Syntax Analysis

Mooly Sagiv

http://www.cs.tau.ac.il/~msagiv/courses/wcc08.html

Textbook:Modern Compiler Design

Chapter 2.2 (Partial)

A motivating example

- Create a desk calculator
- Challenges
  - Non trivial syntax
  - Recursive expressions (semantics)
    - Operator precedence

Solution (lexical analysis)

```
import java_cup.runtime.*;
%%
%cup
%eofval{
   return sym.EOF;
%eofval}
NUMBER=[0-9]+
%%
”+” { return new Symbol(sym.PLUS); }
”-” { return new Symbol(sym.MINUS); }
”*” { return new Symbol(sym.MULT); }
”/” { return new Symbol(sym.DIV); }
”(” { return new Symbol(sym.LPAREN); }
”)” { return new Symbol(sym.RPAREN); }
{NUMBER} {
       return new Symbol(sym.NUMBER, new Integer(yytext()));
}
\n { }
. { }
```

- Parser gets terminals from the Lexer

```
terminal Integer NUMBER;
terminal PLUS,MINUS,MULT,DIV;
terminal LPAREN, RPAREN;
terminal UMINUS;
non terminal Integer expr;
precedence left PLUS, MINUS;
precedence left DIV, MULT;
Precedence left UMINUS;
%%
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()); :}
            | expr:e1 MULT expr:e2
            {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
            | expr:e1 DIV expr:e2
            {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
            | MINUS expr:e1 %prec UMINUS
            {: RESULT = new Integer(0 - e1.intValue(); :}
            | LPAREN expr:e1 RPAREN
            {: RESULT = e1; :}
            | NUMBER:n
            {: RESULT = n; :}
```

Solution (syntax analysis)

```
// input
7 + 5 * 3
```

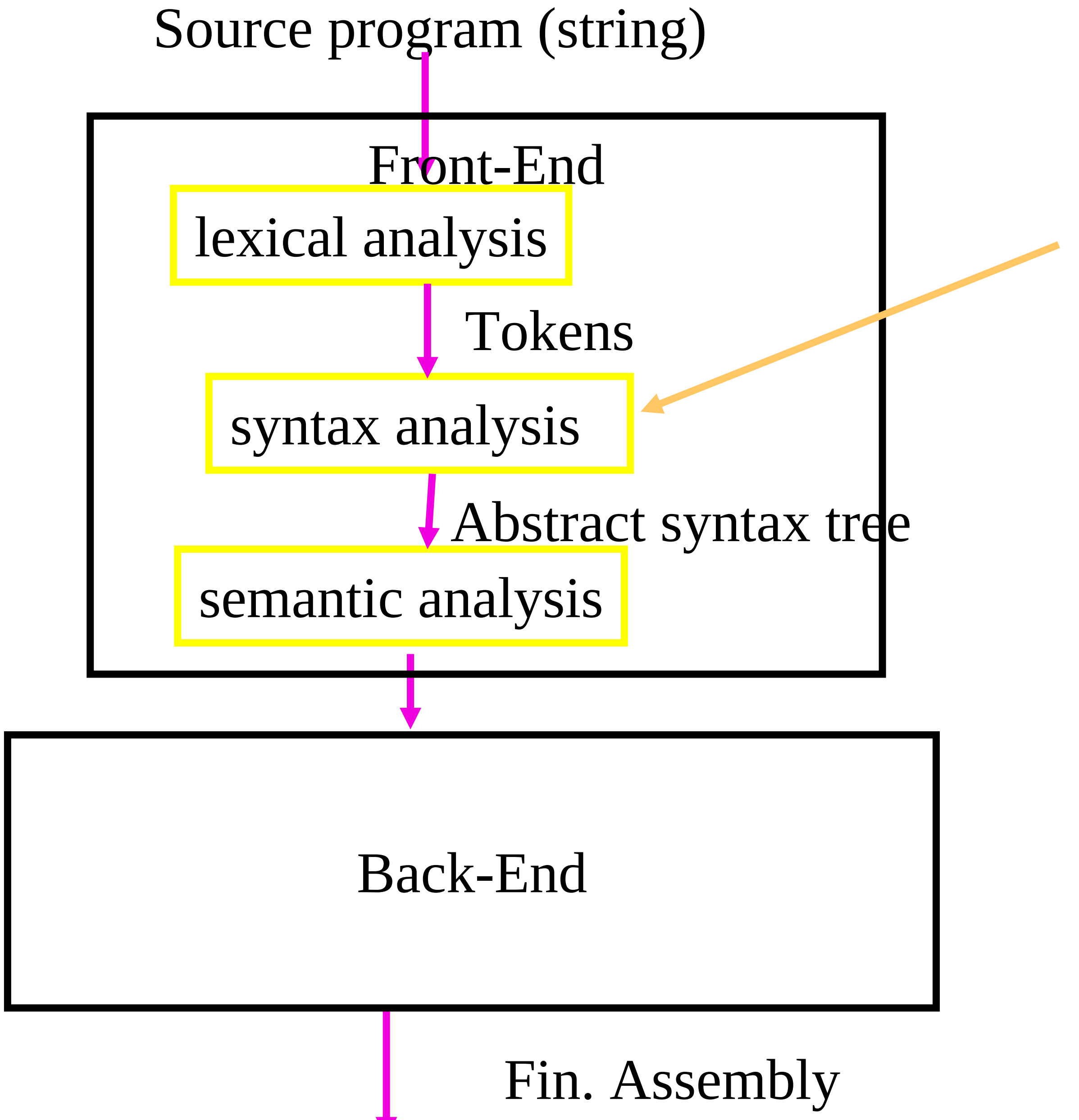
```
calc <input

22
```

Subjects

- The task of syntax analysis
- Automatic generation
- Error handling
- Context Free Grammars
- Ambiguous Grammars
- Top-Down vs. Bottom-Up parsing
- Bottom-up Parsing (next lesson)

Basic Compiler Phases



Syntax Analysis (Parsing)

- input
  - Sequence of tokens
- output
  - Abstract Syntax Tree
- Report syntax errors
    - unbalanced parenthesizes
- [Create “symbol-table”]
- [Create pretty-printed version of the program]
- In some cases the tree need not be generated (one-pass compilers)

Handling Syntax Errors

- Report and locate the error
- Diagnose the error
- Correct the error
- Recover from the error in order to discover more errors
  - without reporting too many “strange” errors

Example

a := a * (b + c * d ;

The Valid Prefix Property

- For every prefix tokens
  - $t_1, t_2, \ldots, t_i$ that the parser identifies as legal:

    - there exists tokens $t_{i+1}, t_{i+2}, \ldots, t_n$
      such that $t_1, t_2, \ldots, t_n$
      is a syntactically valid program
- If every token is considered as single character:
  - For every prefix word u that the parser identifies as legal:
    - there exists w such that
      - u.w is a valid program

Error Diagnosis

- Line number
  - may be far from the actual error
- The current token
- The expected tokens
- Parser configuration

Error Recovery

- Becomes less important in interactive environments
- Example heuristics:
  - Search for a semi-column and ignore the statement
  - Try to “replace” tokens for common errors
  - Refrain from reporting 3 subsequent errors
- Globally optimal solutions
  - For every input w, find a valid program w’ with a “minimal-distance” from w

Why use context free grammars for defining PL syntax?

- Captures program structure (hierarchy)
- Employ formal theory results
- Automatically create “efficient” parsers

Context Free Grammar (Review)

- What is a grammar
- Derivations and Parsing Trees
- Ambiguous grammars
- Resolving ambiguity

Context Free Grammars

- Non-terminals
  - Start non-terminal
- Terminals (tokens)
- Context Free Rules
  <Non-Terminal> $\rightarrow$ Symbol Symbol … Symbol

Example Context Free Grammar

1 $S \rightarrow S \; ; \; S$
2 $S \rightarrow$ id := E
3 $S \rightarrow$ print (L)
4 $E \rightarrow$ id
5 $E \rightarrow$ num
6 $E \rightarrow E + E$
7 $E \rightarrow (S, E)$
8 $L \rightarrow E$
9 $L \rightarrow L, E$

Derivations

- Show that a sentence is in the grammar (valid program)
  - Start with the start symbol
  - Repeatedly replace one of the non-terminals by a right-hand side of a production
  - Stop when the sentence contains terminals only
- Rightmost derivation
- Leftmost derivation

Example Derivations

1 $S \rightarrow S\ ;\ S$
2 $S \rightarrow \text{id} := E$
3 $S \rightarrow \text{print}\ (L)$
4 $E \rightarrow \text{id}$
5 $E \rightarrow \text{num}$
6 $E \rightarrow E + E$
7 $E \rightarrow (S, E)$
8 $L \rightarrow E$
9 $L \rightarrow L, E$

$$\underline{S}$$

$$S\ ;\ \underline{S}$$

$$\underline{S}\ ;\ \text{id} := E$$

$$\text{id} := \underline{E}\ ;\ \text{id} := E$$

$$\text{id} := \text{num}\ ;\ \text{id} := \underline{E}$$

$$\text{id} := \text{num}\ ;\ \text{id} := E + \underline{E}$$

$$\text{id} := \text{num}\ ;\ \text{id} := \underline{E} + \text{num}$$

$$\text{id} := \text{num}\ ;\ \text{id} := \text{num} + \text{num}$$

a 56 b 77 16

Parse Trees

- The trace of a derivation
- Every internal node is labeled by a non-terminal
- Each symbol is connected to the deriving non-terminal

Example Parse Tree

$\underline{S}$

S ; $\underline{S}$

$\underline{S}$; id := E

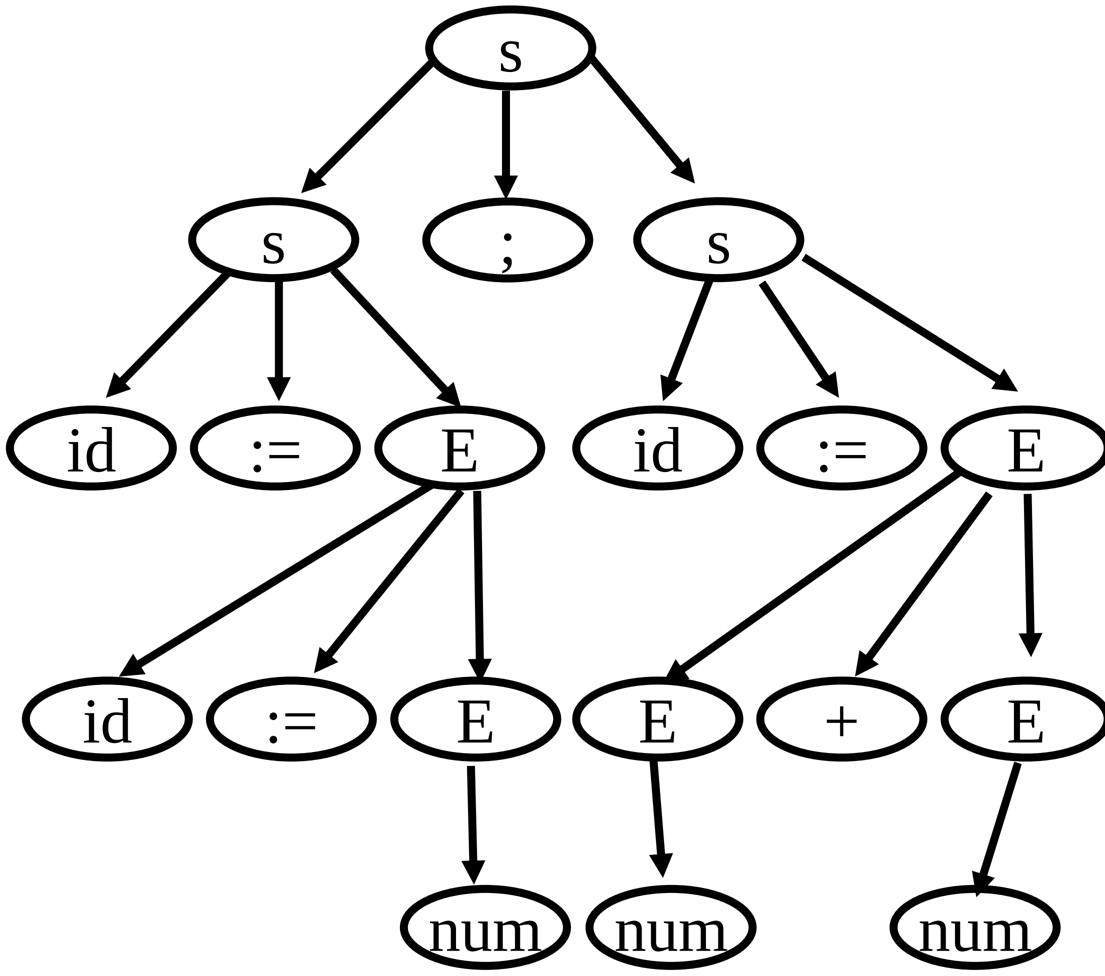
id := $\underline{E}$; id := E

id := num ; id := $\underline{E}$

id := num ; id := E + $\underline{E}$

id := num ; id := $\underline{E}$ + num

id := num ; id :=num + num



Ambiguous Grammars

- Two leftmost derivations
- Two rightmost derivations
- Two parse trees

A Grammar for Arithmetic Expressions

1 $E \rightarrow E + E$
2 $E \rightarrow E * E$
3 $E \rightarrow \text{id}$
4 $E \rightarrow (E)$

Drawbacks of Ambiguous Grammars

- Ambiguous semantics
- Parsing complexity
- May affect other phases

Non Ambiguous Grammar for Arithmetic Expressions

Ambiguous grammar

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

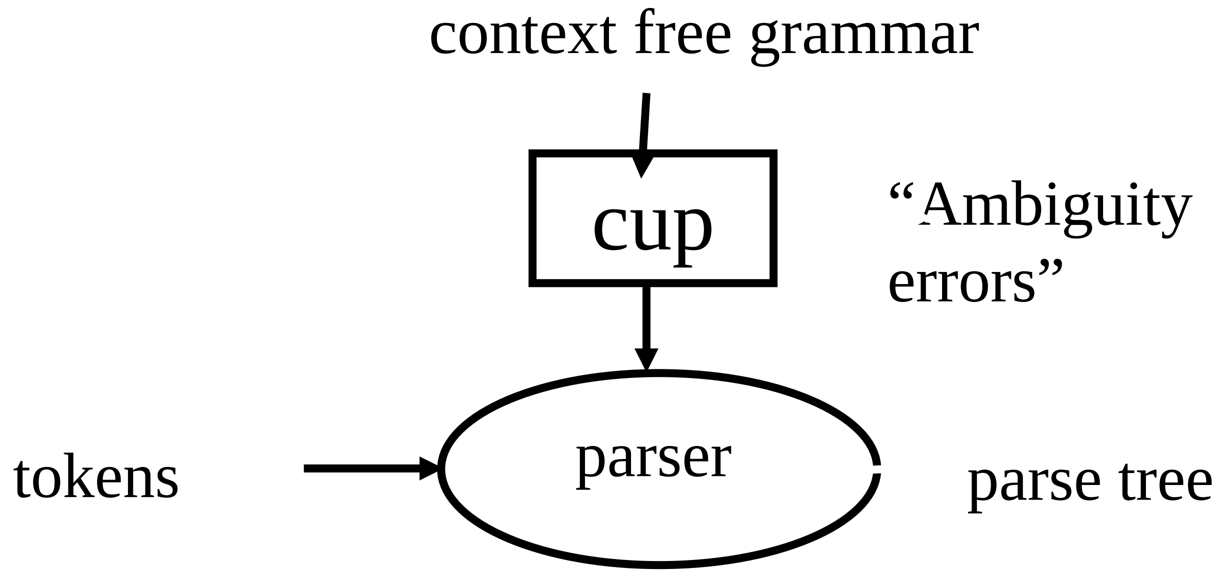
Non Ambiguous Grammars for Arithmetic Expressions

Ambiguous grammar

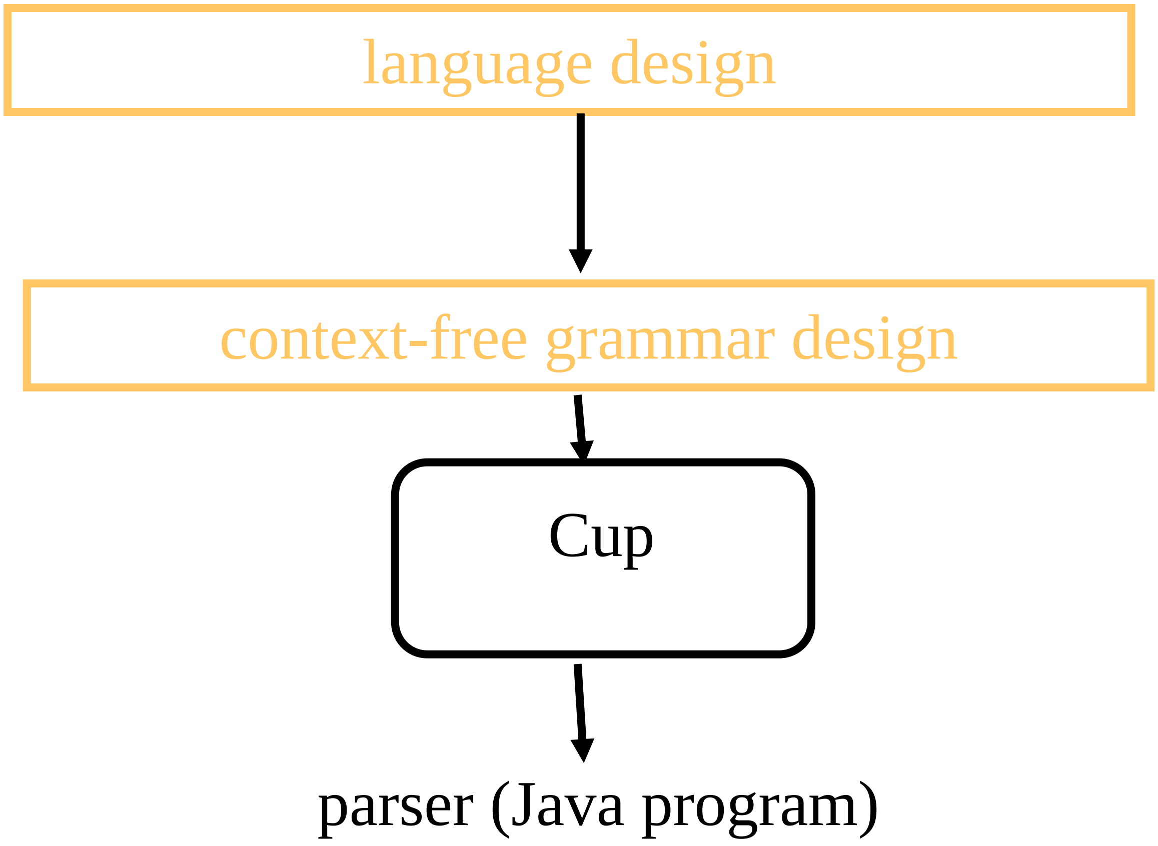
1	$E \rightarrow E + E$	1	$E \rightarrow E + T$	1	$E \rightarrow E * T$
2	$E \rightarrow E * E$	2	$E \rightarrow T$	2	$E \rightarrow T$
3	$E \rightarrow id$	3	$T \rightarrow T * F$	3	$T \rightarrow F + T$
4	$E \rightarrow (E)$	4	$T \rightarrow F$	4	$T \rightarrow F$
		5	$F \rightarrow id$	5	$F \rightarrow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

Efficient Parsers

- Pushdown automata
- Deterministic
- Report an error as soon as the input is not a prefix of a valid program
- Not usable for all context free grammars



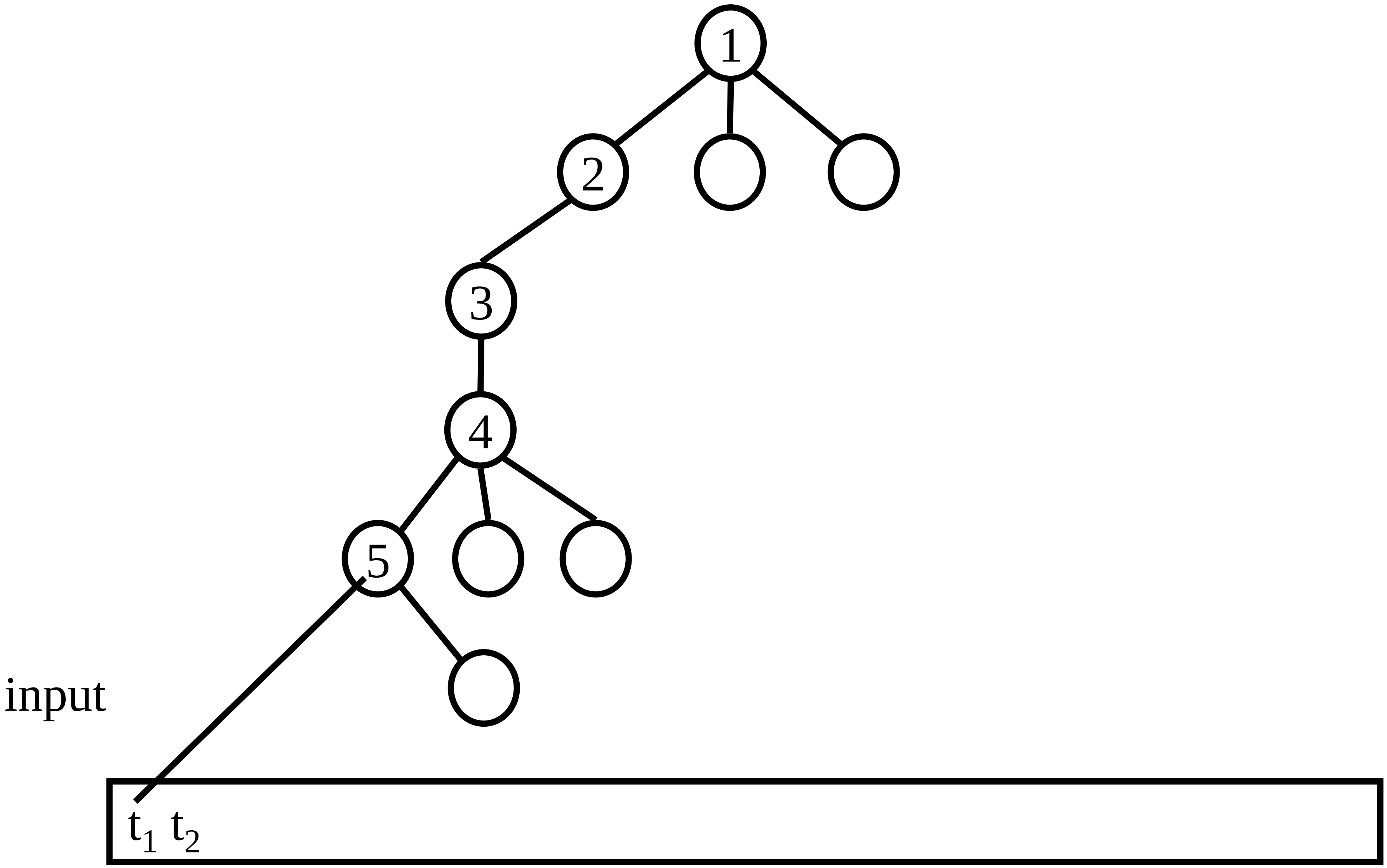
Designing a parser



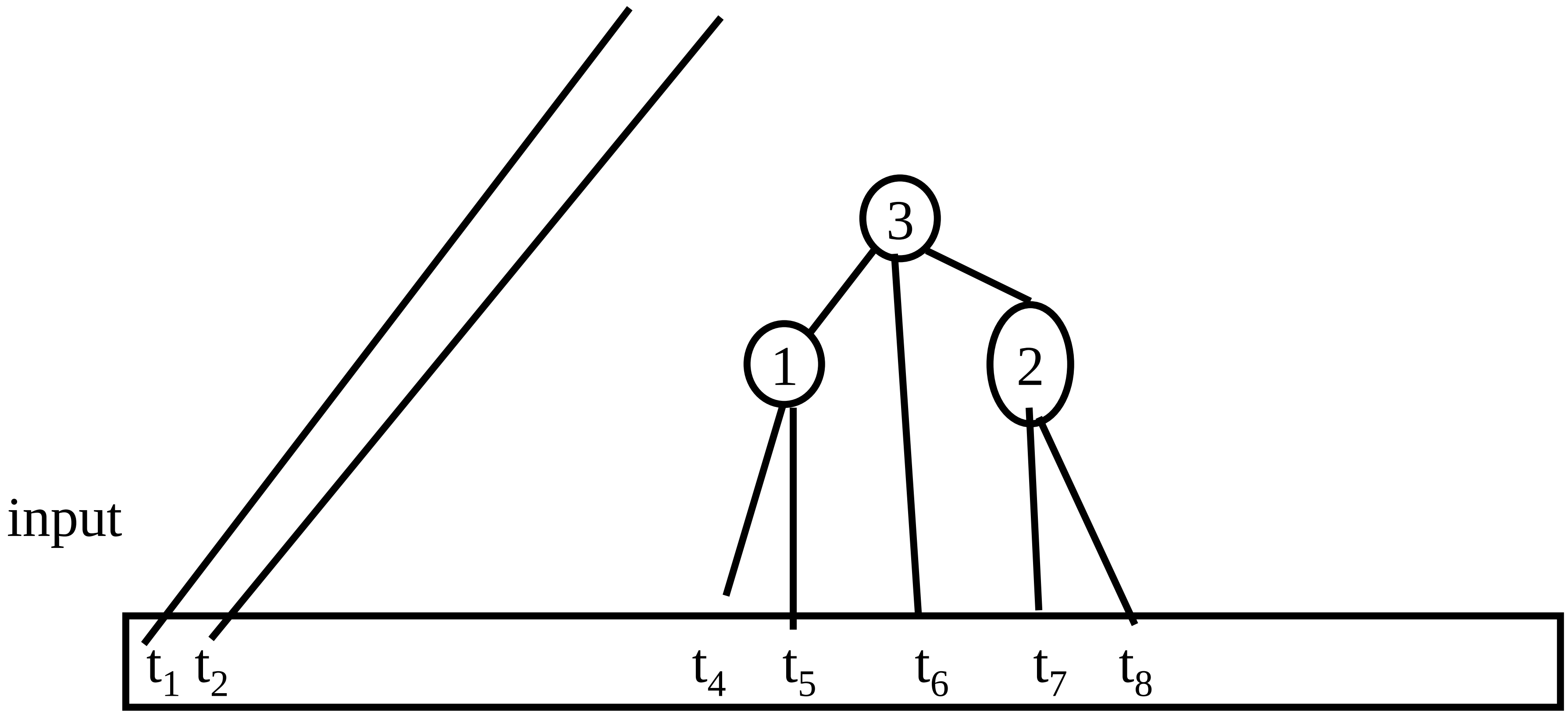
Kinds of Parsers

- Top-Down (Predictive Parsing) LL
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
  - For every non-terminal and token **predict** the next production
  - Preorder tree traversal
- Bottom-Up LR
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order
  - For every potential right hand side and token decide when a production is found
  - Postorder tree traversal

Top-Down Parsing



Bottom-Up Parsing



Example Grammar for Predictive LL Top-Down Parsing

expression $\rightarrow$ digit | ‘(‘ expression operator expression ‘)’

operator $\rightarrow$ ‘+’ | ‘*’

digit $\rightarrow$ ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

```
static int Parse_Expression(Expression **expr_p) {
  Expression *expr = *expr_p = new_expression() ;
  /* try to parse a digit */
  if (Token.class == DIGIT) {
        expr->type=‘D’;  expr->value=Token.repr –’0’;      get_next_token();
        return 1;       }
/* try parse parenthesized expression */
if (Token.class == ‘(‘) {
     expr->type=‘P’;   get_next_token();
     if (!Parse_Expression(&expr->left))  Error(“missing expression”);
     if (!Parse_Operator(&expr->oper))  Error(“missing operator”);
     if (Token.class != ‘)’) Error(“missing )”);
     get_next_token();
     return 1; }
return 0;
}
```

Parsing Expressions

- Try every alternative production
  - For $P \rightarrow A_1\ A_2 \ldots A_n \mid B_1\ B_2 \ldots B_m$
  - If A_1 succeeds
    - Call A_2
    - If A_2 succeeds
      - Call A_3
    - If A_2 fails report an error
  - Otherwise try B_1
- Recursive descent parsing
- Can be applied for certain grammars
- Generalization: LL1 parsing

```
int P(...) {
   /* try parse the alternative  P $\rightarrow$ $A_1$ $A_2$ ... $A_n$ */
   if ($A_1$(...)) {
      if (!$A_2$()) Error(“Missing $A_2$”);
      if (!$A_3$()) Error(“Missing $A_3$”);
      ..
      if (!$A_n$()) Error(Missing $A_n$”);
      return 1;
                 }
/* try parse the alternative  P $\rightarrow$ $B_1$ $B_2$ ... $B_m$ */
   if ($B_1$(...)) {
      if (!$B_2$()) Error(“Missing $B_2$”);
      if (!$B_3$()) Error(“Missing $B_3$”);
      ..
      if (!$B_m$()) Error(Missing $B_m$”);
      return 1;
                 }
      return 0;
```

Predictive Parser for Arithmetic Expressions

- Grammar

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

- C-code?

Summary

- Context free grammars provide a natural way to define the syntax of programming languages
- Ambiguity may be resolved
- Predictive parsing is natural
  - Good error messages
  - Natural error recovery
  - But not expressive enough
- But LR bottom-up parsing is more expressible