

# Informatik II

## Languages, Compilers, and Theory

---

Lectures 3 & 4:  
Automata Theory Review  
and  
Parsing



# Review

---

- Last time:
  - Regular languages, regular expressions, finite automata (deterministic and nondeterministic)
  - Key Algorithms:
    - Conversion from regular expressions to nondeterministic finite automata (NFA)
    - Conversion from nondeterministic finite automata to deterministic finite automata (DFA)
    - Table-based simulation of DFAs
  - Lexical analysis and scanners
    - Using regular expressions to define a language's lexical structure (tokens)
    - Using theory of regular languages to generate a scanner from a regular expression-based description of a programming language's lexical structure



# Regular Expressions (1)

---

- Defined as:
  - A character  $c$  taken from an alphabet  $\Sigma$ , or
  - The empty string  $\epsilon$ , or
  - The alternation of two regular expressions,  $r1 \mid r2$ , or
  - The concatenation of two regular expressions,  $r1 . r2$ , or
  - The Kleene closure of a regular expression,  $r1^*$
- A regular expression is said to **generate** strings of characters from the alphabet  $\Sigma$
- The set of all strings generated by a regular expression  $R$  is called the **language of  $R$**  and is denoted  **$L(R)$**



# Regular Expressions (2)

---

- Notational shortcuts:
  - Strings
    - $'c_1c_2\text{---}c_n' = c_1.c_2.\text{---}.c_n$
  - Sequences
    - $[c_1\text{---}c_n] = c_1|c_2|\text{---}|c_n$
  - Kleene +
    - $r^+ = r.r^*$



# Regular Expressions (3)

Regular Expression...	Generates...
'if' 'then' 'else'	The strings <i>if</i> , <i>then</i> , or <i>else</i> .
$a.(a b c)^*.a$	All strings of a's, b's, and c's which begin and end with an a
$a.(a b c)^*.a$	All strings of a's, b's, and c's which begin with an a and end with a single a.
$(a b c)^*.a.b.a.(a b c)^*$	All strings of a's, b's, and c's which contain <i>aba</i> as a substring.
$(b c)^*.a.(b c)^*.a.(b c)^*.a.(b c)^*$	All strings of a's, b's, and c's which contain exactly three a's.



# Finite Automata (1)

---

- Defined as a 5-tuple,  $M=(Q,\Sigma,q_0,F,\delta)$ , where:
  - $Q$  is a set of symbols called **states**
  - $\Sigma$  is a set of symbols called **the alphabet**
  - $q_0$  is the **start state**
  - $F$  is a possibly empty set of **final states**
  - $\delta$  is a **transition function**
- A finite automaton is said to **accept** strings of characters from the alphabet  $\Sigma$
- The set of all strings accepted by an automaton  $M$  is called the **language of  $M$**  and is denoted  $L(M)$



# Finite Automata (2)

---

- Deterministic Finite Automata (DFA)
  - Transitions are deterministic
    - Single states to single states
    - $\delta_D: Q \times \Sigma \rightarrow Q$
  - Transition function can be written as a **table** or as a **state transition diagram**

# Finite Automata (3)

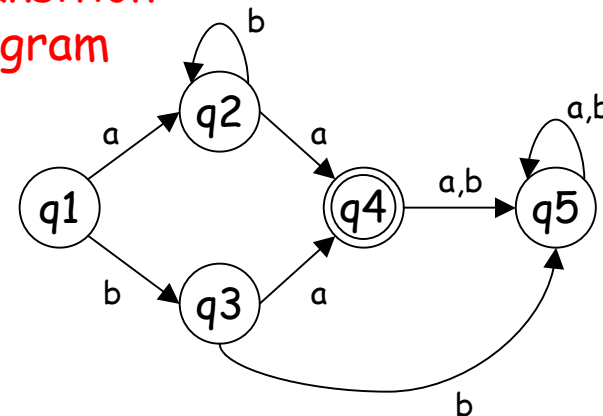
## Example DFA

- $Q = \{q_1, q_2, q_3, q_4, q_5\}$
- $\Sigma = \{a, b\}$
- $q_0 = q_1$
- $F = \{q_4\}$
- $\delta =$ 
  - $\{(q_1, a), q_2\}, \{(q_1, b), q_3\},$
  - $\{(q_2, a), q_4\}, \{(q_2, b), q_2\},$
  - $\{(q_3, a), q_4\}, \{(q_3, b), q_5\},$
  - $\{(q_4, a), q_5\}, \{(q_4, b), q_5\},$
  - $\{(q_5, a), q_5\}, \{(q_5, b), q_5\}$

Table

	a	b
q1	q2	q3
q2	q4	q2
q3	q4	q5
q4	q5	q5
q5	q5	q5

State  
Transition  
Diagram





# Finite Automata (4)

---

- Nondeterministic Finite Automata (NFA)
  - Transitions are nondeterministic
    - Single states to **sets of possible states**
    - $\delta_D: Q \times \Sigma \rightarrow 2^Q$
  - Alphabet is extended to allow transitions on the **empty string**  $\epsilon$

# Finite Automata (5)

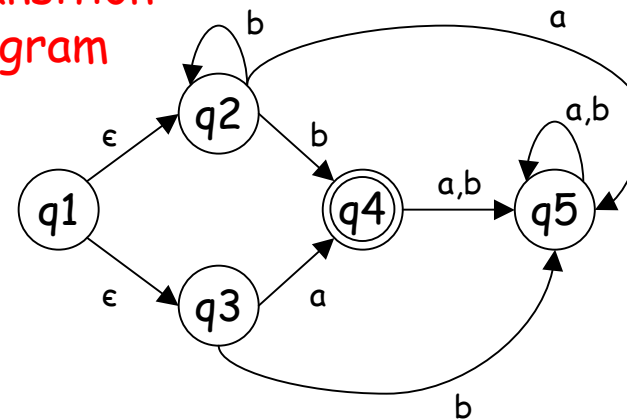
## Example NFA

- $Q = \{q_1, q_2, q_3, q_4, q_5\}$
- $\Sigma = \{a, b, \epsilon\}$
- $q_0 = q_1$
- $F = \{q_4\}$
- $\delta =$   
{  
((q1,  $\epsilon$ ), {q2, q3}),  
((q2, a), {q5}), ((q2, b), {q2, q4}),  
((q3, a), {q4}), ((q3, b), {q5}),  
((q4, a), {q5}), ((q4, b), {q5}),  
((q5, a), {q5}), ((q5, b), {q5})  
}

Table

	$\epsilon$	a	b
q1	{q2, q3}	{q5}	{q5}
q2	{q5}	{q5}	{q2, q4}
q3	{q5}	{q4}	{q5}
q4	{q5}	{q5}	{q5}
q5	{q5}	{q5}	{q5}

State  
Transition  
Diagram



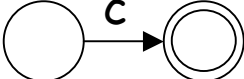
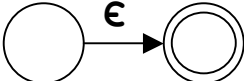
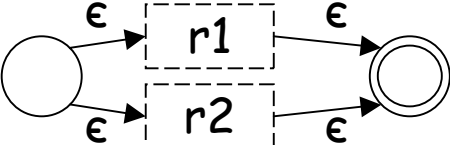
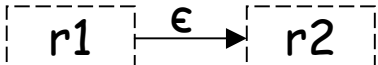
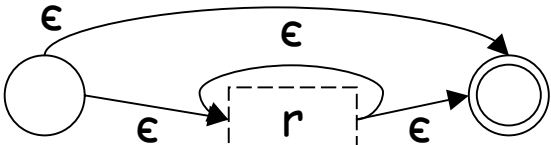


# Regular Languages

---

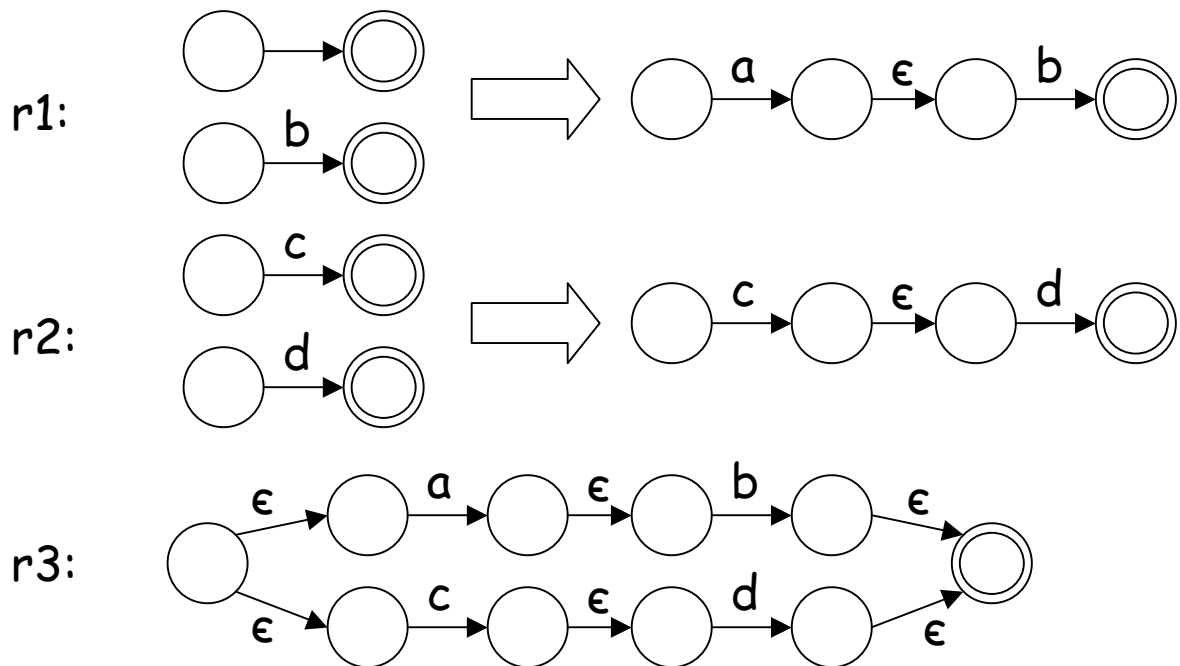
- A language  $L(X)$  is regular if:
  - There exists a regular expression  $R$  s.t.  $L(R) = L(X)$ , or
  - There exists a DFA  $M_D$  s.t.  $L(M_D) = L(X)$ , or
  - There exists a NFA  $M_N$  s.t.  $L(M_N) = L(X)$
- The regular expression languages, DFA languages, and NFA languages **are all regular**
- Given any regular expression  $R$  and NFA  $M_N$ ,  $L(R) = L(M_N)$ 
  - We can convert any regular expression to an NFA and vice versa
- Given any NFA  $M_N$  and DFA  $M_D$ ,  $L(M_N) = L(M_D)$ 
  - We can convert any NFA to a DFA and vice versa

# Regular Expressions to NFAs

Regular Expression	NFA
Character: $c$	
Empty string: $\epsilon$	
Alternation: $r1 r2$	
Concatenation: $r1.r2$	
Kleene Closure: $r^*$	

# Regular Expressions to NFAs

- $r = 'ab'|'cd'$
- Factor:
  - $r1 = 'ab'$
  - $r2 = 'cd'$
  - $r = r1.r2$





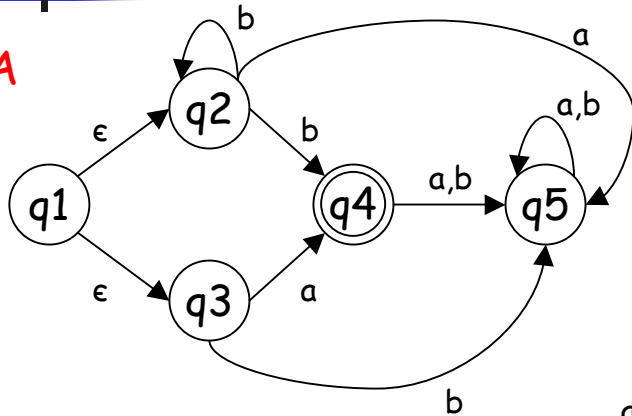
# NFAs to DFAs

---

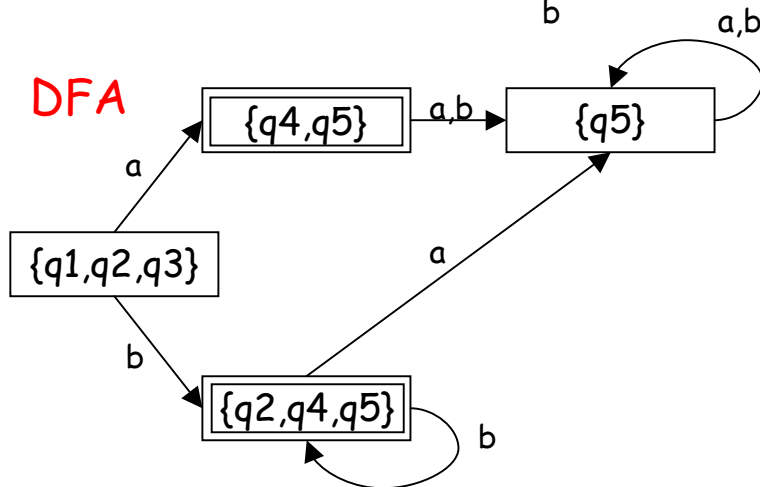
- Definition:
  - $\text{closure}(s)$  is the set containing the states in  $s$  plus all states reachable from the states in  $s$  using only  $\epsilon$  transitions
- Given NFA  $M=(Q,\Sigma,q,F,\delta)$  and DFA  $M_D=(Q_D,\Sigma,q_D,F_D,\delta_D)$ 
  - $Q_D=2^Q$ , i.e.,  $Q_D$  is the set of all subsets of  $Q$
  - $F_D = \{S: \forall S \in Q_D \text{ where } S \cap F \neq \{\}\}$
  - $q_D = \text{closure}(q)$
  - $\delta_D(\{q_1, q_2, \dots, q_k\}, a) = \text{closure}(\delta(q_1, a) \cup \delta(q_2, a) \cup \dots \cup \delta(q_k, a))$

# NFAs to DFAs

NFA



DFA



- Step 1: The start state
  - $q_D = \text{closure}(\{q_1\}) = \{q_1, q_2, q_3\}$
- Step 2: state  $\{q_1, q_2, q_3\}$ 
  - $\delta_D(\{q_1, q_2, q_3\}, a) = \text{closure}(\delta(q_1, a) \cup \delta(q_2, a) \cup \delta(q_3, a)) = \text{closure}(\{q_5\} \cup \{q_4\}) = \{q_4, q_5\}$
  - $\delta_D(\{q_1, q_2, q_3\}, b) = \text{closure}(\delta(q_1, b) \cup \delta(q_2, b) \cup \delta(q_3, b)) = \text{closure}(\{q_2, q_4\} \cup \{q_5\}) = \{q_2, q_4, q_5\}$
- Step 3: state  $\{q_4, q_5\}$ 
  - ...
- Step 3: state  $\{q_2, q_4, q_5\}$ 
  - ...



# Worksheet (1)

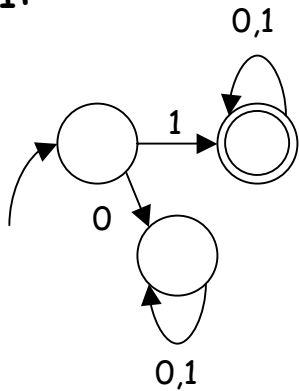
---

- Write regular expressions that generate the following:
  1. Binary numbers (strings over the alphabet  $\{0,1\}$ ) with **no** leading 0's.
    - For example, 101010 is a valid string, but 001010 is not.
  2. Strings over the alphabet  $\{a,b,c\}$  where the **length** of the string is even.
    - For example, abcbbb is a valid string, but abcbb is not.
  3. Strings over the alphabet  $\{a,b,c\}$  where the first a **precedes** the first b.
    - For example, ccabbac is a valid string, but ccbaabc is not.
  4. Strings over the alphabet  $\{a,b,c\}$  that **don't** contain the substring baa.

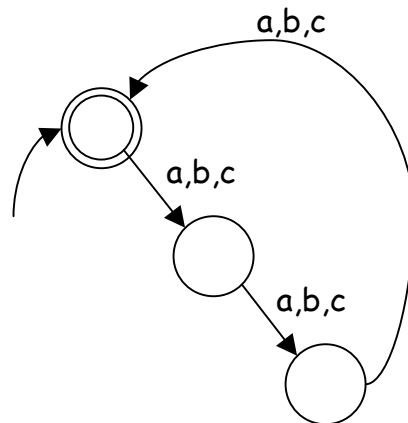
# Worksheet (2)

- Describe the strings which the following automata accept:

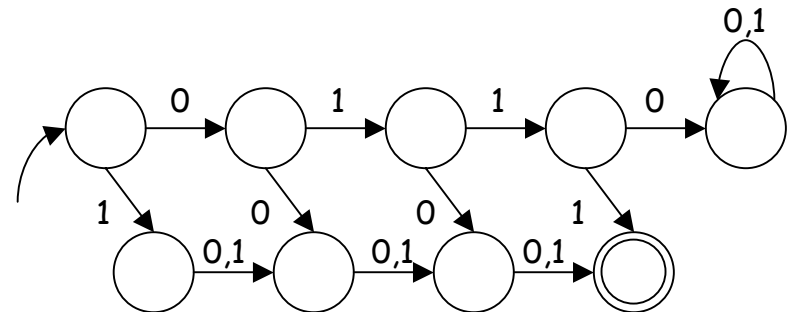
1:



2:

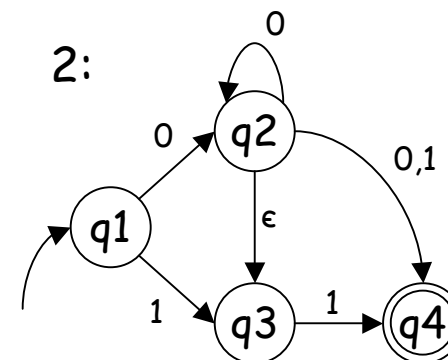
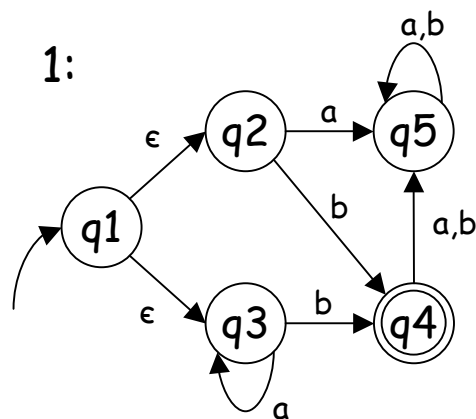


3:

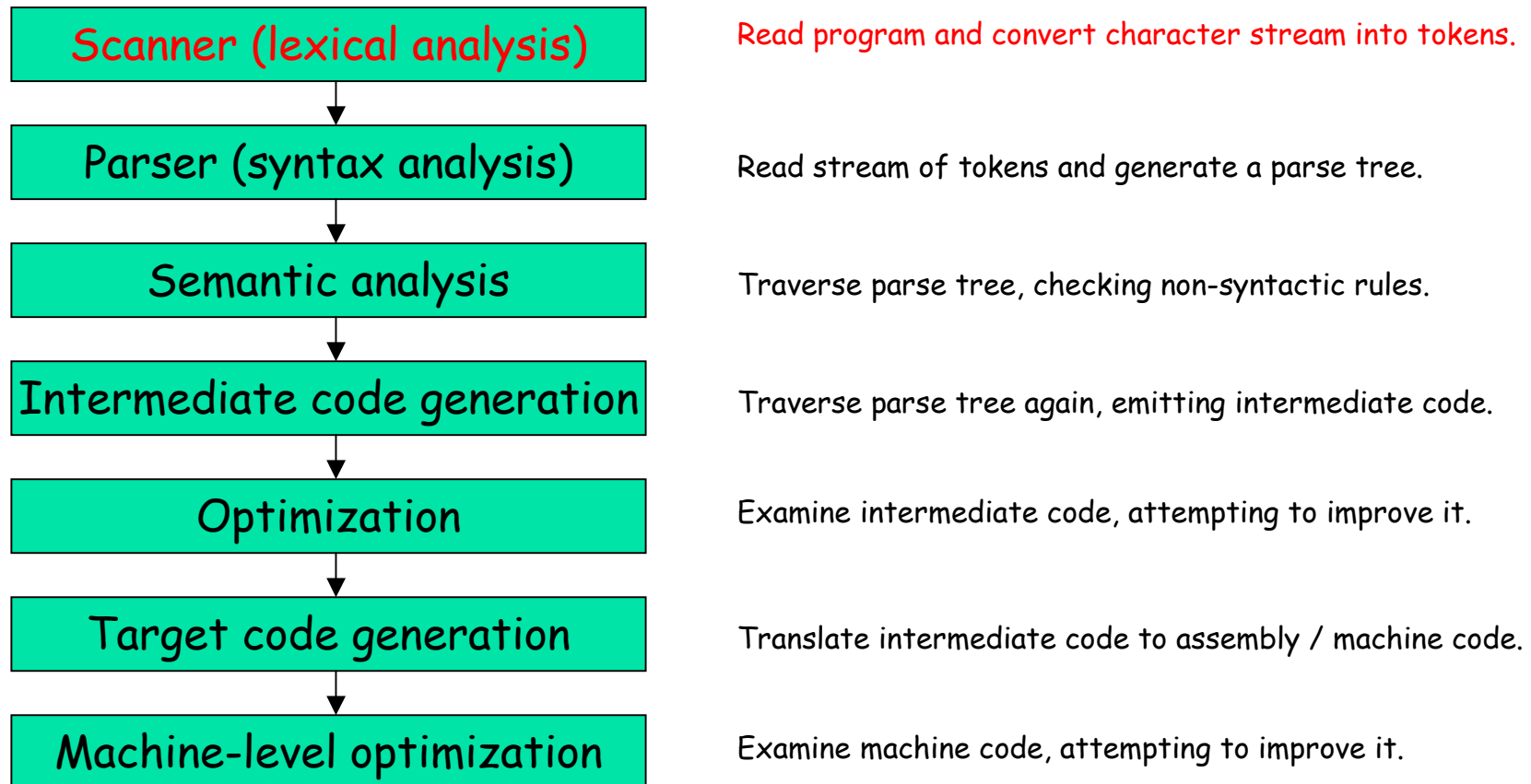


# Worksheet (3)

- Convert the following regular expressions to NFAs:
  1.  $(a|b).c$
  2.  $a.(b|c)^*$
- Convert the following NFAs to DFAs:



# The Compilation Process (Phases)



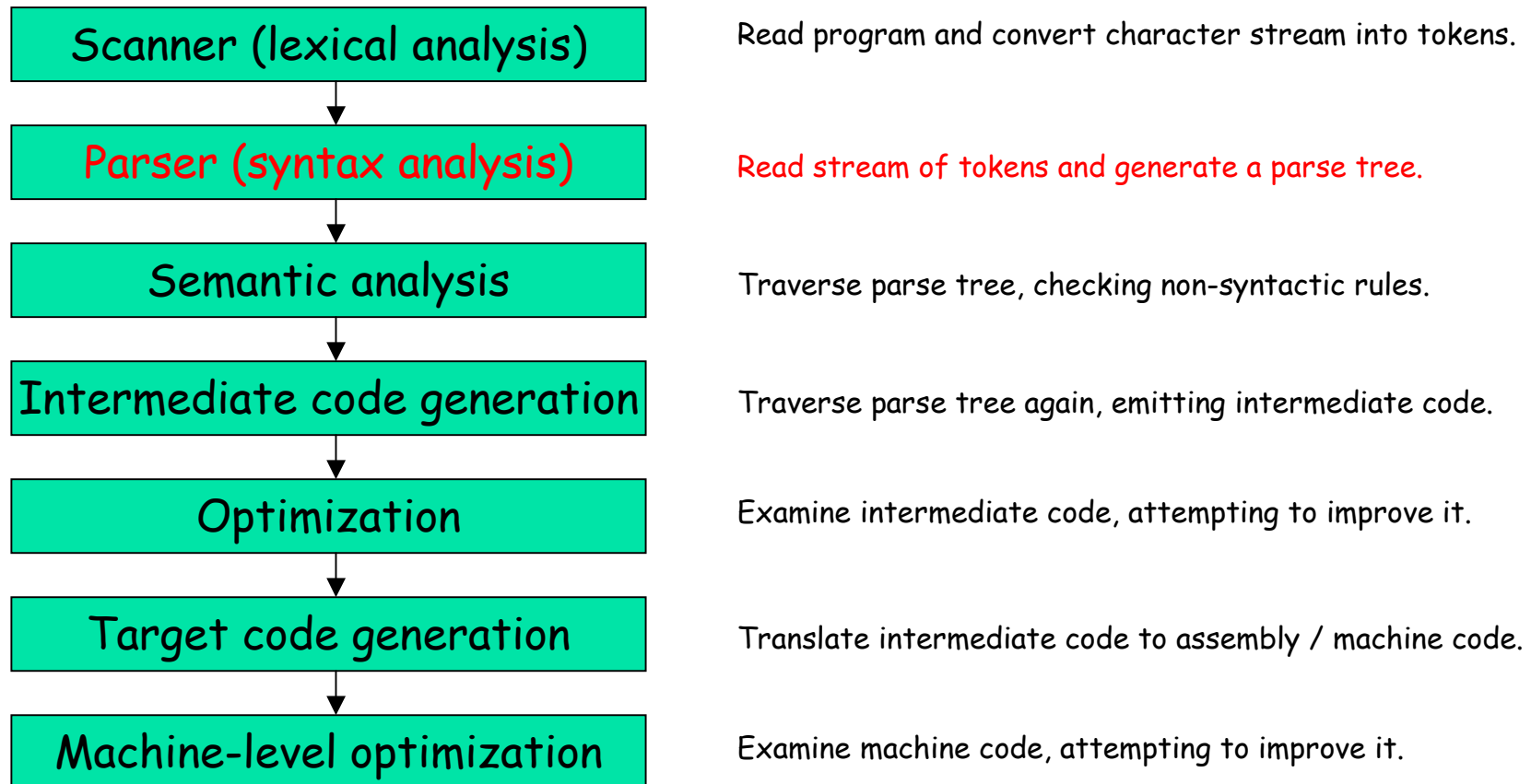


# Lexical analysis

---

- Lexical analysis groups sequence of characters into **tokens**
- Tokens are the smallest “units of meaning” in the compilation process and are the foundation for parsing (syntax analysis)
- The compiler component that performs lexical analysis is called the **scanner** and is often automatically generated from a high-level specification

# The Compilation Process (Phases)





# Syntax analysis

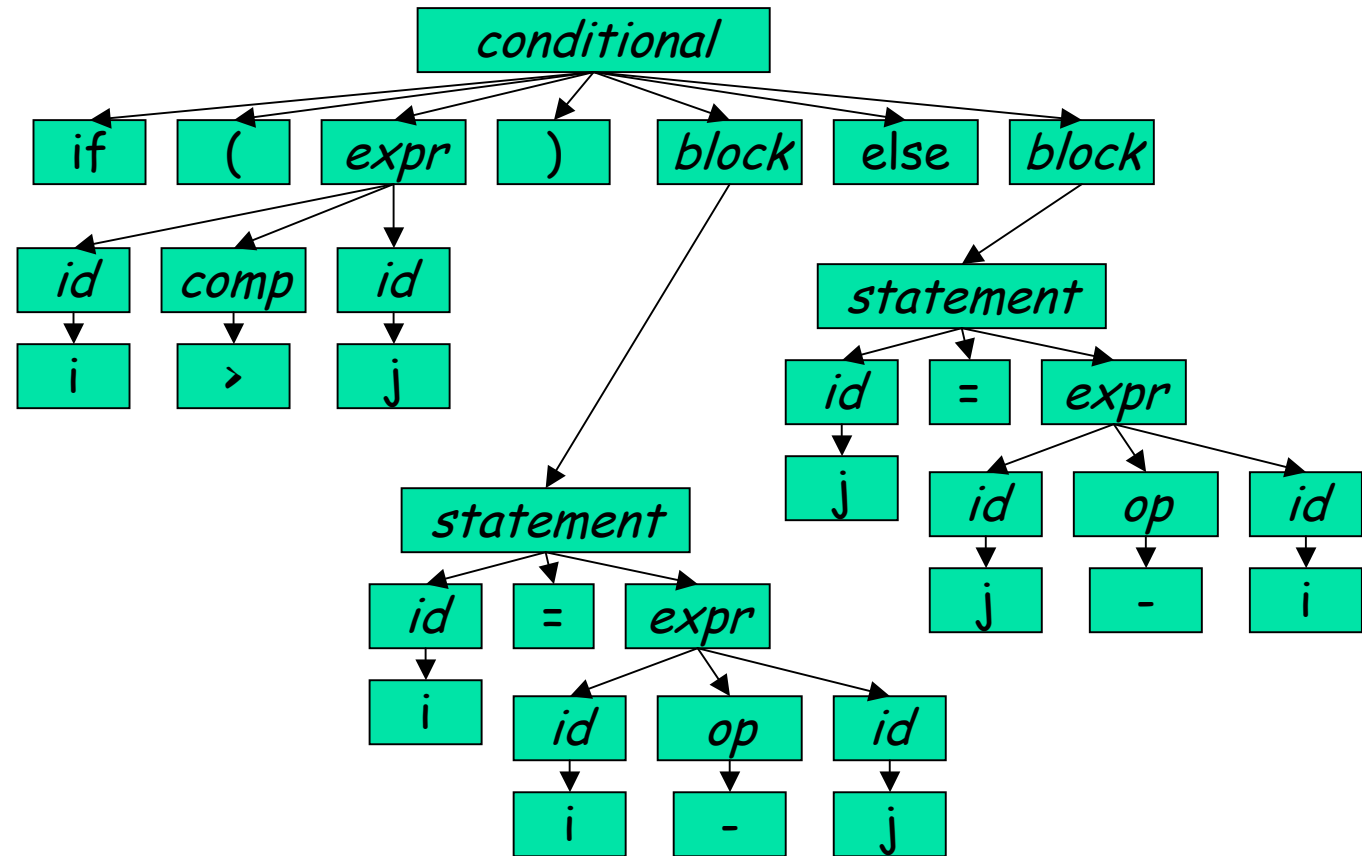
---

- Lexical analysis generates a stream of tokens
- Wrong level of detail for semantic analysis and code generation
- Syntax analysis groups a string of tokens into **parse trees** guided by the **context-free grammar** that specifies the **syntax** of the language being compiled
  - conditional -> `if ( expr ) block else block`
- Parse trees represent the **phrase structure** of the program and are the foundation for semantic analysis and code generation
- The compiler component that performs syntax analysis is called the **parser** and is also often generated from a high-level specification

# Syntax analysis example

Tokens

if	(	i
>	j	)
i	=	i
-	j	;
else	j	=
j	-	i
;	}	





# Describing Syntax

---

- Why can't we use regular expressions to describe the syntax of a programming language?
- Consider the following descriptions.
  - Identities:
    - $\text{digit}=[0-9]$
    - $\text{letter}=[a-z]$
    - $\text{id}=\text{letter}.\text{(letter|digit)}^*$
  - Can remove identities from id by substitution:
    - $\text{id}=[0-9].\text{([a-z]|[0-9])}^*$
    - id is a regular expression
  - Identities:
    - $\text{digits}=[0-9]^+$
    - $\text{sum}=\text{expr}.\text{'+'}.\text{expr}$
    - $\text{expr}=\text{'('}.\text{'sum.'}\text{'|digits}$
  - Can **not** remove identities from expr by substitution:
    - expr is defined by recursion
    - expr is not a regular expression



# Context Free Grammars

---

- A context free grammar (CFG) is a recursive definition of a language with:
  - An alphabet  $\Sigma$  of **symbols**
  - A set of **productions** of the form
    - symbol  $\rightarrow$  symbol symbol ... symbol
  - A **start symbol**
  - A set of **non-terminal symbols** taken from the alphabet  $\Sigma$  which may appear on the left or right hand sides of productions
  - A set of **terminal symbols** taken from the alphabet  $\Sigma$  which may appear only on the right hand sides of productions
- The set of all strings **generated** by a CFG  $G$  is said to be the **language of  $G$**  and is denoted  $L(G)$



# Context Free Grammars

---

- Notational Shortcuts
  - Alternation
    - $s \rightarrow a_1..a_n | b_1..b_n | \dots | z_1..z_n =$
    - $s \rightarrow a_1..a_n$
    - $s \rightarrow b_1..b_n$
    - ...
    - $s \rightarrow z_1..z_n$
  - Kleene \* closure
    - $s \rightarrow s_1^*$
    - $s \rightarrow s_1'$
    - $s_1' \rightarrow s_1 s_1'$
    - $s_1' \rightarrow \epsilon$



# Context Free Grammars

---

- When using a CFG  $G$  to parse programming languages
  - $L(G)$  is the set of **valid source programs**
  - The **terminal symbols** are the **tokens** returned by the scanner
- Example:
  - $\text{expr} \rightarrow \text{LPAREN sum RPAREN}$
  - $\text{expr} \rightarrow \text{INT}$
  - $\text{sum} \rightarrow \text{expr PLUS expr}$
  - Terminals:  $\{\text{PLUS}, \text{LPAREN}, \text{RPAREN}, \text{INT}\}$
  - Non-terminals:  $\{\text{sum}, \text{expr}\}$
  - Start symbol:  $\{\text{expr}\}$
  - $\Sigma = \text{Terminals} \cup \text{Non-terminals}$



# Context Free Grammars

---

- A CFG  $G$  generates strings by:
  - Beginning with the start symbol
    - $s \Rightarrow s_1 s_2 \dots s_n$
  - Replacing a non-terminal symbol  $s_k$  on the right hand side with that non-terminal's right hand side
    - Given:  $s_k \rightarrow k_1 \dots k_m$
    - Then:  $s \Rightarrow s_1 \dots s_k \dots s_n \Rightarrow s_1 \dots k_1 \dots k_m \dots s_n$
  - Repeating until only terminal symbols are left
  - Each step in this process is called a **derivation** and each string of symbols along the way is called **sentential form**
  - The final sentential form containing only terminal symbols is called a **sentence** of the grammar or the **yield** of the derivation process



# Derivations

---

- Grammar:

$\text{expr} \rightarrow (\text{sum})$

$\text{expr} \rightarrow \text{INT}$

$\text{sum} \rightarrow \text{expr} + \text{expr}$

- Possible derivation:

$\text{expr} \Rightarrow (\text{sum})$

$\Rightarrow (\underline{\text{expr}} + \text{expr})$

$\Rightarrow (\underline{\text{INT}} + \text{expr})$

$\Rightarrow (\text{INT} + \underline{\text{sum}})$

$\Rightarrow (\text{INT} + (\underline{\text{expr}} + \text{expr}))$

$\Rightarrow (\text{INT} + (\underline{\text{INT}} + \text{expr}))$

$\Rightarrow (\text{INT} + (\text{INT} + \underline{\text{INT}}))$



# Derivations

---

- Rightmost derivations
  - Replace the rightmost non-terminal symbol at each derivation step
  - Sometimes called the **canonical derivation**
- Leftmost derivations
  - Replace the leftmost non-terminal symbol at each derivation step
  - See previous slide
- Other derivation orders are possible
- Most parsers work by finding either a rightmost or leftmost derivation



# Derivations and Parse Trees

---

- A **parse tree** is a graphical representation of the derivation process
- Parse tree **leaves** correspond to grammar **terminal symbols** (tokens)
- Parse tree **interior nodes** correspond to grammar **non-terminal symbols** (production left hand sides)
- Most parsers construct a parse tree during the derivation process for later analysis

# Derivations and Parse Trees

expr

⇒ ( sum )

⇒ ( expr + expr )

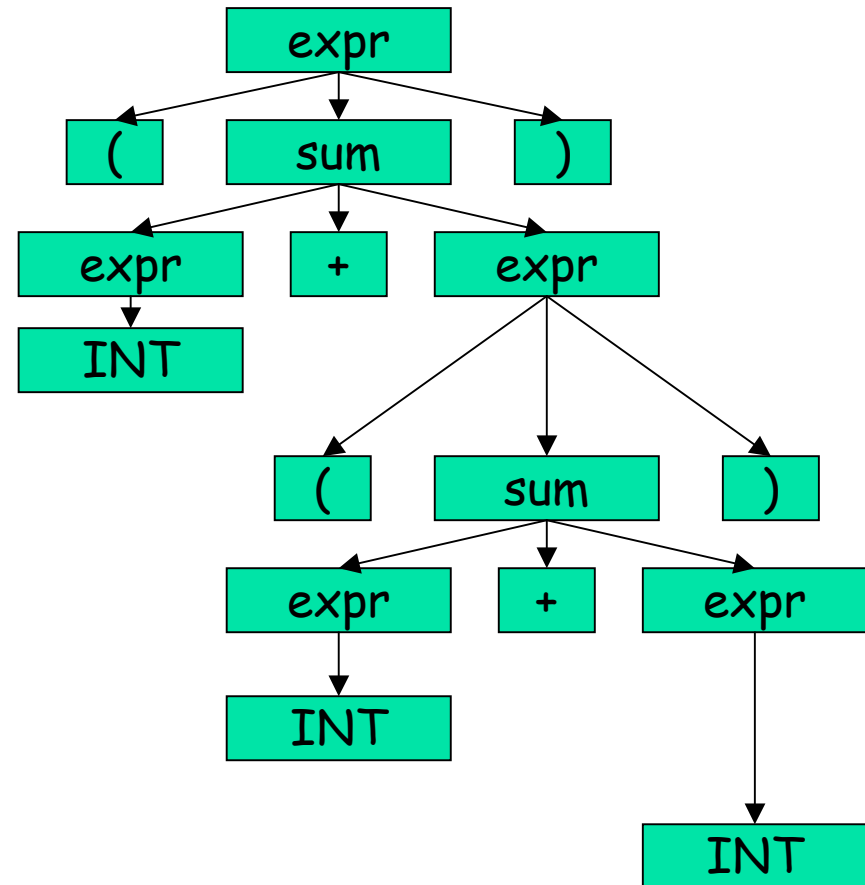
⇒ ( INT + expr )

⇒ ( INT + ( sum ) )

⇒ ( INT + ( expr + expr ) )

⇒ ( INT + ( INT + expr ) )

⇒ ( INT + ( INT + INT ) )



# Ambiguity

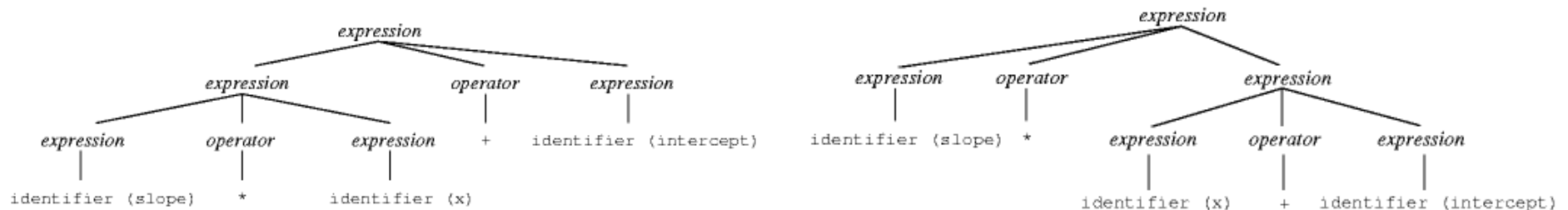
- A grammar is said to be ambiguous if it can derive a sentence with two different parse trees
- Example - leftmost versus rightmost derivation:

- Grammar:

expression  $\rightarrow$  identifier | number | - expression | ( expression ) |  
expression operator expression

operator  $\rightarrow$  + | - | \* | /

- Input: slope \* x + intercept





# Ambiguity

---

- More famous example - "dangling else"
  - Program fragment: `if a then if b then s1 else s2`
  - Can be interpreted as:
    - 1) `if a then { if b then s1 else s2 }`
    - 2) `if a then { if b then s1 } else s2`
- Sometimes ambiguity can be handled by choosing which of several possible parse trees to accept
  - For example, interpretation #1 above is chosen by most parsers for languages with the "dangling else" ambiguity
- In general though, ambiguity is an indication that the grammar is ill-specified and should be rewritten to remove the ambiguity



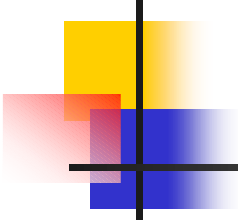
# Parsers and Grammars

---

- Context free grammars **generate** strings (or sentences) through the derivation process
- The context free languages are defined as
  - $L_{CF} = \{L(G) : \text{All context free grammars } G\}$
  - In other words the set of all languages of all context free grammars
- A parser for a context free grammar **recognizes** strings in the grammars language
- Parsers can be automatically generated from context free grammars
- Parsers that can recognize general context free languages can run slowly
- Parsers which can only recognize a subset of the context free languages can be made to run faster



# Bottom-up versus Top-down Parsing



- Top-down or LL Parsing
  - Build parse tree from root of the parse tree down to the leaves
  - At each step, predict which production to use to expand the current non-terminal parse tree node by looking at the next  $k$  input tokens
- Bottom-up or LR Parsing
  - Build parse tree from leaves of the parse tree up to the root
  - At each step, determine whether or not a collection of parse tree nodes can be joined to a single parent node

# Top-down versus Bottom-up Parsing

- Grammar:

$id\_list \rightarrow id\ id\_list\_tail$

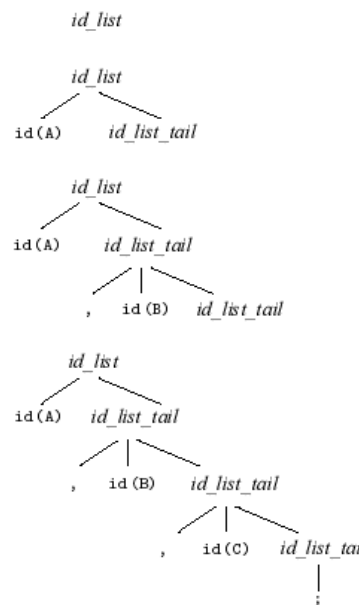
$id\_list\_tail \rightarrow ,\ id\ id\_list\_tail$

$id\_list\_tail \rightarrow ;$

- Example Strings:

A;

A, B, C;



$id\_list \rightarrow id\ id\_list\_tail$   
 $id\_list\_tail \rightarrow ,id\ id\_list\_tail$   
 $id\_list\_tail \rightarrow ;$






# Recursive Descent Parsing

---

- Recursive descent is a way to implement LL(1) parsers
  - Recall: LL(1) parsers make leftmost derivations using at most 1 token of look ahead to decide which production right hand side to use for the leftmost non-terminal being replaced in a sentential form
  - Example:
    - Grammar fragment:
      - $factor \rightarrow ( expr ) \mid [ sexpr ]$
    - When sentential form is "n1 ... nk factor sm ... sn", should the next sentential form be:
      - "n1 ... nk ( expr ) sm ... sn" or
      - "n1 ... nk [ sexpr ] sm ... sn"
- Easy to write by hand
  - No parser generator required
- Every non-terminal symbol in the grammar has a procedure call

# Recursive Descent Parsing



## Grammar for calculator language:

```
program → stmt_list $$
stmt_list → stmt stmt_list | ε
stmt → id := expr | read id | write expr
expr → term term_tail
term_tail → add_op term term_tail | ε
term → factor factor_tail
factor_tail → mult_op factor factor_tail | ε
factor → ( expr ) | id | literal
add_op → + | -
mult_op → * | /
```

## Sample input:

```
read A
read B
sum := A + B
write sum
write sum / 2
```

```
procedure match (expected)
  if input_token = expected
    consume input_token
  else error
```

-- this is the start routine:

```
procedure program
  case input_token of
    id, read, write, $$ :
      stmt_list
    match ($$)
  otherwise error
```

```
procedure stmt_list
  case input_token of
    id, read, write : stmt; stmt_list
    $$ : skip -- epsilon production
  otherwise error
```

```
procedure stmt
  case input_token of
    id : match (id); match (:=); expr
    read : match (read); match (id)
    write : match (write); expr
  otherwise error
```

```
procedure expr
  case input_token of
    id, literal, ( : term; term_tail
  otherwise error
```

```
procedure term_tail
  case input_token of
    +, - : add_op; term; term_tail
    ), id, read, write, $$ :
      skip -- epsilon production
  otherwise error
```

```
procedure term
  case input_token of
    id, literal, ( : factor; factor_tail
  otherwise error
```

```
procedure factor_tail
  case input_token of
    *, / : mult_op; factor; factor_tail
    +, -, ), id, read, write, $$ :
      skip -- epsilon production
  otherwise error
```

```
procedure factor
  case input_token of
    id : match (id)
    literal : match (literal)
    ( : match ( ( ); expr; match ( ) )
  otherwise error
```

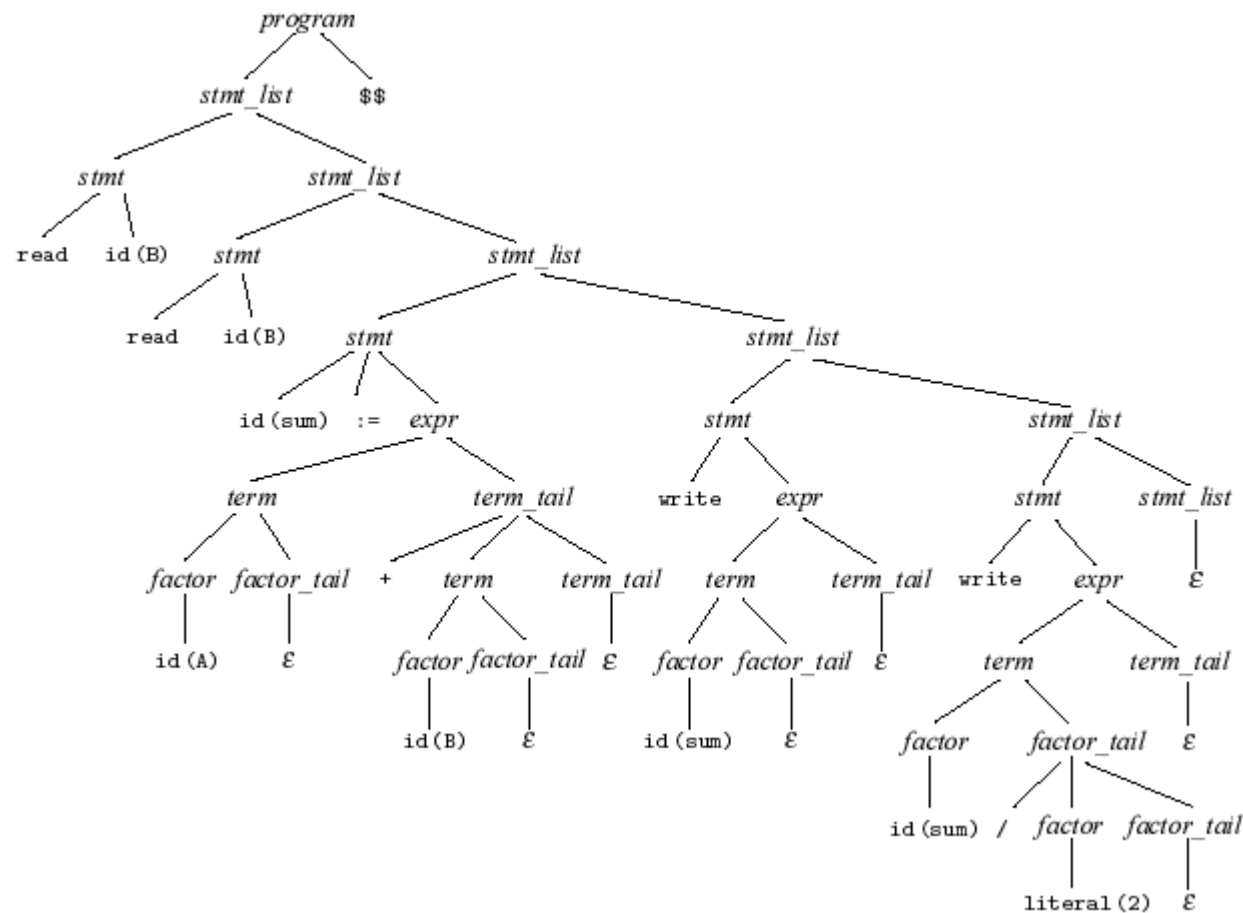
```
procedure add_op
  case input_token of
    + : match (+)
    - : match (-)
  otherwise error
```

```
procedure mult_op
  case input_token of
    * : match (*)
    / : match (/)
  otherwise error
```

# Recursive Descent Parsing

Parse tree for  
sample input:

```
read A
read B
sum := A + B
write sum
write sum / 2
```





# Problems with LL Parsing

---

- Left recursion
  - If grammar contains left-recursive productions then there can be no LL parser for it
    - Parser would go into an infinite loop
  - Example:
    - `id_list -> id_list_prefix ;`
    - `id_list_prefix -> id_list_prefix , id | id`
  - Left recursion can be eliminated by rewriting the grammar
    - `id_list -> id id_list_tail`
    - `id_list_tail -> , id id_list_tail | ;`



# Problems with LL Parsing

---

- Common Prefixes
  - Occur when two different productions with the same left hand side begin with the same symbols
    - Parser can't decide which production to choose
  - Example:
    - `stmt -> id := expr`
    - `stmt -> id ( argument_list )`
  - Common prefixes can be eliminated by rewriting the grammar
    - `stmt -> id stmt_list_tail`
    - `stmt_list_tail -> expr | ( argument_list )`



# Problems with LL Parsing

---

- Eliminating left recursion and common prefixes **does not** guarantee that the grammar becomes LL parse-able
- When we can't find an LL parser for a grammar, we must use a more powerful technique
  - For example, LALR(1) - grammars handled by the yacc and bison parser generators

# Building a Recursive Descent Parser

- Each non-terminal in the grammar has a subroutine

- Example:

- factor  $\rightarrow$  ( expr )
- factor  $\rightarrow$  [ sexpr ]

- Hard part:

- Figuring out which tokens label the 'case' arms of the switch statement

```
void factor (void) {  
    switch(next_token()) {  
        case '(':  
            expr(); match('('); break;  
        case '[':  
            sexpr(); match('['); break;  
    }  
}
```

# Building a Recursive Descent Parser

## ■ PREDICT Sets

- Tell us which production right hand side to choose in a leftmost derivation when there are several choices
- Defined in terms of FIRST, FOLLOW, and NULLABLE sets
  - NULLABLE( $X$ ) is true if  $X$  can derive the empty string
  - FIRST( $y$ ) is the set of all terminal symbols that can begin strings of symbols derived from  $y$
  - FOLLOW( $X$ ) is the set of all terminal symbols that can immediately follow  $y$ 
    - In other words,  $t \in \text{FOLLOW}(X)$  if there is any derivation containing  $Xt$



# Computing FIRST, FOLLOW, and NULLABLE

---

- Algorithm:
- For each terminal symbol  $Z$ ,  $FIRST(Z) = \{Z\}$
- For each production  $X \rightarrow Y_1 Y_2 \dots Y_k$ 
  - if  $y_1 \dots y_k$  are all nullable or if  $k = 0$ 
    - then  $NULLABLE(X) = \text{true}$
  - For each  $i$  from 1 to  $k$ , each  $j$  from  $i+1$  to  $k$ 
    - if  $Y_1 \dots Y_{i-1}$  are all nullable or if  $i = 1$ 
      - then  $FIRST(X) = FIRST(X) \cup FIRST(Y_i)$
    - if  $Y_{i+1} \dots Y_k$  are all nullable or if  $i = k$ 
      - then  $FOLLOW(Y_i) = FOLLOW(Y_i) \cup FOLLOW(X)$
    - if  $Y_{i+1} \dots Y_{j-1}$  are all nullable or if  $i+1 = j$ 
      - then  $FOLLOW(Y_i) = FOLLOW(Y_i) \cup FIRST(Y_j)$

# Computing FIRST, FOLLOW, and NULLABLE

$Z \rightarrow d$	$Y \rightarrow \epsilon$	$X \rightarrow Y$
$Z \rightarrow XYZ$	$Y \rightarrow c$	$X \rightarrow a$

Step 1

	NULLABLE	FIRST	FOLLOW
X	False		
Y	False		
Z	False		

Step 2

	NULLABLE	FIRST	FOLLOW
X	False	{a}	{c,d}
Y	True	{c}	{d}
Z	False	{d}	

Step 3

	NULLABLE	FIRST	FOLLOW
X	True	{a,c}	{a,c,d}
Y	True	{c}	{a,c,d}
Z	False	{a,c,d}	



# Computing PREDICT

---

- We can extend FIRST and NULLABLE to strings of symbols
  - $\text{NULLABLE}(s_1..s_n) = \text{true}$  iff  $\text{NULLABLE}(s_i)$  for  $1 \leq i \leq n$
  - Given grammar symbol  $X$  and possibly empty string of symbols  $\gamma$ 
    - $\text{FIRST}(X \gamma) = \text{FIRST}(X)$  if not  $\text{NULLABLE}(X)$
    - $\text{FIRST}(X \gamma) = \text{FIRST}(X) \cup \text{FIRST}(\gamma)$  if  $\text{NULLABLE}(X)$
- $\text{PREDICT}(A \rightarrow \gamma) = \text{FIRST}(\gamma) \cup \text{FOLLOW}(A)$  if  $\text{NULLABLE}(\gamma)$
- $\text{PREDICT}(A \rightarrow \gamma) = \text{FIRST}(\gamma)$  if not  $\text{NULLABLE}(\gamma)$

# Computing PREDICT

Input Grammar

$Z \rightarrow d$	$Y \rightarrow \epsilon$	$X \rightarrow Y$
$Z \rightarrow XYZ$	$Y \rightarrow c$	$X \rightarrow a$

	NULLABLE	FIRST	FOLLOW
X	True	{a,c}	{a,c,d}
Y	True	{c}	{a,c,d}
Z	False	{a,c,d}	

	PREDICT
$X \rightarrow Y$	{a,c,d}
$X \rightarrow a$	{a}
$Y \rightarrow \epsilon$	{a,c,d}
$Y \rightarrow c$	{c}
$Z \rightarrow d$	{d}
$Z \rightarrow XYZ$	{a,c,d}

- Predict set tells us which set of look ahead tokens select a production right hand side
- This grammar is NOT LL(1) because there are duplicate tokens in the predict sets for all three non-terminals
  - Highlighted in red



# Parsers and Parse Trees

---

- The sample parsers that we've looked at so far are recognizers only
  - Determine if an input is syntactically correct, but don't build a parse tree
- How do we construct a parse tree then?
  - In recursive descent parsers, make each non-terminal function:
    - Construct a correct parse tree node for itself with links to its children
    - Return the constructed parse tree node to the caller

# Parse Trees and Recursive Descent Parsers

- Each non-terminal subroutine constructs and returns a parse tree node
  - Example:
    - factor -> ( expr )
    - factor -> [ sexpr ]
  - Not all tokens become parse tree nodes
    - Example: '(', ')', '[', ']'
  - This kind of parse tree is called an abstract syntax tree (AST)

```
node *factor (void) {
    switch(next_token()) {
        case '(' :
            node = factor_node(expr());
            match(')'); break;
        case '[' :
            node = factor_node(sexpr());
            match(']'); break;
    }
    return node;
}
```

# Parser Generators and Parsing

- Parser generators produce a parser from the context free grammar describing a programming language
- Productions may have **semantic actions** attached
  - When parser recognizes a production, the semantic action is called
  - Mainly used for explicit construction of parse trees
- Parser generator output is a high-level language program (e.g., C, C++, or Java) which takes a stream of tokens from a lexical analyzer and which produces a parse tree for subsequent compiler phases

# CUP specification for a simple expression evaluator

```
// An expression evaluator grammar
import java_cup.runtime.*;
```

```
/* Preliminaries to set up and use the scanner. */
init with { : scanner.init(); :};
scan with { : return scanner.next_token(); :};
```

```
/* Terminals (tokens returned by scanner). */
terminal SEMI, PLUS, MINUS;
terminal LPAREN, RPAREN;
terminal Integer NUMBER;
```

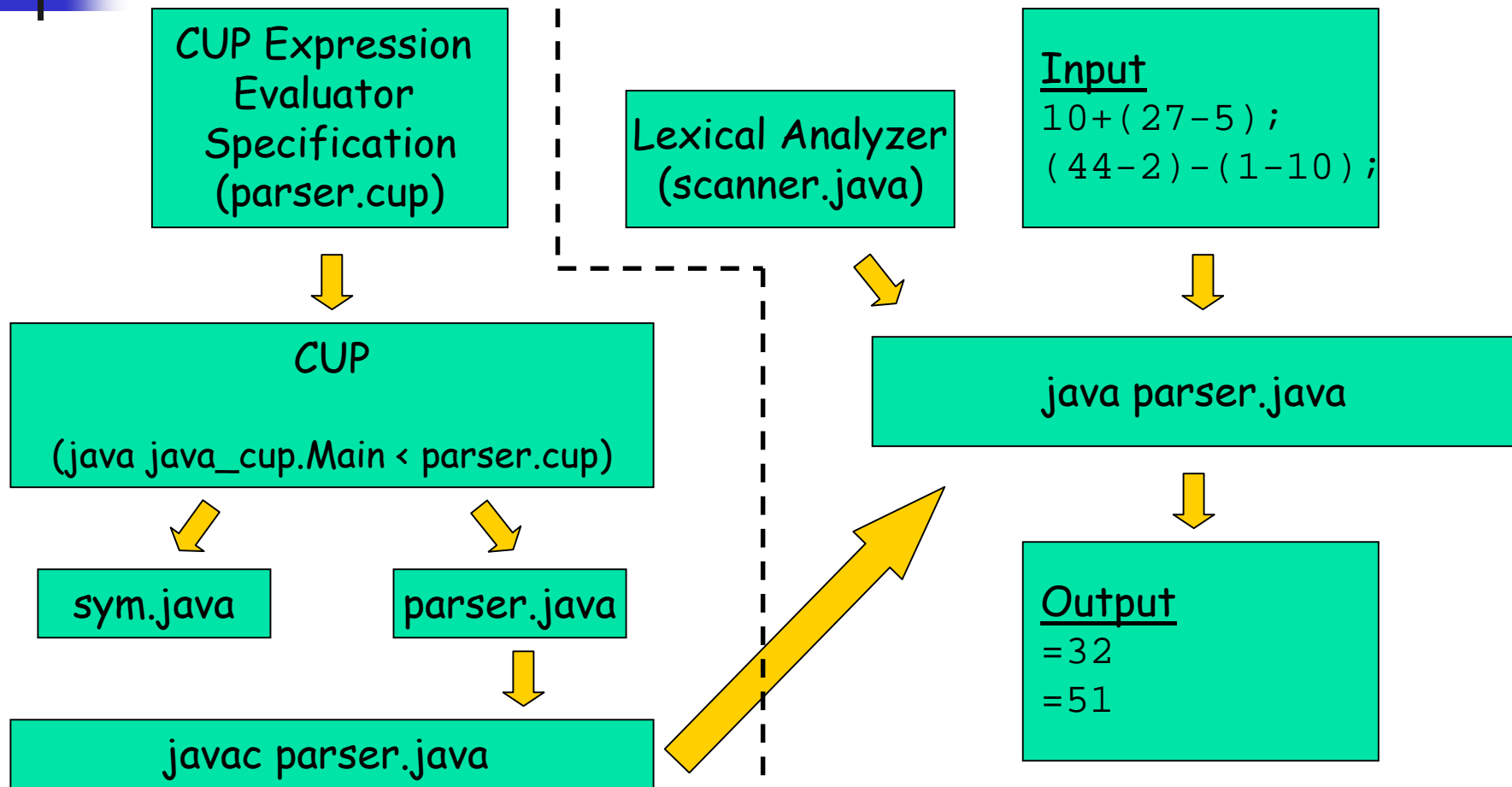
```
/* Non-terminals */
non terminal expr_list, expr_part;
non terminal Integer expr;
```

```
/* The grammar */
expr_list ::= expr_list expr_part
           |
           expr_part;

expr_part ::= expr:e
           { : System.out.println("=" + e); :}
           SEMI
           ;

expr ::= expr:e1 PLUS expr:e2
      { : RESULT = new Integer(e1.intValue() + e2.intValue()); :}
      |
      expr:e1 MINUS expr:e2
      { : RESULT = new Integer(e1.intValue() - e2.intValue()); :}
      |
      NUMBER:n
      { : RESULT = n; :}
      |
      LPAREN expr:e RPAREN
      { : RESULT = e; :}
      ;
```

# Parser Generator Example with CUP





# Roadmap of remaining lectures

---

- Next lecture
  - Review of parsing
  - Names scopes and bindings