

# Formal Syntax of Programming Languages

Mooly Sagiv

<http://www.cs.tau.ac.il/~msagiv/courses/pl16.html>

# Benefits of formal definitions

- Intellectual
- Better understanding
- Formal proofs
- Mechanical checks by computer
- Tool generation
  - Consistency
  - Entailment
  - Query evaluation

# What is a good formal definition?

- Natural
- Concise
- Easy to understand
- Permits effective mechanical reasoning

# Syntax vs. Semantics

- The pattern of formation of sentences or phrases in a language
- Examples
  - Regular expressions
  - Context free grammars
- The study or science of meaning in language
- Examples
  - Interpreter
  - Compiler
  - Better mechanisms will be given in the course

# Benefits of formal syntax for programming language

- Intellectual
- Simplicity
- Better understanding
  - Interaction between different parts
- Abstraction
  - Portability
- Tool generations
  - Parser

# Tokens

- Basic units of the programming language
- Usually defined by regular expressions
- Examples
  - Keywords
    - “if”
    - “while”
  - Identifier  $\{\text{letter}\}(\{\text{letter}\}|\{\text{digit}\}|\_)*$
  - Numbers  $\{0-9\}^+$

# Example Tokens

Type	Examples
ID	foo n_14 last
NUM	73 00 517 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

# Example Non Tokens

Type	Examples
comment	<code>/* ignored */</code>
preprocessor directive	<code>#include &lt;foo.h&gt;</code>
	<code>#define NUMS 5, 6</code>
macro	<code>NUMS</code>
whitespace	<code>\t \n \b</code>



# Recursive Syntax Definitions

- The syntax of programming languages is naturally defined recursively
- Valid program are represented as syntax trees

# Expression Definitions

- Every **identifier** is an **expression**
- If E1 and E2 are **expressions** and **op** is a binary operation then so is '**E<sub>1</sub> op E<sub>2</sub>**' is an **expression**

$$\langle E \rangle \rightarrow \text{id} \mid \langle E \rangle \langle \text{op} \rangle \langle E \rangle$$
$$\langle \text{op} \rangle \rightarrow + \mid - \mid * \mid /$$

# Statement Definitions

- If **id** is a **identifier** and **E** is an expression then '**id := E**' is a **statement**
- If **S<sub>1</sub>** and **S<sub>2</sub>** are statements and **E** is an expression then
  - '**S<sub>1</sub> ; S<sub>2</sub>**' is a statement
  - '**if (E) S<sub>1</sub> else S<sub>2</sub>**' is a statement

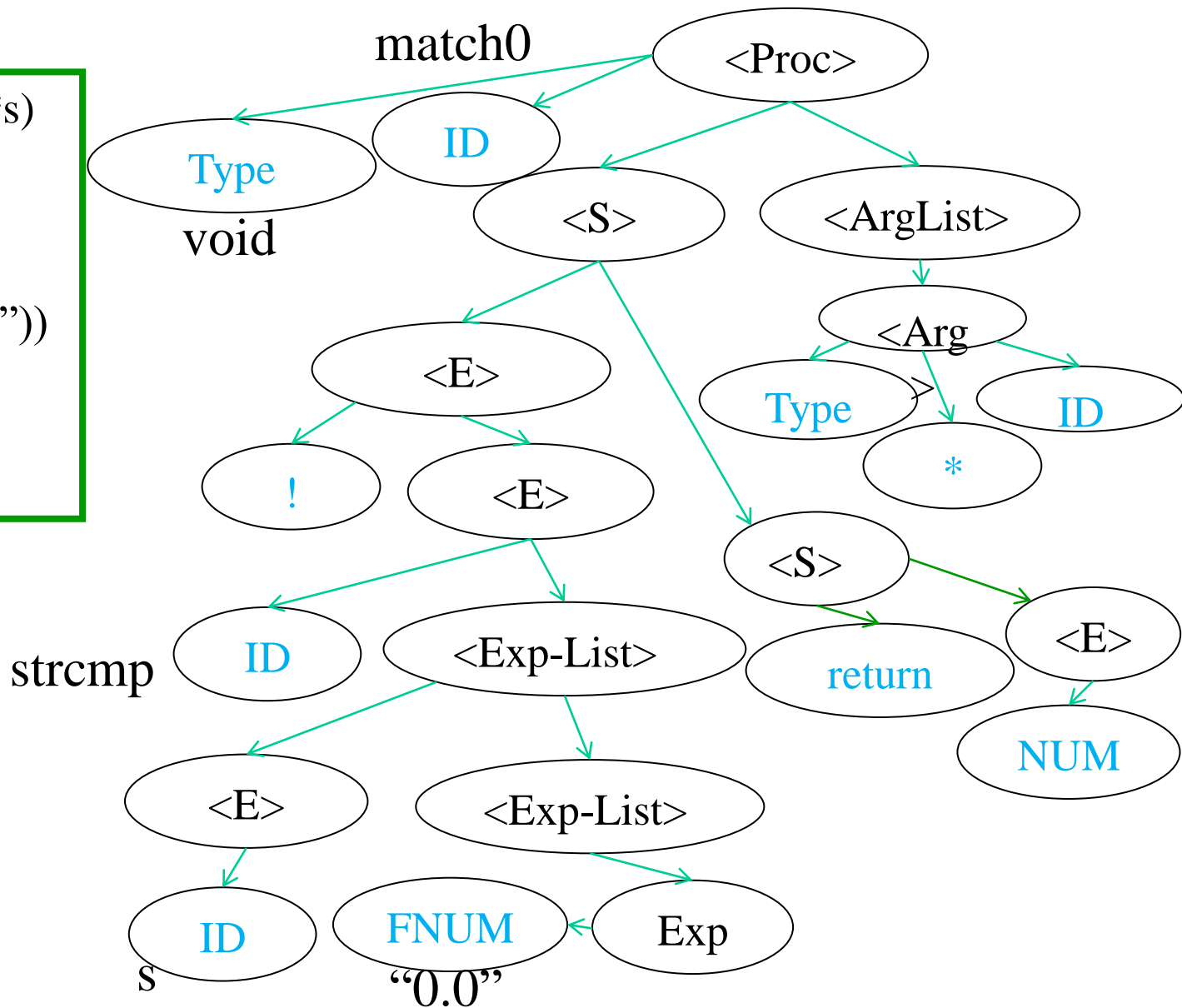
$\langle S \rangle \rightarrow \text{id} := \langle E \rangle$

$\langle S \rangle \rightarrow \langle S \rangle ; \langle S \rangle$

$\langle S \rangle \rightarrow \text{if} (\langle E \rangle) \langle S \rangle \text{ else } \langle S \rangle$

# C Example

```
void match0(char *s)
/* find a zero */
{
  if (!strcmp(s, "0.0"))
    return 0 ;
}
```



# Context Free Grammars

- Non-terminals
  - Start non-terminal
- Terminals (tokens)
- Context Free Rules  
 $\langle \text{Non-Terminal} \rangle \rightarrow \text{Symbol Symbol} \dots \text{Symbol}$

# Example Context Free Grammar

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

# 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

# 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

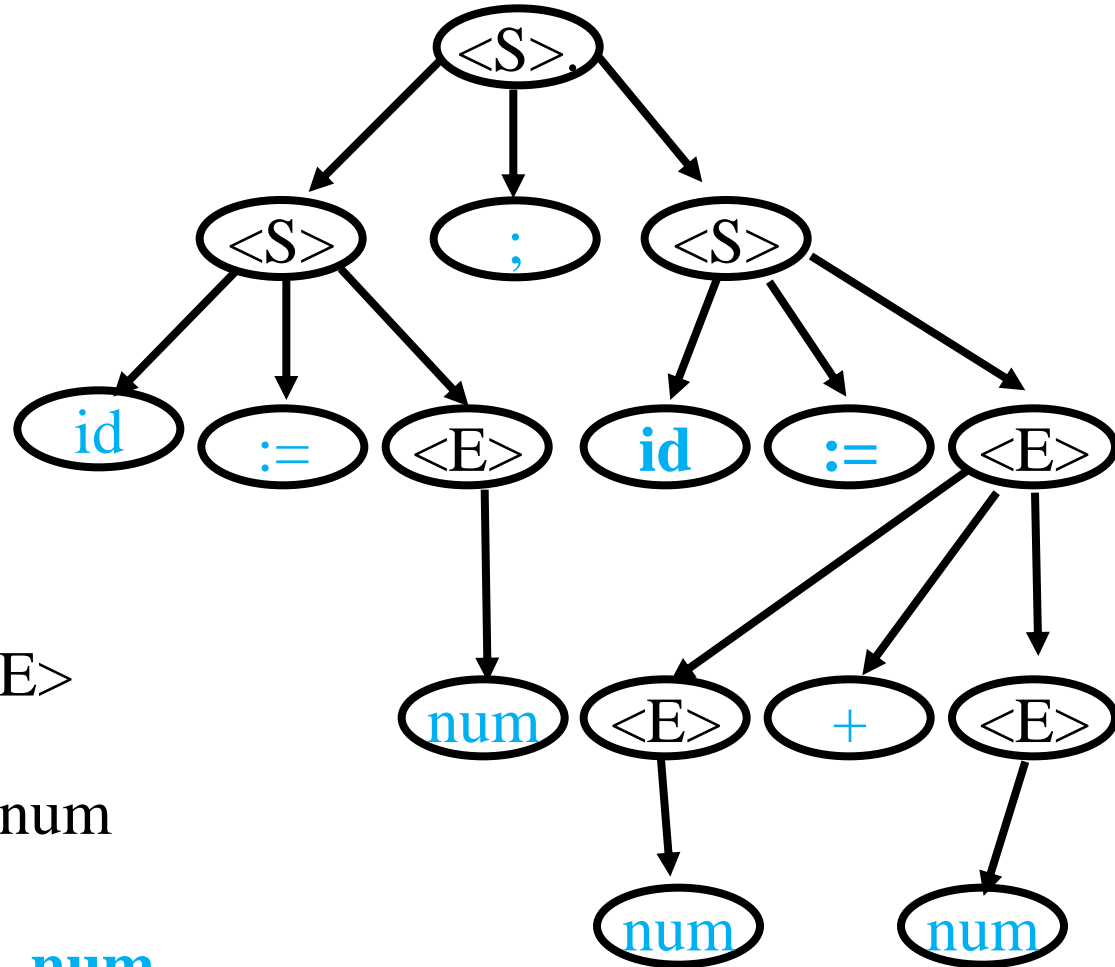
id := <E> ; id := <E>

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  $\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$

2  $\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$

3  $\langle E \rangle \rightarrow \text{id}$

4  $\langle E \rangle \rightarrow (\langle E \rangle)$

# Drawbacks of Ambiguous Grammars

- Ambiguous semantics
- Parsing complexity
- May affect other phases
- But how can we express the syntax of PL using non-ambiguous gramars?

# Non Ambiguous Grammar for Arithmetic Expressions

Ambiguous grammar

$$1 \quad \langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$$

$$2 \quad \langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$$

$$3 \quad \langle E \rangle \rightarrow \mathbf{id}$$

$$4 \quad \langle E \rangle \rightarrow (\langle E \rangle)$$

$$1 \quad \langle E \rangle \rightarrow \langle E \rangle + \langle T \rangle$$

$$2 \quad \langle E \rangle \rightarrow \langle T \rangle$$

$$3 \quad \langle T \rangle \rightarrow \langle T \rangle * \langle F \rangle$$

$$4 \quad \langle T \rangle \rightarrow \langle F \rangle$$

$$5 \quad \langle F \rangle \rightarrow \mathbf{id}$$

$$6 \quad \langle F \rangle \rightarrow (\langle E \rangle)$$

# Non Ambiguous Grammars for Arithmetic Expressions

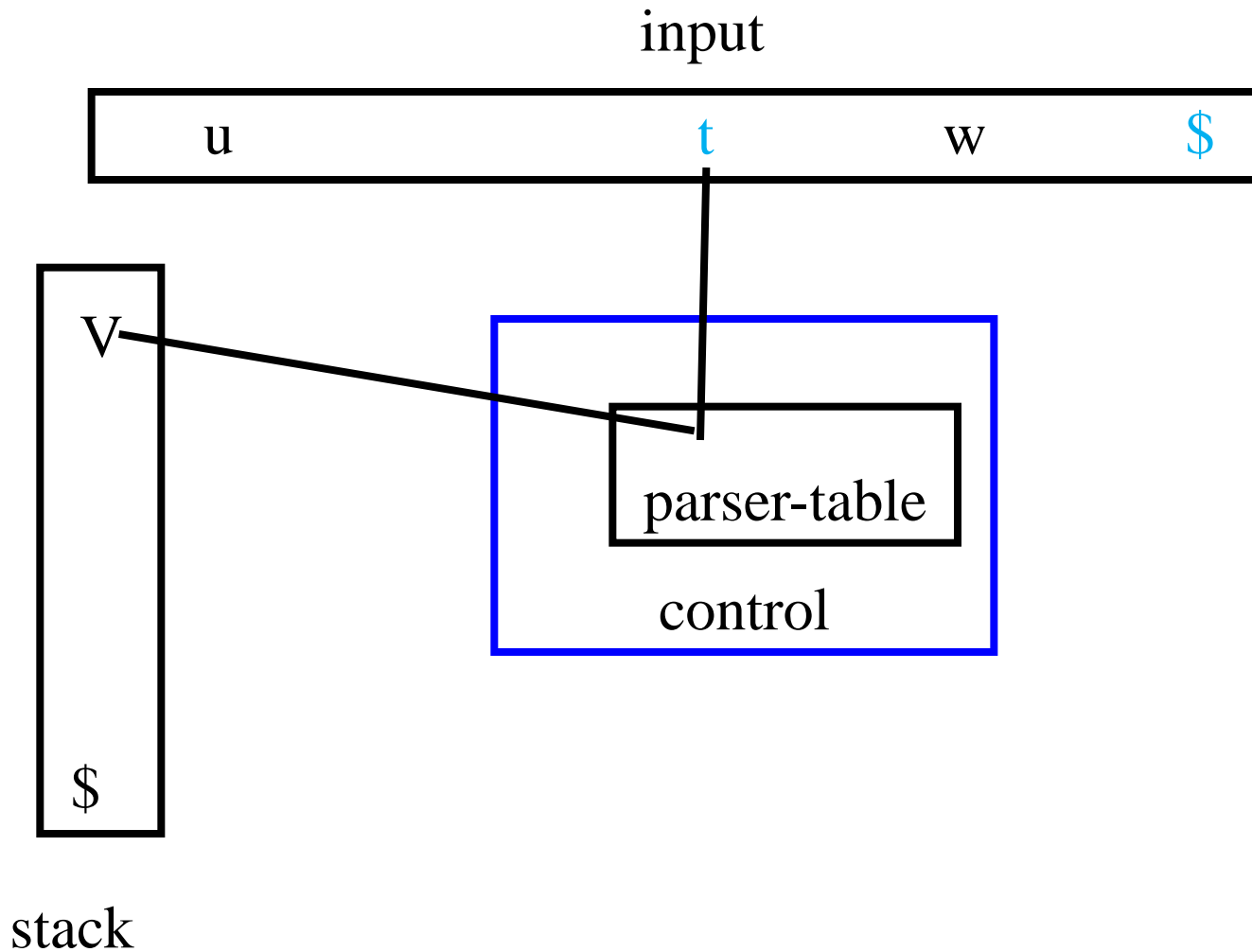
Ambiguous grammar

- |   |   |   |
|---|---|---|
| 1 $\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$ | 1 $\langle E \rangle \rightarrow \langle E \rangle + \langle T \rangle$ | 1 $\langle E \rangle \rightarrow \langle E \rangle * \langle T \rangle$ |
| 2 $\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$ | 2 $\langle E \rangle \rightarrow \langle T \rangle$                     | 2 $\langle E \rangle \rightarrow \langle T \rangle$                     |
| 3 $\langle E \rangle \rightarrow \mathbf{id}$                           | 3 $T \rightarrow \langle T \rangle * \langle F \rangle$                 | 3 $\langle T \rangle \rightarrow \langle F \rangle + \langle T \rangle$ |
| 4 $\langle E \rangle \rightarrow (\langle E \rangle)$                   | 4 $T \rightarrow \langle F \rangle$                                     | 4 $\langle T \rangle \rightarrow \langle F \rangle$                     |
|   | 5 $F \rightarrow \mathbf{id}$   | 5 $\langle F \rangle \rightarrow \mathbf{id}$                           |
|   | 6 $F \rightarrow (\langle E \rangle)$                                   | 6 $\langle F \rangle \rightarrow (\langle E \rangle)$                   |

# Parser Generators

- Input: A context free grammar
- Output: A parser for this grammar
  - Reports syntax error
  - Generates syntax tree
- Example tools
  - yacc, bison, CUP

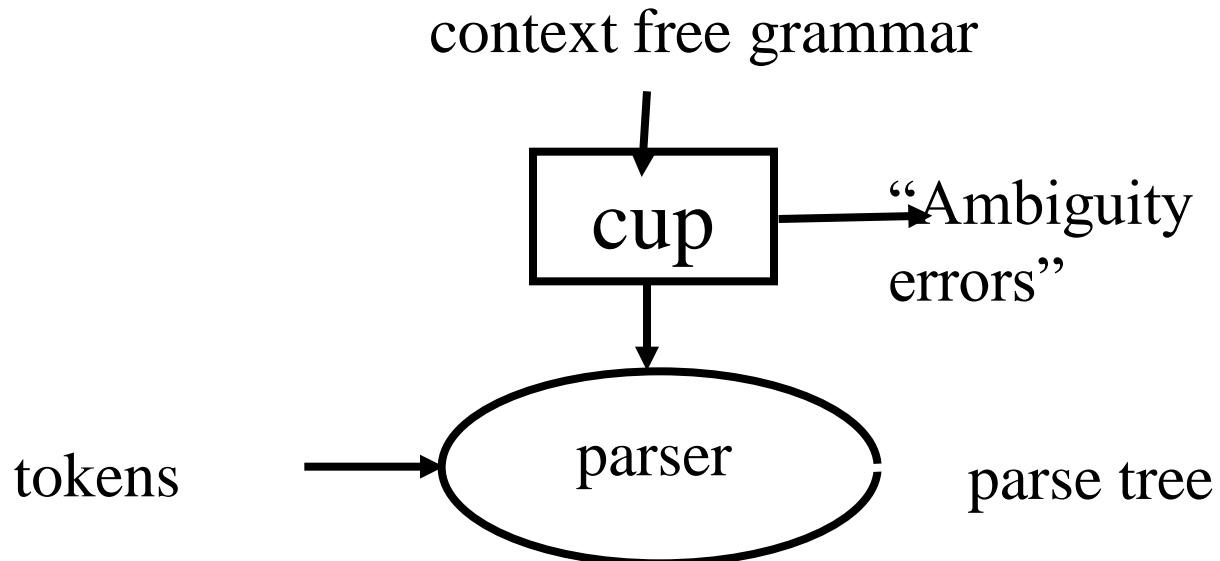
# Pushdown Automaton



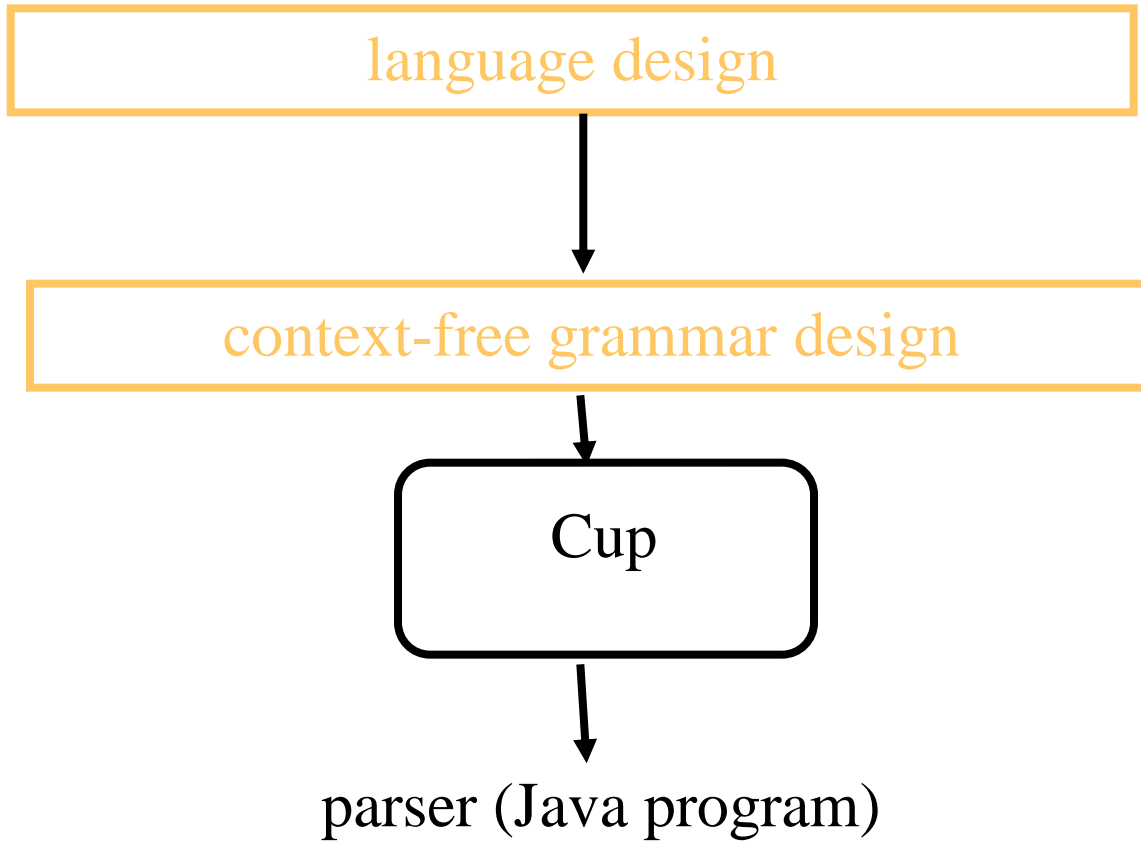


# Efficient Parsers

- 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



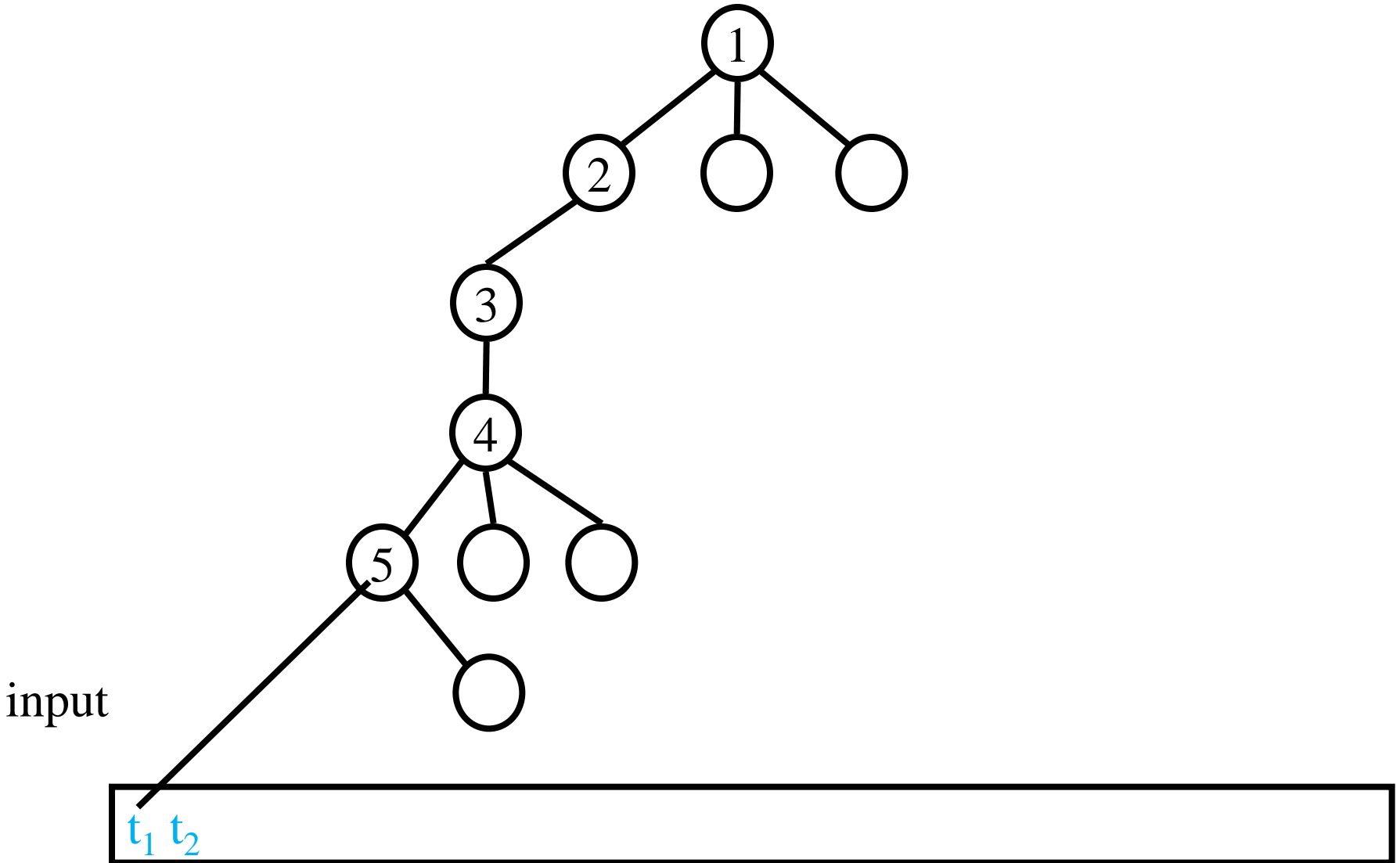
# Designing a parser



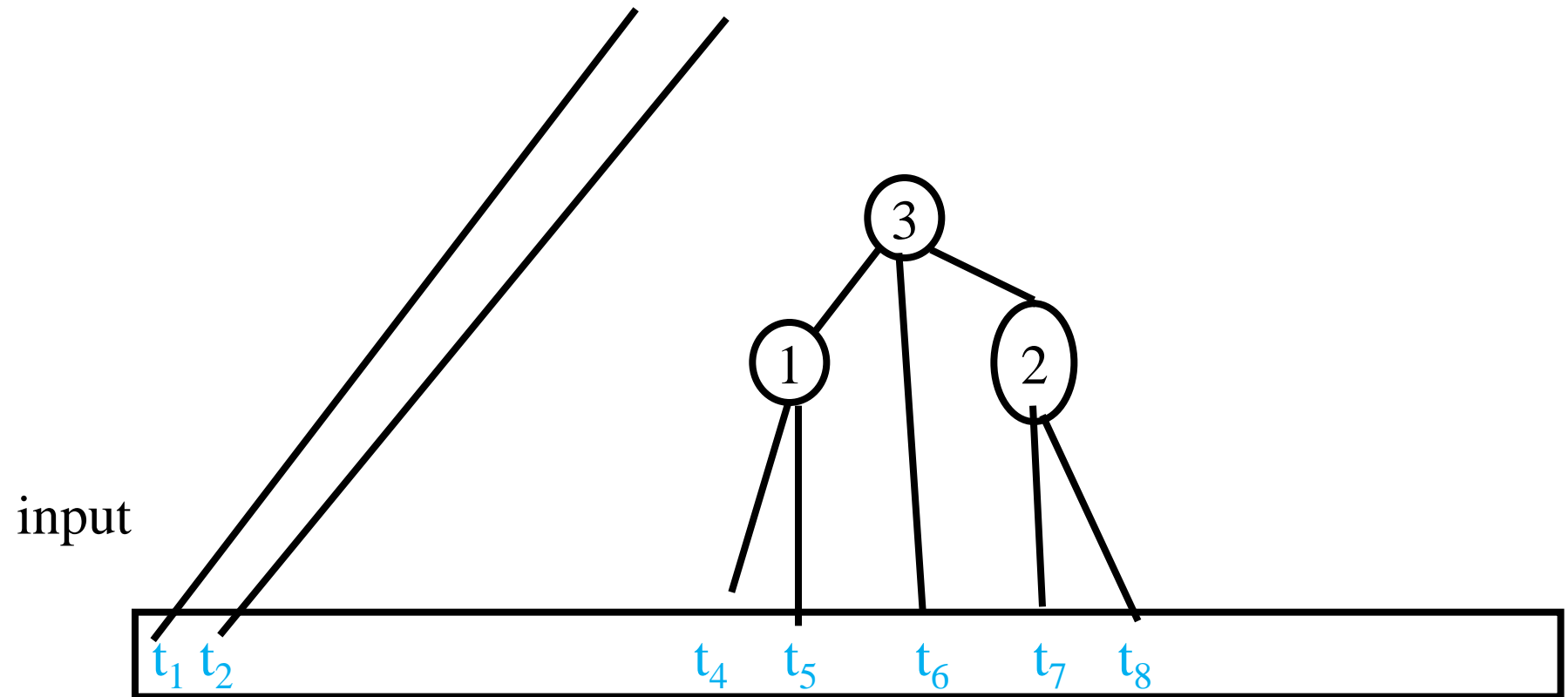
# Kinds of Parsers

- Top-Down (Predictive Parsing) LL
  - Construct parse tree in a top-down manner
  - 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 Parsing

$\langle S \rangle \rightarrow \mathbf{id} := \langle E \rangle$

$\langle S \rangle \rightarrow \langle S \rangle ; \langle S \rangle$

$\langle S \rangle \rightarrow \mathbf{if} (\langle E \rangle) \langle S \rangle \mathbf{else} \langle S \rangle$

$\langle E \rangle \rightarrow \langle T \rangle \langle EP \rangle$

$\langle T \rangle \rightarrow \mathbf{id} \mid (\langle E \rangle)$

$\langle EP \rangle \rightarrow \varepsilon \mid + \langle E \rangle$

# Example Predictive Parser

```
<S> → id := <E>
<S> → if (<E>) <S> else <S>
<E> → <T> <EP>
<T> → id | (<E>)
<EP> → ε | + <E>
```

```
<S> → <S> ; <S> ?
```

```
def parse_S():
    if id(input):
        match(input, id)
        match(input, assign)
        parse_E()
    elif if_tok(input):
        match(input, if_tok)
        match(input, lp)
        parse_E()
        match(input, rp)
        parse_S()
        match(input, else_tok)
        parse_S()
    else:
        syntax_error()
```

```
def parse_E():
    parse_T()
    parse_EP()

def parse_T():
    if id(input):
        match(input, id)
    elif lp(input):
        match(input, lp)
        parse_E()
        match(input, rp)
    else:
        syntax_error()
```

```
def parse_EP():
    if plus(input):
        match(input, plus)
        parse_E()
    elif
        rp(input) or
        else_tok(input) or
        eof(input):
        return // ε
    else:
        syntax_error()
```

# Bad Example for Predictive Parsing

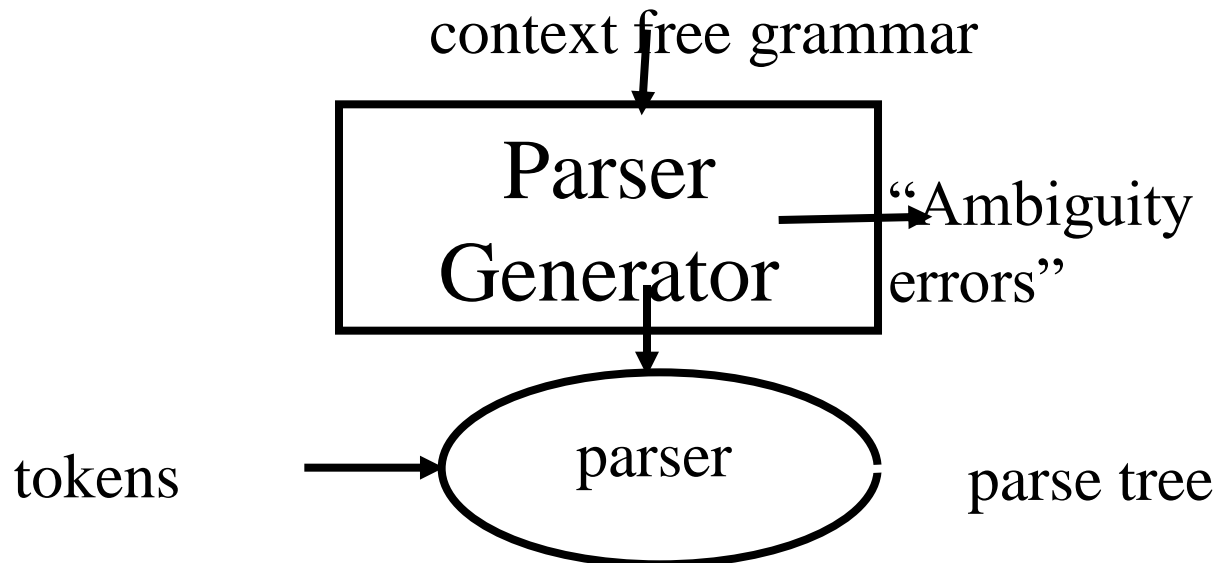
$$\begin{aligned} \langle S \rangle &\rightarrow \langle A \rangle \mathbf{c} \mid \langle B \rangle \mathbf{d} \\ \langle A \rangle &\rightarrow \mathbf{a} \\ \langle B \rangle &\rightarrow \mathbf{a} \end{aligned}$$

```
def parse_S():
    if a(input):
        match(input, a)
        parse_A()
        match(input, c)
    elif a(input):
        match(input, a)
        parse_A()
        match(input, c)
    else report syntax error
```



# Efficient Predictive 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



# Predictive Parser Generation

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

# Bad Example for Predictive Parsing

$\langle S \rangle \rightarrow \langle A \rangle c \mid \langle B \rangle d$

$\langle A \rangle \rightarrow a$

$\langle B \rangle \rightarrow a$

$\alpha$	$\text{First}(\alpha)$
a	{a}
c	{c}
d	{d}
A	{a}
B	{a}
$\langle S \rangle$	{a}
Ac	{a}
Bd	{a}

# Computing First Sets

- For tokens  $t$ , define  $\text{First}(t) = \{t\}$
- For Non-terminals  $\langle A \rangle$ , defines  $\text{First}(\langle A \rangle)$  inductively
  - If  $\langle A \rangle \rightarrow V\alpha$  then  $\text{First}(V) \subseteq \text{First}(A)$
- Can be computed iteratively
- For  $\alpha = V\beta$  define
$$\text{First}(\alpha) = \text{First}(V)$$

# Computing First Iteratively

For each token  $t$ ,  $\text{First}(t) := \{t\}$

For each non-terminal  $\langle A \rangle$ ,  $\text{First}(\langle A \rangle) := \{\}$

while changes occur do

  if there exists a non-terminal  $\langle A \rangle$  and

    a rule  $\langle A \rangle \rightarrow V\alpha$  and

    a token  $t \in \text{First}(V)$  and

$t \notin \text{First}(\langle A \rangle)$

  add  $t$  to  $\text{First}(\langle A \rangle)$

# A Simple Example

$\langle E \rangle \rightarrow ( \langle E \rangle ) \mid \text{ID}$

First(ID)	First( $\langle E \rangle$ )
{ID}	{}
	{ID}
	{ID, (}

# Constructing Predictive 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  $t \in \text{First}(\alpha)$  apply the rule  
 $\langle A \rangle \rightarrow \alpha$ 
  - Otherwise report an error

# Handling Nullable Non-Terminals

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



# Predictive Parser

```
<S> → id := <E>  
<S> → if (<E>) <S> else <S>  
<E> → id <EP> | (<E>)  
<EP> → ε | + <E> <EP>
```

```
def parse_S():  
    if id(input):  
        match(input, id)  
        match(input, assign)  
        parse_E()  
    elif if_tok(input):  
        match(input, if_tok)  
        match(input, lp)  
        parse_E()  
        match(input, rp)  
        parse_S()  
        match(input, else_tok)  
        parse_S()  
    else:  
        syntax_error()
```

```
def parse_E():  
    if id(input):  
        match(input, id)  
        parse_EP()  
    elif lp(input):  
        match(input, lp)  
        parse_E()  
        match(input, rp)  
    else:  
        syntax_error()
```

```
def parse_EP():  
    if plus(input):  
        match(input, plus)  
        parse_E()  
        parse_EP()  
    elif  
        rp(input) or  
        else_tok(input) or  
        eof(input):  
        return // ε  
    else:  
        syntax_error()
```

# Handling Nullable Non-Terminals

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

# Computing First For Nullable Non-Terminals

For each token  $t$ ,  $\text{First}(t) := \{t\}$

For each non-terminal  $\langle A \rangle$ ,  $\text{First}(\langle A \rangle) = \{\}$

while changes occur do

  if there exists a non-terminal  $\langle A \rangle$  and

    a rule  $\langle A \rangle \rightarrow V_1 V_2 \dots V_n \alpha$  and

$V_1, V_2, \dots, V_{n-1}$  are nullable

    a token  $t \in \text{First}(V_n)$  and

$t \notin \text{First}(\langle A \rangle)$

  add  $t$  to  $\text{First}(\langle A \rangle)$

# An Imperative View

- Create a table with  
#Non-Terminals  $\times$  #Tokens  
Entries
- If  $t \in \text{select}(\langle A \rangle \rightarrow \alpha)$  apply the rule  
“apply the rule  $\langle A \rangle \rightarrow \alpha$ ”
- Empty entries correspond to syntax errors

# A Simple Example

$\langle E \rangle \rightarrow ( \langle E \rangle ) \mid \text{ID}$

	(	)	ID	\$
$\langle E \rangle$	$\langle E \rangle \rightarrow ( \langle E \rangle )$		$\langle E \rangle \rightarrow \text{id}$	

# Left Factoring

- If a grammar contains two productions
  - $\langle A \rangle \rightarrow a \alpha \mid a \beta$   
then it is not LL(1)
- Left factor  $A \rightarrow a A'$ 
  - $A' \rightarrow \alpha \mid \beta$
- Can be done for a general grammar
- The resulting grammar may or may not be LL(1)

# Left Recursion

- Left recursive grammar is never LL(1)
  - $\langle A \rangle \rightarrow \langle A \rangle a \mid b$
- Convert into a right-recursive grammar
- Can be done for a general grammar
- The resulting grammar may or may not be LL(1)

# History & Theoretical Properties

- Rich history in linguistics (Panini, Chomsky)
- Used in Algol'60 (Naur)
- Decidable problems
  - Emptiness
  - Finiteness
  - Derivation
  - LL(1) grammar
- Undecidable problems
  - Inclusion
  - Equivalence
  - Ambiguity
  - “LL(1) language”



# Abstract Syntax

- An ambiguous grammar used to concisely define the hierarchy of sentences
  - Define the structure of a tree
- $\text{Exp} ::= \text{Exp BinOp Exp} \mid \text{ID} \mid \text{Const}$

# Context Sensitive Features

- Some PL features cannot be defined using context free grammars
  - Identifiers
  - Types
- Some are awkward to define

# 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
- [Bottom-up parsing is more expressible
  - Good tools, Bison, yacc, CUPP]
- Useful beyond PL