described by REs theory
  - The Python regular expression r"(a*)b\1" recognizes $a^n b a^n$
    - \1 matches the same string matched by the parenthesized sub-expression.

Chomsky Hierarchy

Type-0	unrestricted	Turing machine (TM)
Type-1	Context sensitive	Linear bounded TM
Type-2	Context free	Non deterministic pushdown automaton
Type-3	Regular	Finite State Automaton



Chomsky Hierarchy



For Regular Grammars there is always a corresponding deterministic automaton. For Context-free grammars there is none unless the grammar is unambiguous.

Context-free Grammar

A formal grammar is define as

G=(T,N,S,P)

1) *Tokens* (terminals)
2) *Constructs* (nonterminals)
3) *Starting nonterminal* (main construct)
4) *Productions* (rules that define nonterminal in terms of sequences of terminals and nonterminals)

A context-free grammar (CFG) is a formal grammar in which every production rule is of the form V → w where

- V is a single nonterminal symbol
- w is a string of terminals and/or non-terminals (w can be empty).

The language of a grammar G=(T;N;R; S) usually denoted by L(G), is defined as L(G)={s | S-> * s, s in T*}

BNF and Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Meant to describe the syntax of natural languages
  - Context free means replacing non-terminals in any order (i.e., regardless of context) produces same result (as long as you use same productions).
  - If we use the full power of the context-free languages we get compilers which in general are inefficient, and probably not so good in handling erroneous input

- Backus-Naur Form (1959)
  - Invented by John Backus to describe Algol 58
  - BNF is equivalent to context-free grammars

BNF Fundamentals

- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals
- *Terminals* are lexemes
- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

    `<ident_list>` → `identifier | identifier, <ident_list>`

    `<if_stmt>` → **`if`** `<logic_expr>` **`then`** `<stmt>`
- Grammar: a finite non-empty set of rules
- A *start symbol* is a special element of the nonterminals of a grammar

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

Describing Lists

- Syntactic lists are described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

Example: Grammar sentences

- Consider the following grammar

  <s> → <A> a <B> b

  <A> → <A>b | b

  <B> → a<B> | a

Which of the following sentences are in the language generated by this grammar?

1. baab
2. bbbab
3. bbaaaa
4. bbaab

Example: Regular Grammar

- Regular Grammar

  <S>→ a<S>

  <S>→b<A>

  <A>→e | c<A>

- Regular expression

  a*bc*

Non Regular Grammar

- Context free grammar

  <S>→ a<S>b | ab

- $\{a^n b^n : n>0\}$ is not regular since the number of a's controls the number of b's
  - This type of operation is not allowed according to the definition of regular expressions

Context free grammars

- If we remove the regular-grammar restrictions, we get a class of grammars known as the Type 2 or context-free grammars. Such grammars can "count" arbitrarily high, at least under the right circumstances.
  - For example, here is a language of correctly nested parentheses
    - S → S '(' S ')' | e
    - It recognizes the empty string, '()', '()()', '(())', '((()))'

Example: Grammar Derivations

```
<program> → <stmts>
 <stmts> → <stmt> | <stmt> ; <stmts>
 <stmt> → <var> = <expr>
 <var> → a | b | c | d
 <expr> → <term> + <term> | <term> - <term>
 <term> → <var> | const
```

Example: Grammar Derivation

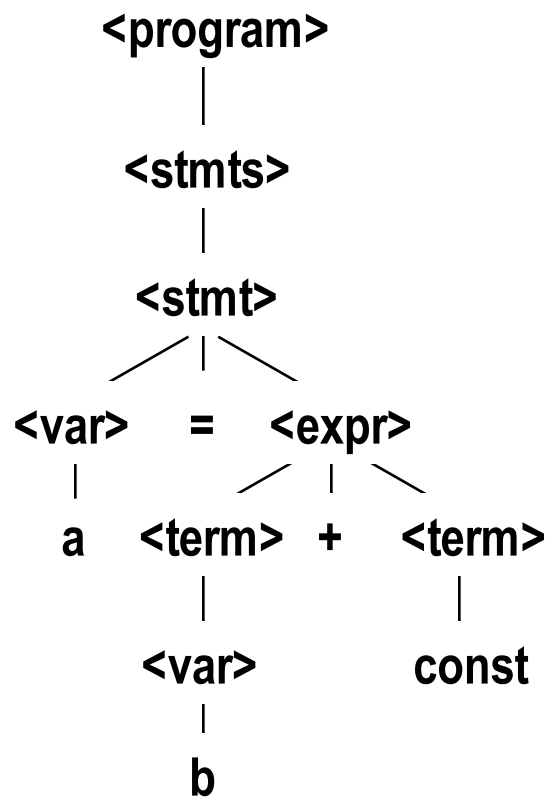
```
<program> => <stmts> => <stmt>
                     => <var> = <expr>
                     => a = <expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```

Derivations

- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* always choose leftmost non-terminals to be expanded.

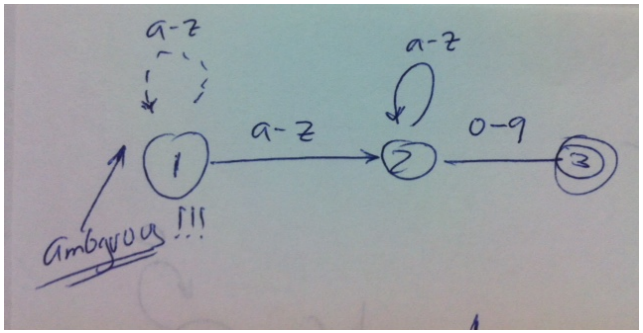
Parse Tree

- A hierarchical representation of a derivation



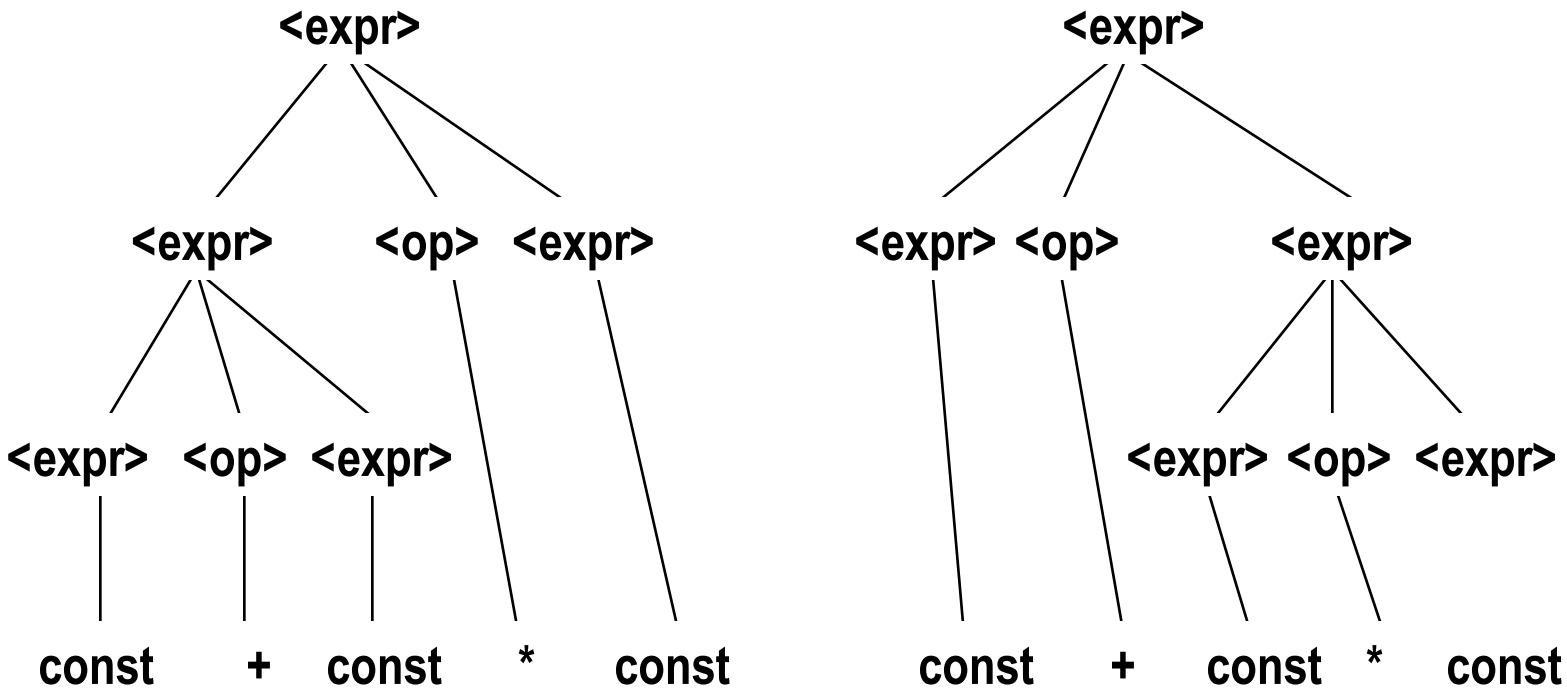
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees
- Checking a grammar is ambiguous is undecidable
  - Reduction from Post Correspondence problem
- Ambiguous grammars lead to ambiguous semantics!



An Ambiguous Expression Grammar

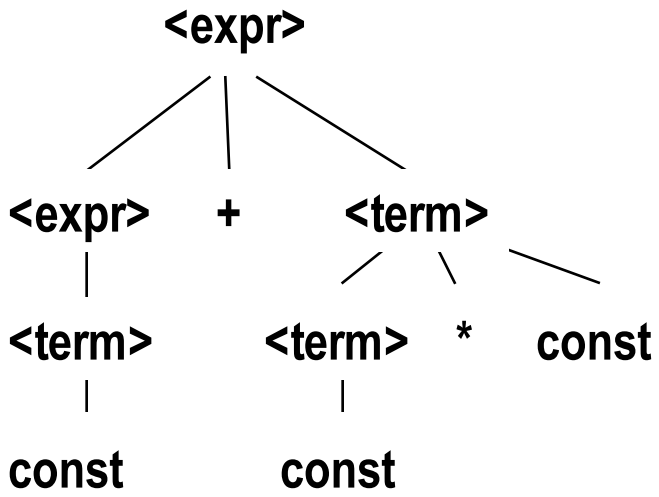
```
<expr> → <expr> <op> <expr>  |  const
<op> → *  |  +
```



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

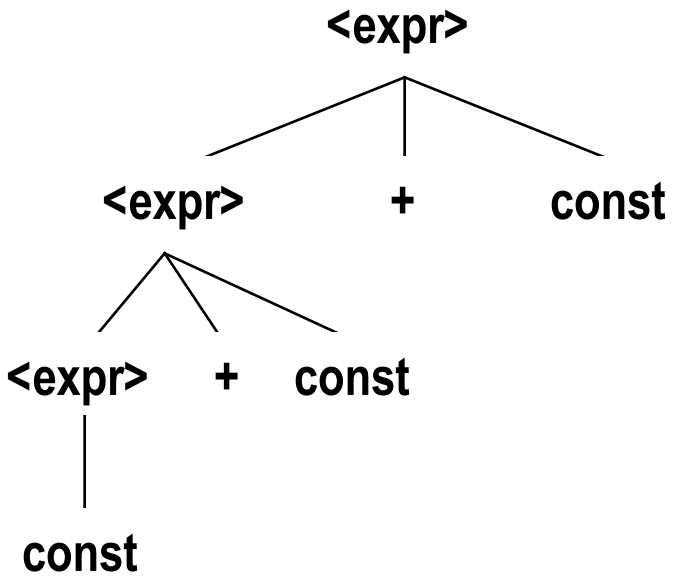
```
<expr> → <expr> + <term>  |  <term>
<term> → <term> * const| const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Extended BNF

- Optional parts are placed in brackets [ ]

  `<proc_call> -> ident [(<expr_list>)]`

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

  `<term> → <term> (+|-) const`

- Repetitions (0 or more) are placed inside braces { }

  `<ident> → letter {letter|digit}`

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → {<expr>(+ | -)} <term>
<term> → {<term>(* | /)}<factor>
```

Example: Grammar Definition

Function: type IDENTIFIER LEFT_PARENT optional_arguments RIGHT_PAREN
LEFT_PAREN LEFT_CURLY statements RIGHT_CURLY

Type: TYPE_KEYWORD | user_defined_type

Optional-arguments: null | argument | argument COMMA arguments

Statements: statement | statement statements

Statement: declaration | equals_expr | …..

Declaration: type IDENTIFIER optional_declarations SEMICOLON

equals_expr: IDENTIFIER EQUAL_SIGN expression

Expression: plus_expr | equals_expr | …

plus_expression: expression PLUS_SIGN expression

Example: Java language specification

PrimitiveType:
    NumericType
    boolean

NumericType:
    IntegralType
    FloatingPointType

IntegralType: one of
    byte short int long char

FloatingPointType: one of
    float double

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *parser* (mathematically, a push-down automaton based on a context-free grammar)

Lexical Analysis

- Breaks the text down into the smallest useful atomic units, known as tokens
  - while throwing away (or at least, putting to one side) information, such as white space and comments
- Groups characters into tokens to facilitate the parsing process
  - X+Y *
  - X+Y++

Lexical Analysis

- Approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description
  - Design a state diagram that describes the tokens and write a program that implements the state diagram

Example: Lexical Analysis

```
int main ()
{
   float foo, bar, baz;
   foo = bar + baz;
}
```

Lexeme	Token Category
'int'	TYPE_KEYWORD
'main'	IDENTIFIER
'('	LEFT_PAREN
')'	RIGHT_PAREN
'{'	LEFT_CURLY
'float'	TYPE_KEYWORD
'foo'	IDENTIFIER
','	COMMA
'bar'	IDENTIFIER
','	COMMA
'baz'	IDENTIFIER
';'	SEMICOLON
'foo'	IDENTIFIER
'='	OPERATOR
'bar'	IDENTIFIER
'+'	OPERATOR
'baz'	IDENTIFIER
';'	SEMICOLON
'}'	RIGHT_CURLY

Example: Lexical Analysis

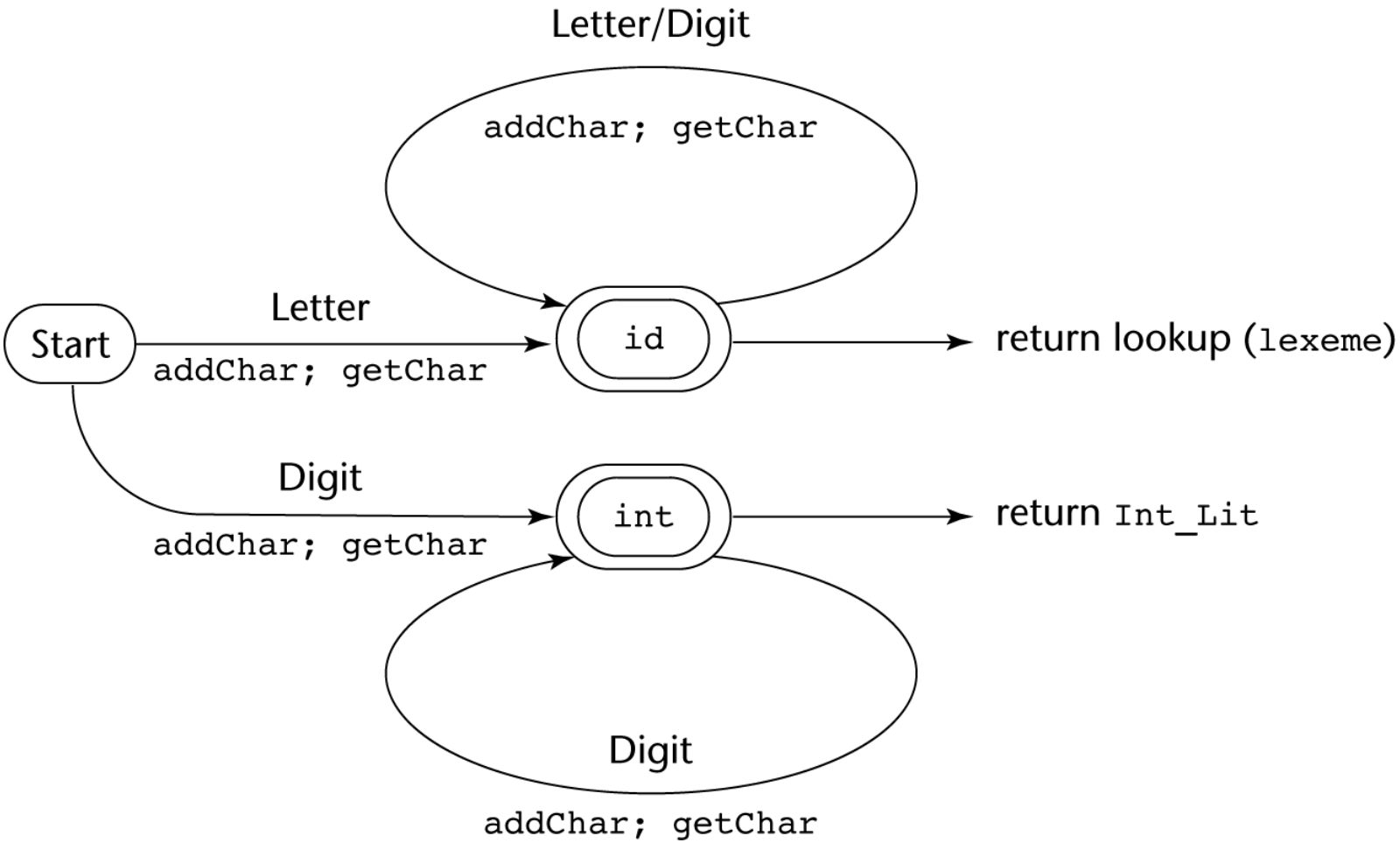
Lexeme	Token	CFG Definition
‘int’	TYPE_KEYWORD	type
‘main’	IDENTIFIER	IDENTIFIER
‘(‘	LEFT_PAREN	LEFT_PAREN
Nothing	nothing	optional_arguements
‘)’	RIGHT_PAREN	RIGH_PAREN
‘{‘	LEFT_CURLY	LEFT_CURLY
‘float’	TYPE_KEYWORD	Statements
‘foo’	IDENTIFIER	
‘,’	COMMA	
‘bar’	IDENTIFIER	
‘,’	COMMA	
‘baz’	IDENTIFIER	
‘;’	SEMICOLON	
‘foo’	IDENTIFIER	
‘=‘	OPERATOR	
‘bar’	IDENTIFIER	
‘+’	OPERATOR	
‘baz’	IDENTIFIER	
‘;’	SEMICOLON	
‘}’	RIGHT_CURLY	RIGHT_CURLY

Example: Lexical Analysis

```
If (i== j)
  z = 0; /* comment */
else
  z= 1;

\tif(i== j)\n
\t\tz = 0; /* comment*/\n
\telse\n
\t\tz= 1;
```

IF, LPAR, ID("i"), EQUALS, ID("j"), RPAR, ID("z"), ASSIGN,INTLIT("0"), SEMI, ELSE, ID("z"), ASSIGN, INTLIT("1"), SEMI

State Diagram



Implementing the State Diagram

```
int CharClass;
char lexeme [100];
char nextChar;
int lexLen;
int LETTER=0;
int DIGIT=1;
int UNKNOWN=-1;


/*addChar - a function to add nextChar to lexeme */

void addChar(){
   if (lexLen <=99)
      lexeme[lexLen++]=nextChar;
   else printf("Error: Lexeme is too long \n");
}



/* getNonBlank - call getChar until it returns a non-whitespace character */

void getNonBlank(){
   while(isspace(nextChar))
      getChar();
}
```

Implementing the State Diagram

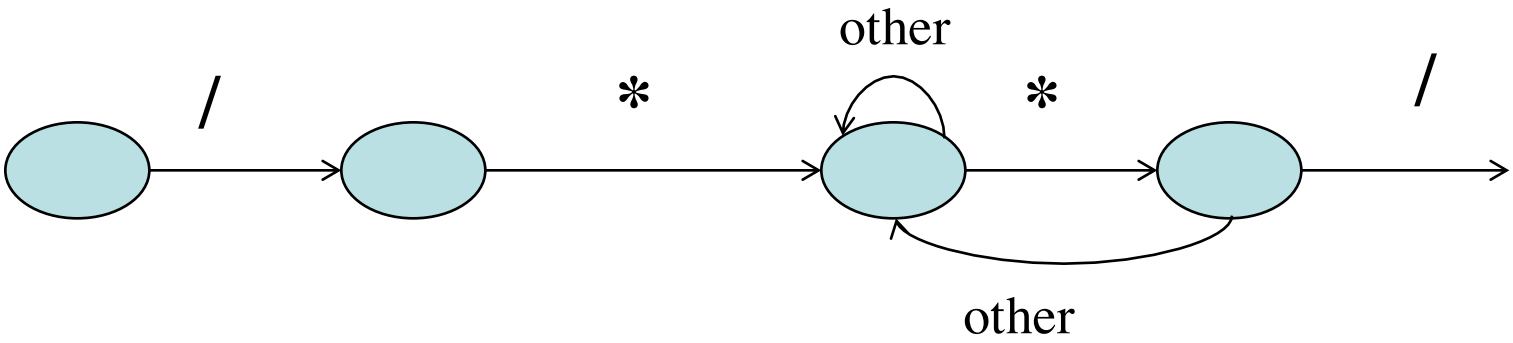
```
/*getChar - a function to get the next character of input and determine its
   character class*/

void getChar(){
   if(isalpha(nextChar))
      charClass = LETTER;
     else if (isdigit(nextChar))
         charClass = DIGIT;
       else
          charClass = UNKNOWN
}

/*lex - a simple lexical analyzer*/

int lex(){
     lexlen =0;
     static int first=1;

     if (first){
         getChar();
         first=0;
     }

     getNonBlank();

     switch (CharClass){
```

Implementing the State Diagram

```
    switch (charClass){

      case LETTER:
        addChar();
        getChar();
        while (charClass == LETTER || charClass == DIGIT){
              addChar();
              getChar();
        }
       return lookup(lexeme);
       break;

      case DIGIT:
        addChar();
        getChar();
        while (charClass == DIGIT){
              addChar();
              getChar();
        }
       return INT_LIT;
       break;

   } /*End of switch*/

} /*End of lex*/
```

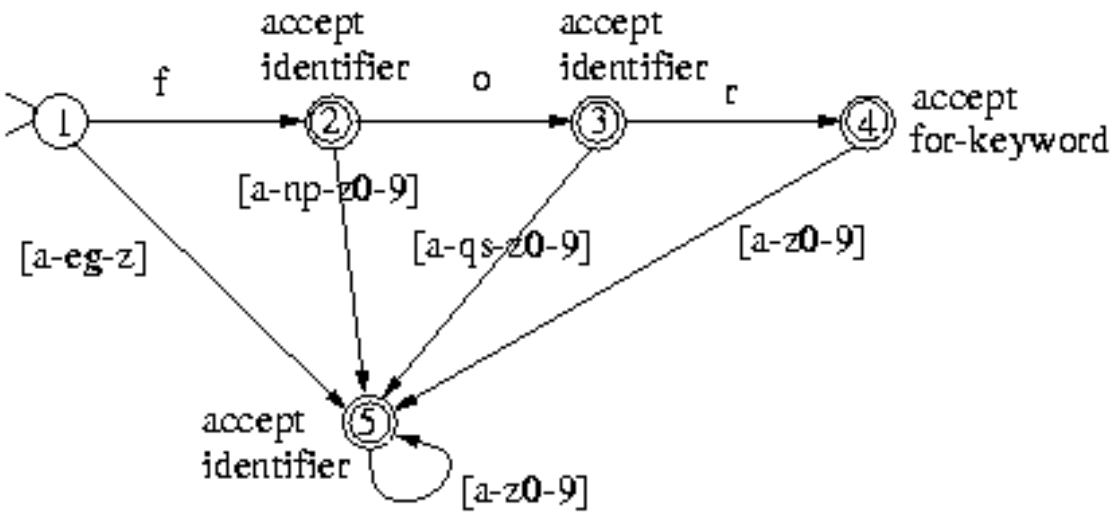
State Diagram

- Problem: Design a state diagram to recognize one form of the comments of C-based programming languages, those that begin with /* and end with */



Example: state diagram

For-Keyword = for
Identifier = [a-z][a-z0-9]*



Lexical Analyzer with Lex

- Lex is a domain-specific programming language for creating programs to process streams of input characters.
- A Lex program has the following form:

  declarations
  %%
  translation rules
  %%
  auxiliary functions

- The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty.
- The translation rules are each of the form pattern {action}
  - Each pattern is a regular expression which may use regular definitions defined in the declarations section.
  - Each action is a fragment of C-code. The braces around the action may be omitted if the action is a single statement.

Lexical Analyzer with Lex

a\+b matches the string a+b.
. matches any character except a newline.
^ matches the empty string at the beginning of a line.
$ matches the empty string at the end of a line.
[a-z] matches any lowercase letter between a and z.
[A-Za-z0-9] matches any alphanumeric character.
[^abc] matches any character except an a, or a b, or a c.
[^0-9] matches any nonnumeric character.
a* matches a string of zero or more a's.
a+ matches a string of one or more a's.
.a? matches a string of zero or one a's.
a{2,5} matches any string consisting of two to five a's.
a/b matches an a when followed by a b.
\n matches a newline.
\t matches a tab

Example with Lex

```
int num_words = 0,num_numbers = 0, num_lines = 0;
word [A-Za-z]+
number [0-9]+
%%
 {word} {++num_words;}
 {number} {++num_numbers;}
 \n {++num_lines; }
. { }
%%
int main() {
        yylex();
        printf("# of words = %d, # of numbers = %d, # of lines = %d\n",
                        num_words, num_numbers, num_lines );
}
```

Example with Lex

- Put lex program into a file file.l
- Compile the lex program with the command
  - lex file.l
    - This command produces an output file lex.yy.c.
- Compile this output file with the C compiler and the lex library –ll
  - gcc lex.yy.c –ll
    - The resulting a.out is the lexical processor.

The Parsing Problem

- Given an input program:
  - Operates on tokens and groups them into useful grammatical structures.
  - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program

The Parsing Problem (cont.)

- Two categories of parsers
  - *Top down* – produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
  - *Bottom up* – produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation

The Parsing Problem (cont.)

- Top-down Parsers
  - Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A
- The most common top-down parsing algorithms:
  - Recursive descent - a coded implementation
  - LL parsers - table driven implementation

The Parsing Problem (cont.)

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  - The most common bottom-up parsing algorithms are in the LR (Left-to-right, Rightmost derivation) type.

Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing (cont.)

- A grammar for simple expressions:

```
<expr> → <term> {(+ | -) <expr>}
<term> → <factor> {(* | /) <term>}
<factor> → id | int_constant | ( <expr> )
```

Recursive-Descent Parsing (cont.)

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`
- The coding process when there is only one RHS (Right Hand Side):
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing (cont.)

```
/* Function expr
   Parses strings in the language
   generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void expr() {

/* Parse the first term */

  term();
/* As long as the next token is + or -, call
   lex to get the next token and parse the
   next term */

  while (nextToken == ADD_OP ||
         nextToken == SUB_OP){
    lex();
    term();
  }
}
```

Recursive-Descent Parsing (cont.)

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

Recursive-Descent Parsing (cont.)

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>)
*/
void term() {
  printf("Enter <term>\n");
/* Parse the first factor */
  factor();
/* As long as the next token is * or /,
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
  printf("Exit <term>\n");
} /* End of function term */
```

Recursive-Descent Parsing (cont.)

```
/* Function factor
   Parses strings in the language
   generated by the rule:
   <factor> -> id  |  (<expr>)  */

 void factor() {

 /* Determine which RHS */
   if (nextToken) == ID_CODE || nextToken == INT_CODE)

 /* For the RHS id, just call lex */
     lex();

/* If the RHS is (<expr>) - call lex to pass over the left parenthesis,
   call expr, and check for the right parenthesis */
   else if (nextToken == LP_CODE) {
    lex();
      expr();
     if (nextToken == RP_CODE)
        lex();
      else
        error();
    }  /* End of else if (nextToken == ...  */

   else error(); /* Neither RHS matches */
}
```

Recursive-Descent Parsing (cont.)

Trace of the recursive descent parser for the string a + b * c

```
Call lex  /* returns a */
Enter <expr>
Enter <term>
Enter <factor>
Call lex  /* returns + */
Exit <factor>
Exit <term>
Call lex  /* returns b */
Enter <term>
Enter <factor>
Call lex  /* returns * */
Exit <factor>
Call lex  /* returns c */
Enter <factor>
Call lex  /* returns end-of-input */
Exit <factor>
Exit <term>
Exit <expr>
```

Bottom-up Parsing

- The Bottom-up Parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation
- LR Parsers
  - Canonical LR (Knuth, 1965): original LR algorithm

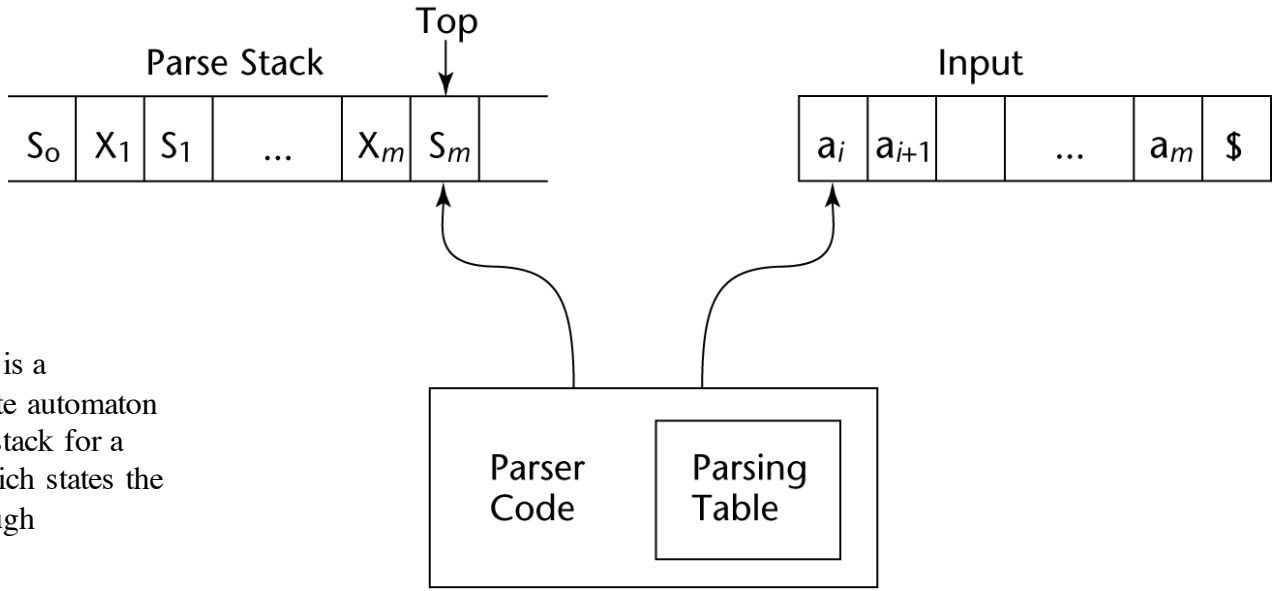
Bottom-up Parsing (cont.)

- Advantages of LR parsers:
  - They will work for nearly all grammars that describe programming languages.
  - They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser.
  - The LR class of grammars is a superset of the class parsable by LL parsers.

Structure of An LR Parser

- An LR configuration stores the state of an LR parser

$$(S_0X_1S_1X_2S_2\ldots X_mS_m,\ a_ia_i{+}1\ldots a_n\$)$$



A Pushdown automata is a deterministic finite state automaton with the addition of a stack for a memory indicating which states the parser has passed through

Bottom-up Parsing (cont.)

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
  - The ACTION table specifies the action of the parser, given the parser state and the next token
    - Rows are state names; columns are terminals
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
    - Rows are state names; columns are nonterminals

Bottom-up Parsing (cont.)

- Initial configuration: $(S_0, a_1 \ldots a_n\$)$
- Parser actions:
  - If ACTION$[S_m, a_i]$ = Shift S, the next configuration is:

    $(S_0X_1S_1X_2S_2 \ldots X_mS_ma_iS, a_{i+1} \ldots a_n\$)$
  - If ACTION$[S_m, a_i]$ = Reduce $A \rightarrow \beta$ and S = GOTO$[S_{m-r}, A]$, where r = the length of β, the next configuration is

    $(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS, a_ia_{i+1} \ldots a_n\$)$

Bottom-up Parsing (cont.)

- Parser actions (continued):
  - If ACTION[S_m, a_i] = Accept, the parse is complete and no errors were found.
  - If ACTION[S_m, a_i] = Error, the parser calls an error-handling routine.

LR Parsing Table

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

	Action						Goto		
State	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
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Trace of a parse

Stack	*Input*	*Action*
0	id * (id + id) $	Shift 5
0id5	* (id + id) $	Reduce 6 (Use GOTO[0, F])
0F3	* (id + id) $	Reduce 4 (Use GOTO[0, T])
0T2	* (id + id) $	Shift 7
0T2*7	(id + id) $	Shift 4
0T2*7(4	id + id) $	Shift 5
0T2*7(4id5	+ id) $	Reduce 6 (Use GOTO[4, F])
0T2*7(4F3	+ id) $	Reduce 4 (Use GOTO[4, T])
0T2*7(4T2	+ id) $	Reduce 2 (Use GOTO[4, E])
0T2*7(4E8	+ id) $	Shift 6
0T2*7(4E8+6	id) $	Shift 5
0T2*7(4E8+6id5	)$	Reduce 6 (Use GOTO[6, F])
0T2*7(4E8+6F3	)$	Reduce 4 (Use GOTO[6, T])
0T2*7(4E8+6T9	)$	Reduce 1 (Use GOTO[4, E])
0T2*7(4E8	) $	Shift 11
0T2*7(4E8)11	$	Reduce 5 (Use GOTO[7, F])
0T2*7F10	$	Reduce 3 (Use GOTO[0, T])
0T2	$	Reduce 2 (Use GOTO[0, E])
0E1	$	ACCEPT

Parsers Classification

- LL(k) Top down parser
  - LL(1) are simple parsers but cannot recognize all context free grammars
- LR(k) Bottom up parser
  - LR(1) are more powerful
  - LALR(1)
    - Not as powerful as full LR(1) but simpler to implement
    - Yacc uses LALR(1) parse tables to construct a restricted LR

Recommended Links

- Parsing Simulator
- Compiler component generators
  - lexical analyzer generators: lex, flex
  - syntax analyzer generator: yacc, bison
  - PLY (Python Lex-Yacc)
  - ANTLR lexer-parser generator
  - Compact Guide to Lex & Yacc
  - Flex
  - GNU Bison
  - JavaCC

Summary

- BNF and context-free grammars are equivalent meta-languages to describe syntax
- Regular expressions and Finite Automata
- Syntax analysis
  - Lexical Analysis (Produce tokens)
  - Parsing (Produces a parse tree)
- Lexical Analyzer
- Parsing
  - Top-down approach
    - Recursive-descent parser
  - Bottom-up approach
    - The LR family of shift-reduce parsers is the most common bottom-up parsing approach