Syntax Analysis

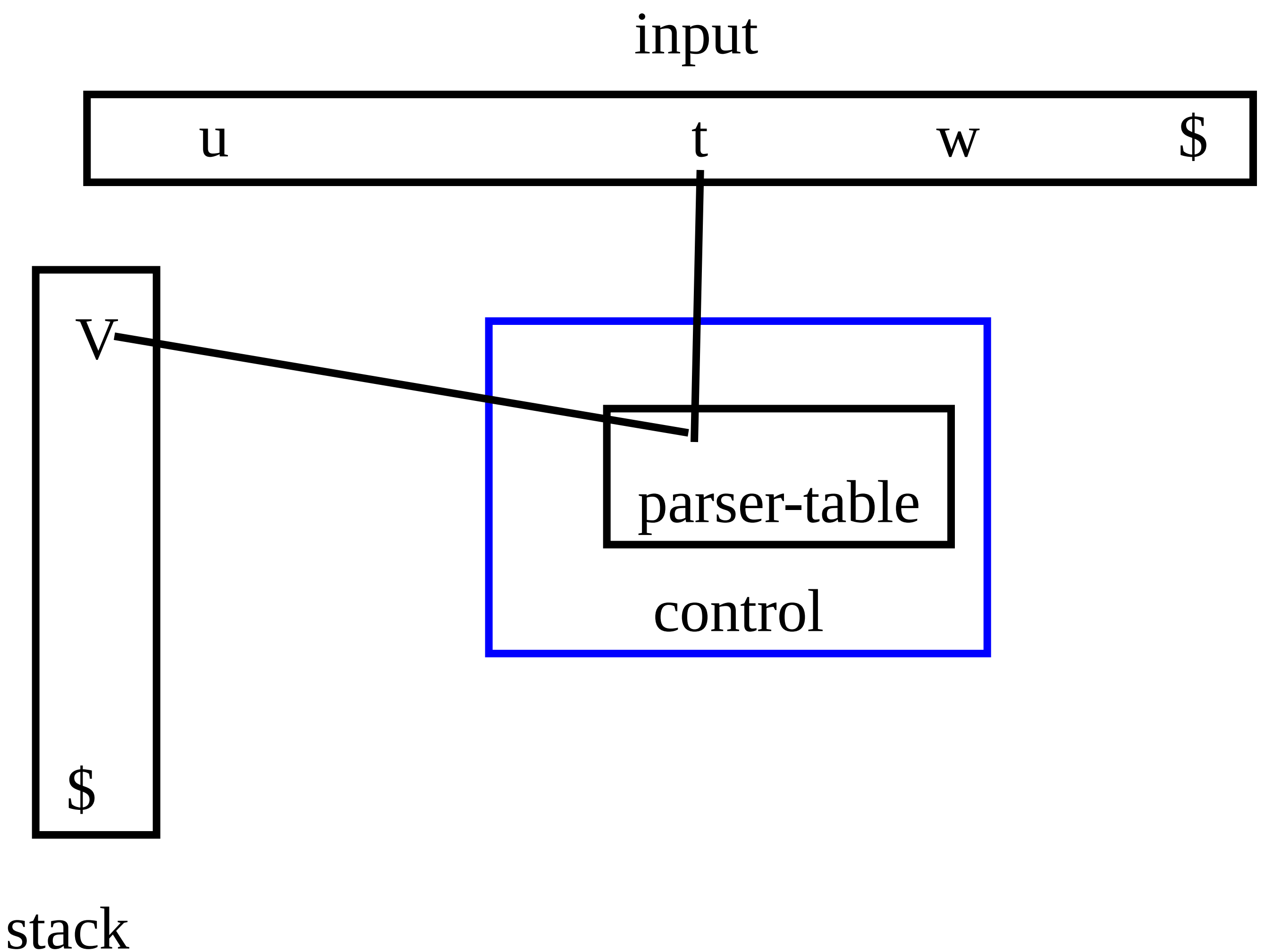
Mooly Sagiv

http://www.cs.tau.ac.il/~msagiv/courses/wcc13.html

Textbook:Modern Compiler Design

Chapter 2.2 (Partial)

Pushdown Automaton



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

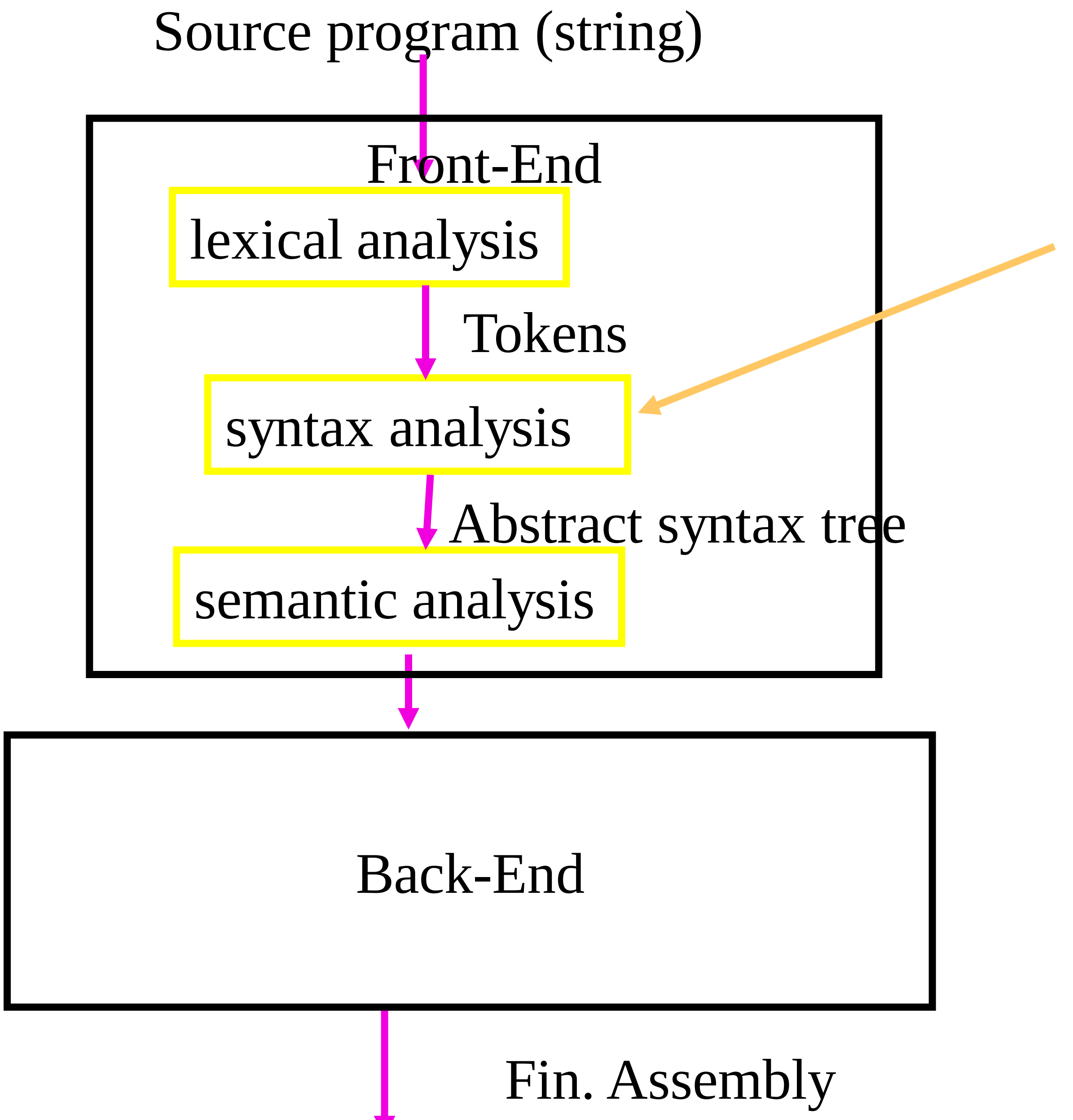
calc <input

22

Subjects

- The task of syntax analysis
- Automatic generation
- Error handling
- Context Free Grammars
- Ambiguous Grammars
- Top-Down vs. Bottom-Up parsing
- Simple Top-Down 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> → Symbol Symbol … Symbol

Example Context Free Grammar

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

S

S ; S

S ; id := E

id := E ; id := E

id := num ; id := E

id := num ; id := E + E

id := num ; id := E + num

id := num ; id :=num + 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

S

S ; S

S ; id := E

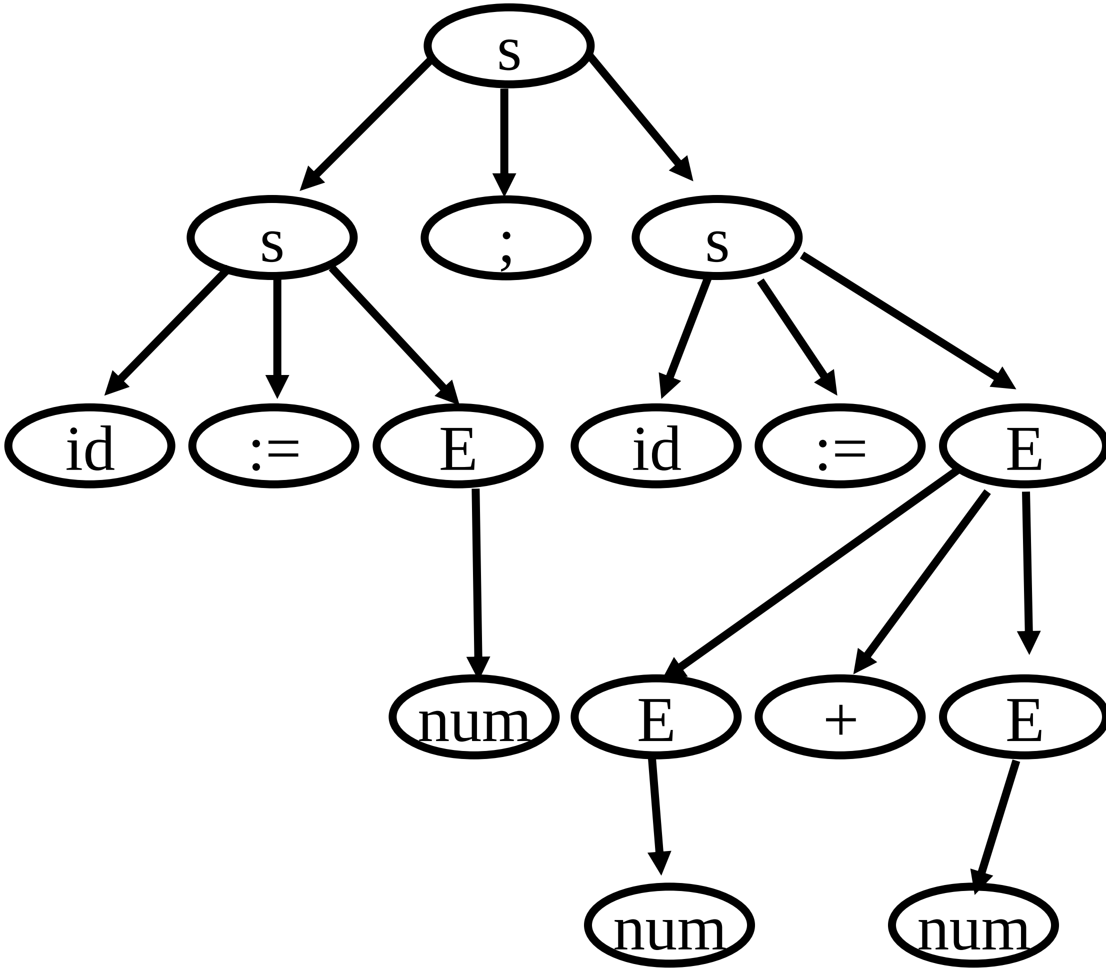
id := E ; id := E

id := num ; id := E

id := num ; id := E + E

id := num ; id := 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 $E \rightarrow id$
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 $E \rightarrow E + E$
2 $E \rightarrow E * E$
3 $E \rightarrow id$
4 $E \rightarrow (E)$

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

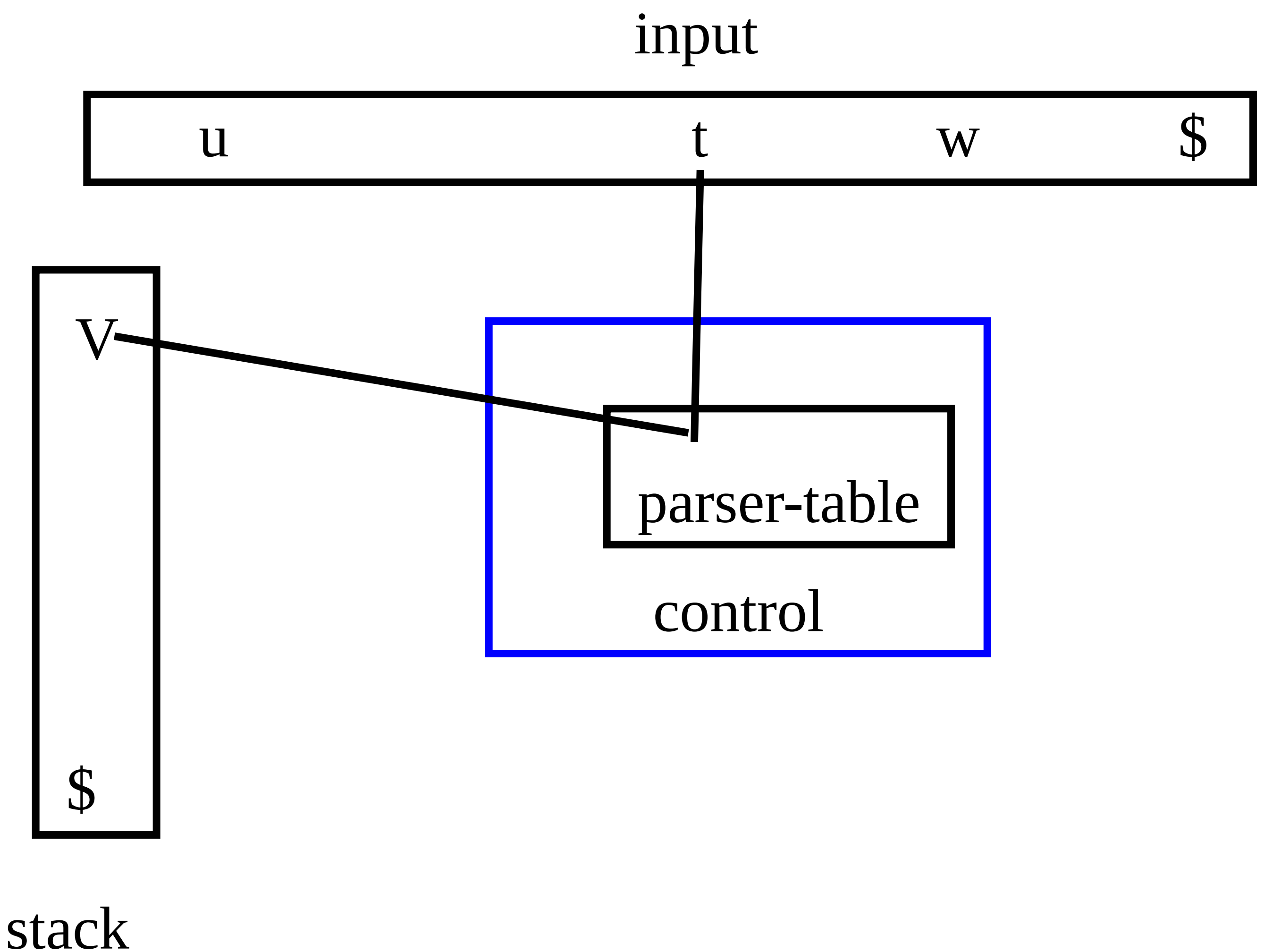
Non Ambiguous Grammars for Arithmetic Expressions

Ambiguous grammar

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

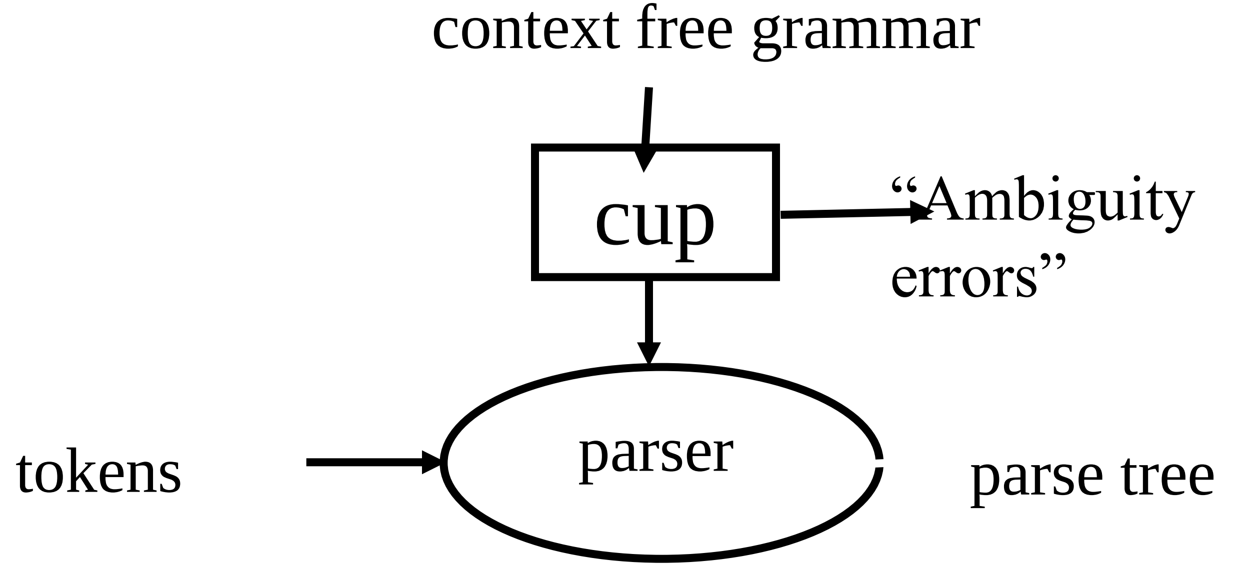
1	$E \rightarrow E + T$	1	$E \rightarrow E * T$
2	$E \rightarrow T$	2	$E \rightarrow T$
3	$T \rightarrow T * F$	3	$T \rightarrow F + T$
4	$T \rightarrow F$	4	$T \rightarrow F$
5	$F \rightarrow id$	5	$F \rightarrow id$
6	$F \rightarrow (E)$	6	$F \rightarrow (E)$

Pushdown Automaton

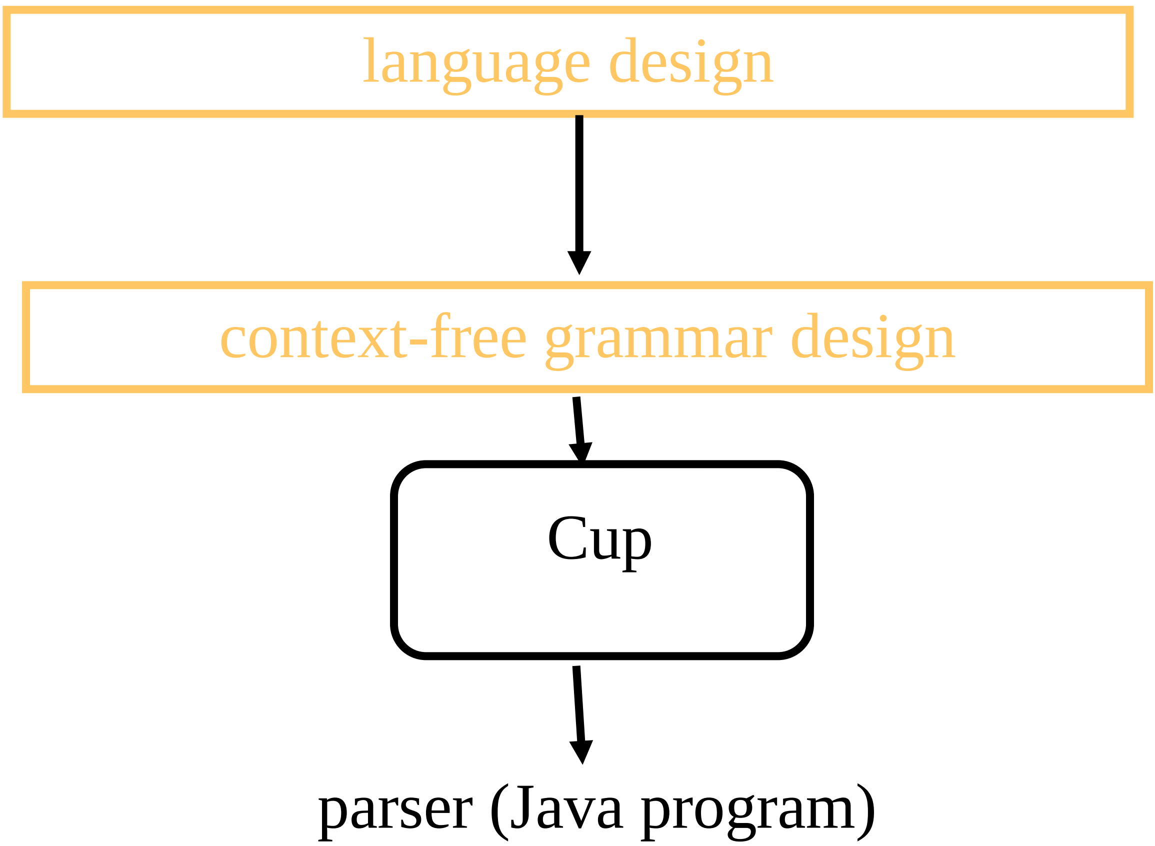


Efficient Parsers

- 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



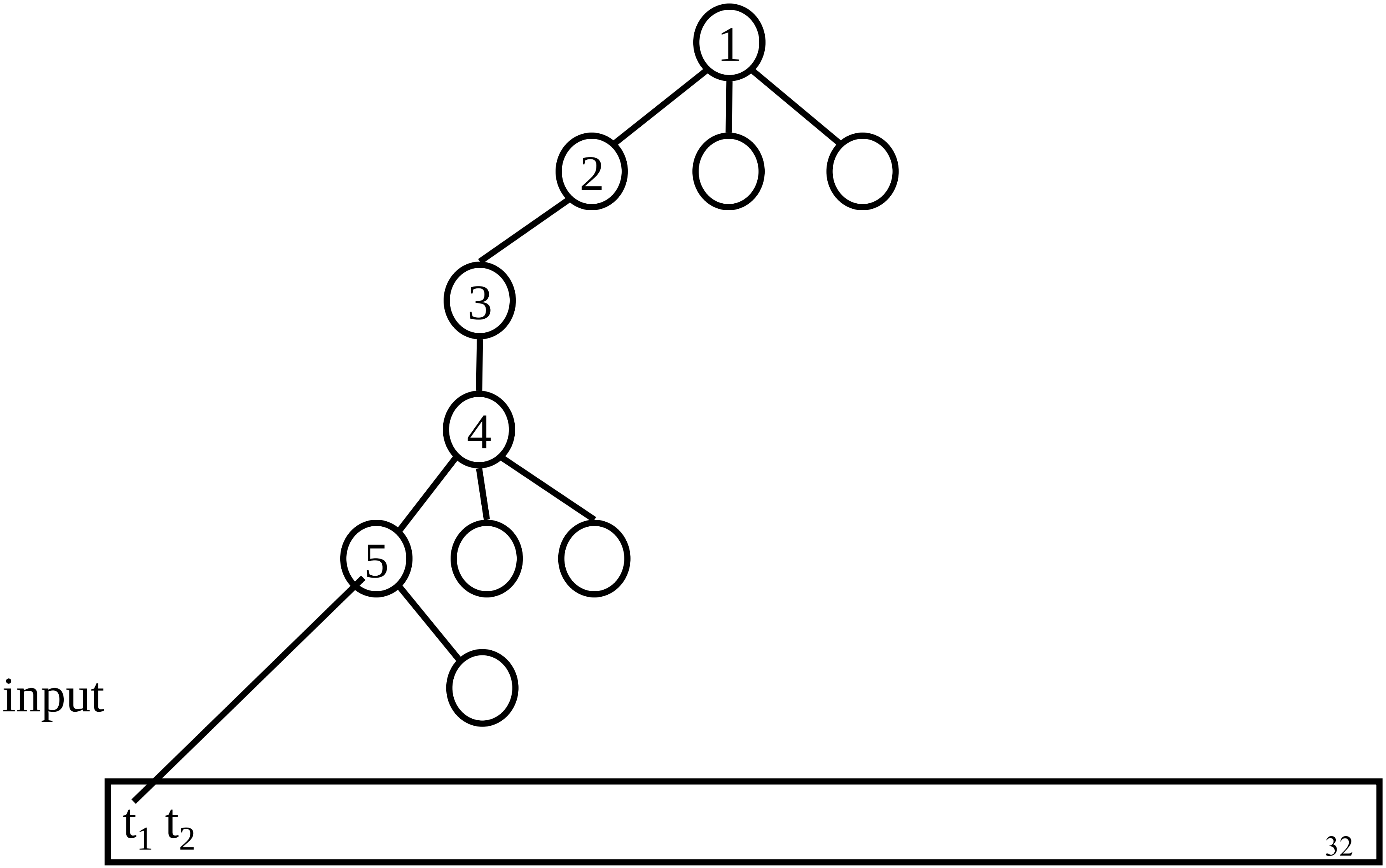
Designing a parser



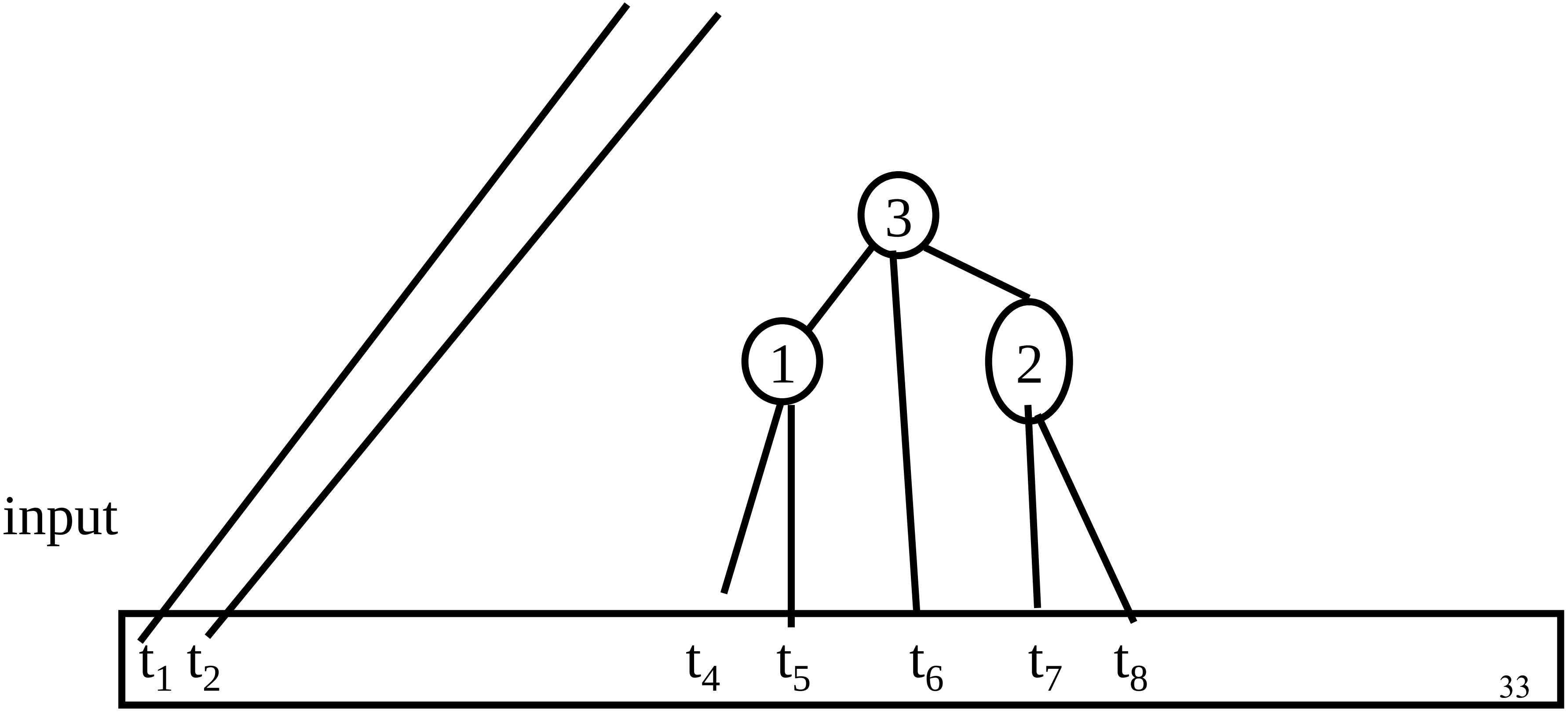
Kinds of Parsers

- 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 → digit | ‘(‘ expression operator expression ‘)’

operator → ‘+’ | ‘*’

digit → ‘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 $P \rightarrow A_1\ A_2\ \dots\ A_n \mid B_1\ B_2\ \dots\ B_m$
  - 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

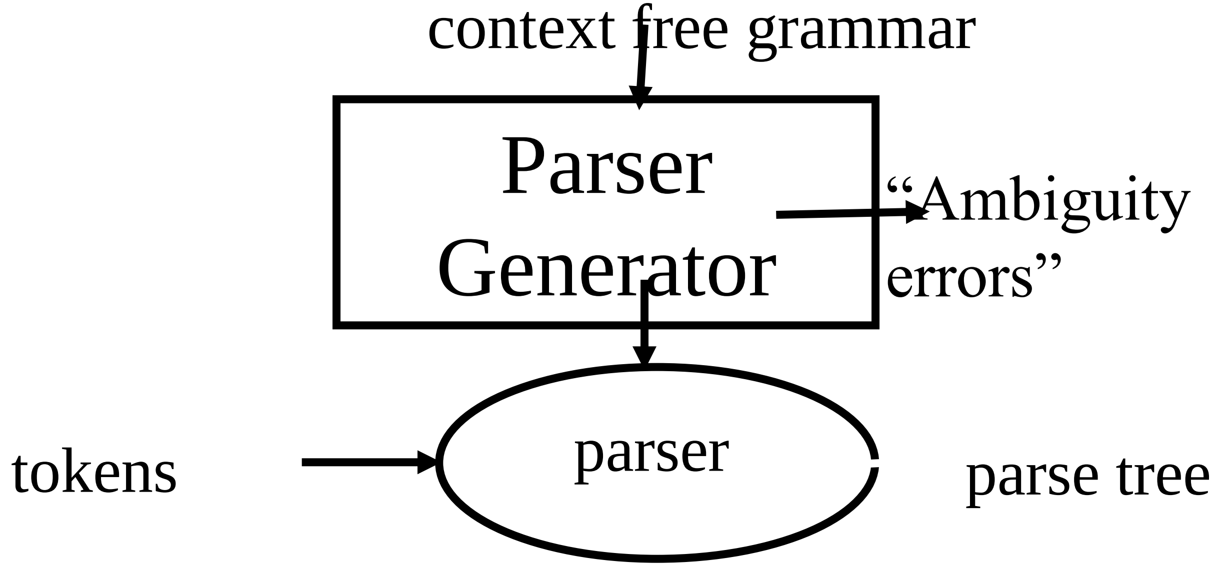
```
int P(...) {
   /* try parse the alternative  P → A1 A2 ... An */
   if (A1(...)) {
      if (!A2()) Error("Missing A2");
      if (!A3()) Error("Missing A3");
      ..
      if (!An()) Error(Missing An");
      return 1;
             }
/* try parse the alternative  P → B1 B2 ... Bm */
   if (B1(...)) {
      if (!B2()) Error("Missing B2");
      if (!B3()) Error("Missing B3");
      ..
      if (!Bm()) Error(Missing Bm");
      return 1;
             }
      return 0;
```

Bad Example for Top-Down Parsing

$S \rightarrow A\ c \mid B\ d$
$A \rightarrow a$
$B \rightarrow a$

Efficient Top-Down Parsers

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



Top-Down Parser Generation

- First assume that all non-terminals are not nullable
  - No possibility for $A \rightarrow^* \varepsilon$
- Define for every string of grammar symbols α
  - $First(\alpha) = \{ t \mid \exists\beta: \alpha \rightarrow^* t\beta \}$
- The grammar is LL(1) if for every two grammar rules $A \rightarrow \alpha$ and $A \rightarrow \beta$
  - $First(\alpha) \cap First(\beta) = \emptyset$

Bad Example for Top-Down Parsing

S → A c | B d
A → a
B → a

α	First(α)
a	{a}
c	{c}
d	{d}
A	{a}
B	{a}
S	{a}
Ac	{a}
Bd	{a}

Computing First Sets

- For tokens t, define First(t) = {t}
- For Non-terminals A, defines First(A) inductively
  - If A $\rightarrow$ Vα then First(V) $\subseteq$ First(A)
- Can be computed iteratively
- For α = V β define

  First(α) = First(V)

Computing First Iteratively

For each token t, First(t) := {t}
For each non-terminal A, First(A) := {}

while changes occur do
    if there exists a non-terminal A and
        a rule A → Vα and
        a token t $\in$ First(V) and
        t $\notin$ First(A)
    add t to First(A)

A Simple Example

$E \rightarrow (E) \mid ID$

Constructing Top-Down Parser

- Construct First sets
- If the grammar is not LL(1) report an error
- Otherwise construct a predictive parser
- A procedure for every non-terminals
- For tokens $t \in First(\alpha)$ apply the rule $A \rightarrow \alpha$
  - Otherwise report an error

Handling Nullable Non-Terminals

- Which tokens predicate empty derivations?
- For a non-terminal A define
  - Follow(A) = $\{ t \mid \exists\beta: S \rightarrow^{*} At\beta \}$
- Follow can be computed iteratively
- First need to be updated too

Handling Nullable Non-Terminals

- For a non-terminal A define
  - Follow(A) = $\{ t \mid \exists\beta: S \rightarrow^* At\beta \}$
- For a rule $A \rightarrow \alpha$
  - If α is nullable then
    select($A\rightarrow \alpha$) = First(α) $\cup$ Follow(A)
    - Otherwise select($A\rightarrow \alpha$) = First(α)
- The grammar is LL(1) if for every two grammar rules $A \rightarrow \alpha$ and $A \rightarrow \beta$
  - Select($A \rightarrow \alpha$) $\cap$ Select($A \rightarrow \beta$) = $\emptyset$

Computing First For Nullable Non-Terminals

For each token t, First(t) := {t}
For each non-terminal A, First(A) = {}

while changes occur do
    if there exists a non-terminal A and
        a rule $A \rightarrow V_1 V_2 \ldots V_n \alpha$ and
        $V_1, V_2, \ldots, V_{n-1}$ are nullable
        a token $t \in First(V_n)$ and
        $t \notin First(A)$
    add t to First(A)

An Imperative View

- Create a table with
  #Non-Terminals $\times$ #Tokens
  Entries
- If $t \in$ select(A $\rightarrow$ α) apply the rule "apply the rule A $\rightarrow$ α"
- Empty entries correspond to syntax errors

A Simple Example

E → (E) | ID

	(	)	ID	$
E	E → (E)		E→id	

Left Recursion

- Left recursive grammar is never LL(1)
  - A → Aa | b
- Convert into a right-recursive grammar
- Can be done for a general grammar
- The resulting grammar may or may not be LL(1)

Predictive Parser for Arithmetic Expressions

- Grammar

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

- C-code?

Computing First Sets for Expression Grammar

- Grammar

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

Modified First Sets
First(F) = {id}
First(F) = {id, (}
First(T) = {id, (}
First(E)={id, (}
First(E+T) = {id, (}
First(T*F) = {id, (}

Rule	Select
1	{id, (}
2	{id, (}
3	{id, (}
4	{id, (}
5	id
6	(

Select sets for modified expression grammar

1 E → T E’
2 E’ → ε
3 E’ →+ T E’
4 *T → F T’*
5 T’ → ε
6 T’ → * F T’
7 F → id
8 F → (E)

First Sets
First(F) = {id, (}
First(T) = {id, (}
First(E)={id, (}
First(E’) = {+}
First(T’) = {*}
First(T E’)= {id, (}
First(ε) = {}
First(+ T E’) = {+}
First(F T) = {id, (}
First(* F T’) = {*}
First(id) = {id}
First((E)) = {(}

Follow Sets
Follow(E) = {$,)}
Follow(E’) = {$,)}
Follow(T) = {+,), $}
Follow(T’)={+,), $}
Follow(F) = {*, +,), $}

Rule	Select
1	{id, (}
2	{$,)}
3	{+}
4	{id, (}
5	{ +,), $}
6	{*}
7	{id}
8	{(}

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
- Bttom-up parsing is more expressible