# Formal Syntax of Programming Languages

Mooly Sagiv

http://www.cs.tau.ac.il/~msagiv/courses/pl15.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 {letter}({letter}|{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 | /* ignored */ |
| preprocessor directive | #include <foo.h> |
| | #define NUMS 5, 6 |
| macro | NUMS |
| whitespace | \t \n \b |

# 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 '$E_1$ op $E_2$' is an expression

<E> → id | <E> <op> <E>
<op> → + | - | * | /

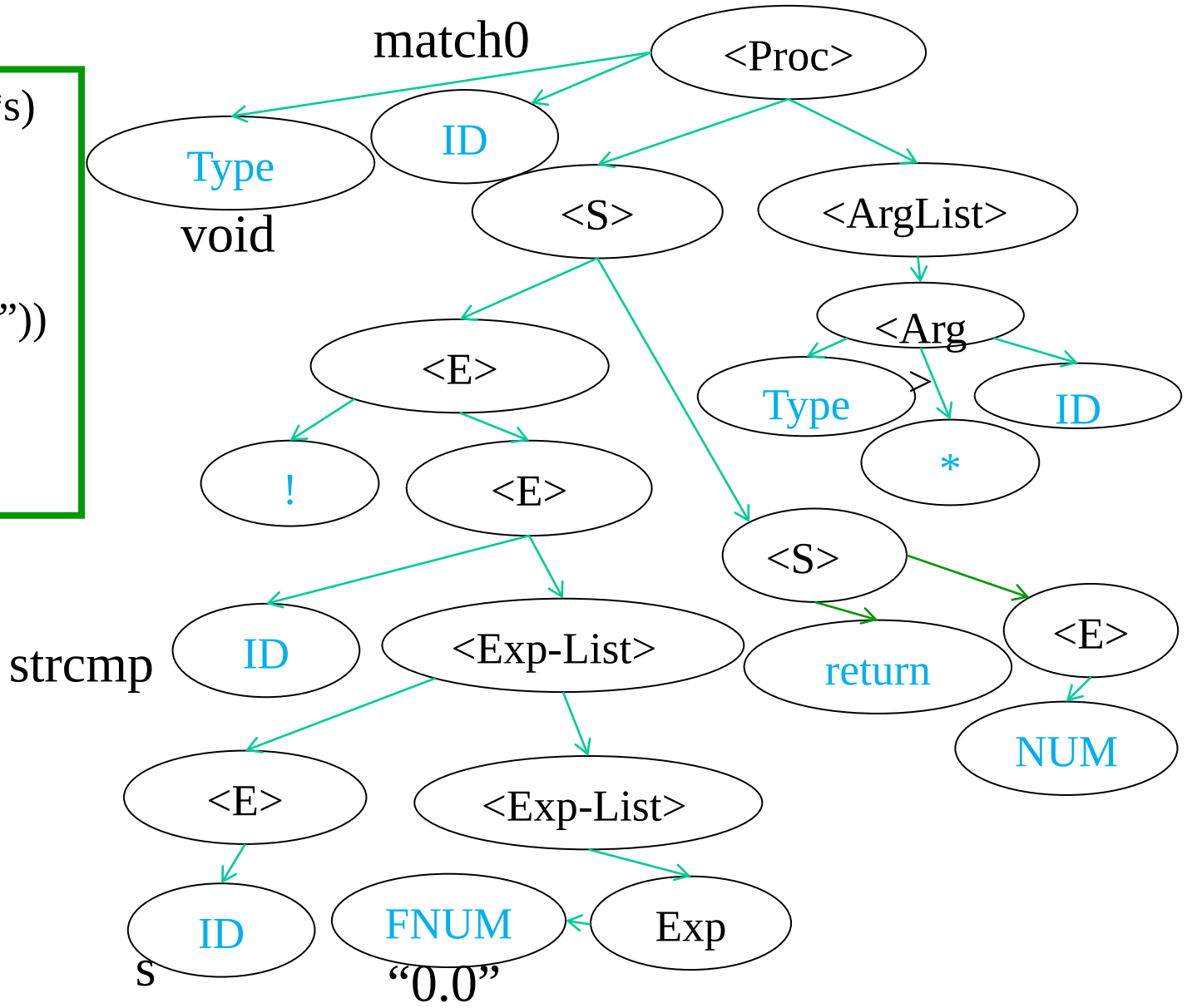
# Statement Definitions

- If id is a identifier and E is an expression then 'id := E' is a statement
- If $S_1$ and $S_2$ are statements and E is an expression then
  - '$S_1$ ; $S_2$' is a statement
  - 'if (E) $S_1$ else $S_2$' is a statement

<S> → id := <E>
<S> → <S> **;** <S>
<S> → **if (**<E>**)** <S> **else** <S>

# 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
  <Non-Terminal> $\rightarrow$ 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

# 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

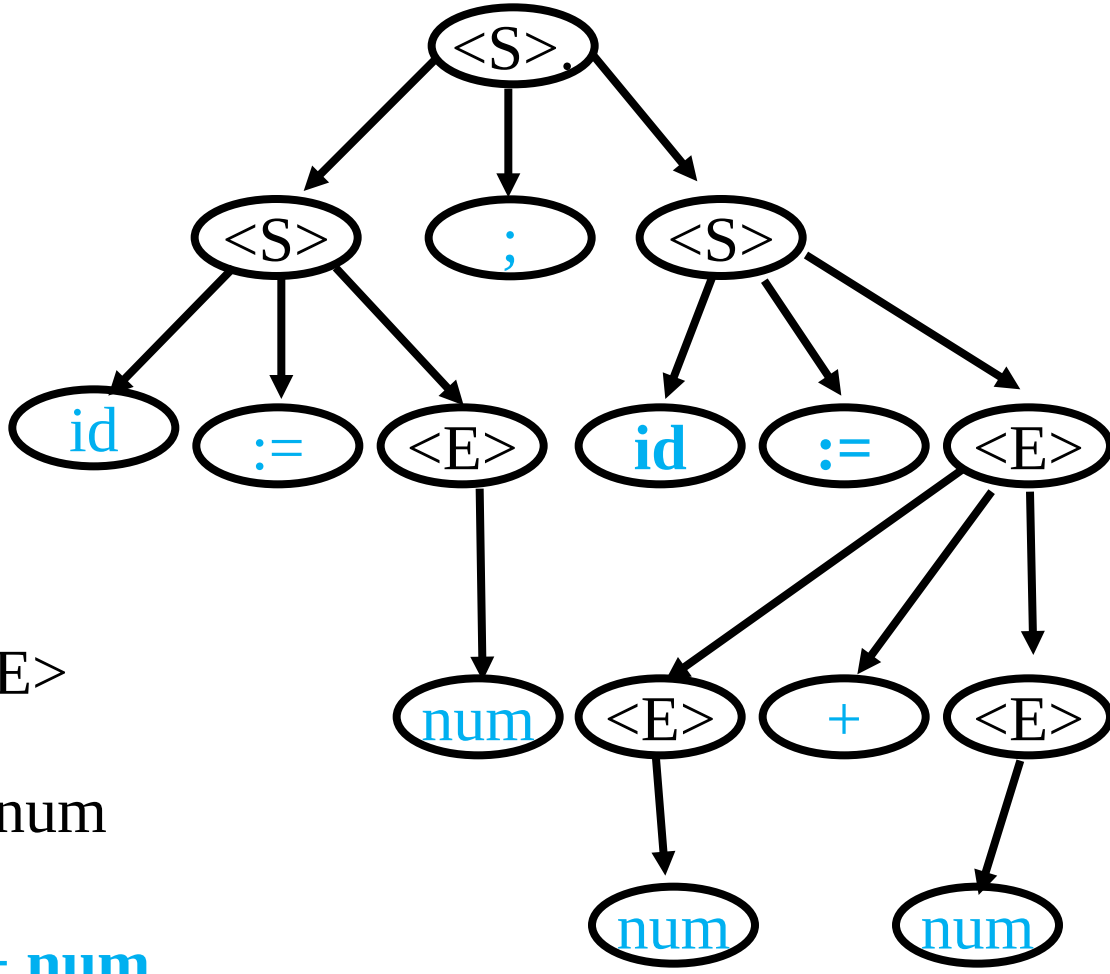
**id :=** <E> **; id :=** <E>

**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> → <E> + <E>
2 <E> → <E> * <E>
3 <E> → id
4 <E> → (<E>)

# 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 <E> → <E> **+** <E>
2 <E> → <E> ***** <E>
3 <E> → **id**
4 <E> → **(**<E>**)**

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

# Non Ambiguous Grammars for Arithmetic Expressions

Ambiguous grammar

1 <E> → <E> + <E>
2 <E> → <E> * <E>
3 <E> → id
4 <E> → (<E>)

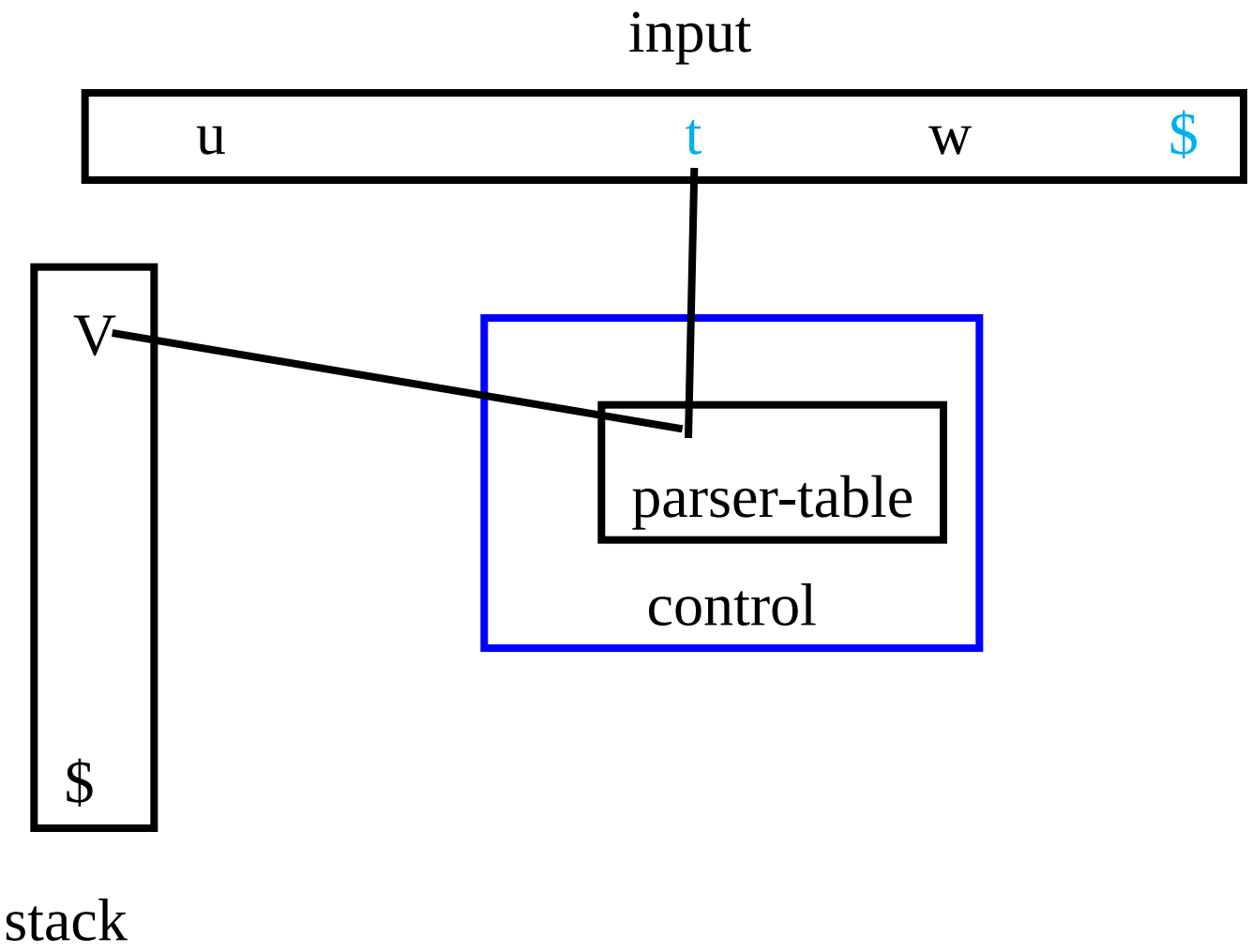
1 <E> → <E> + <T>
2 <E> → <T>
3 T → <T> * <F>
4 T → <F>
5 F → id
6 F → (<E>)

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

# Parser Generators

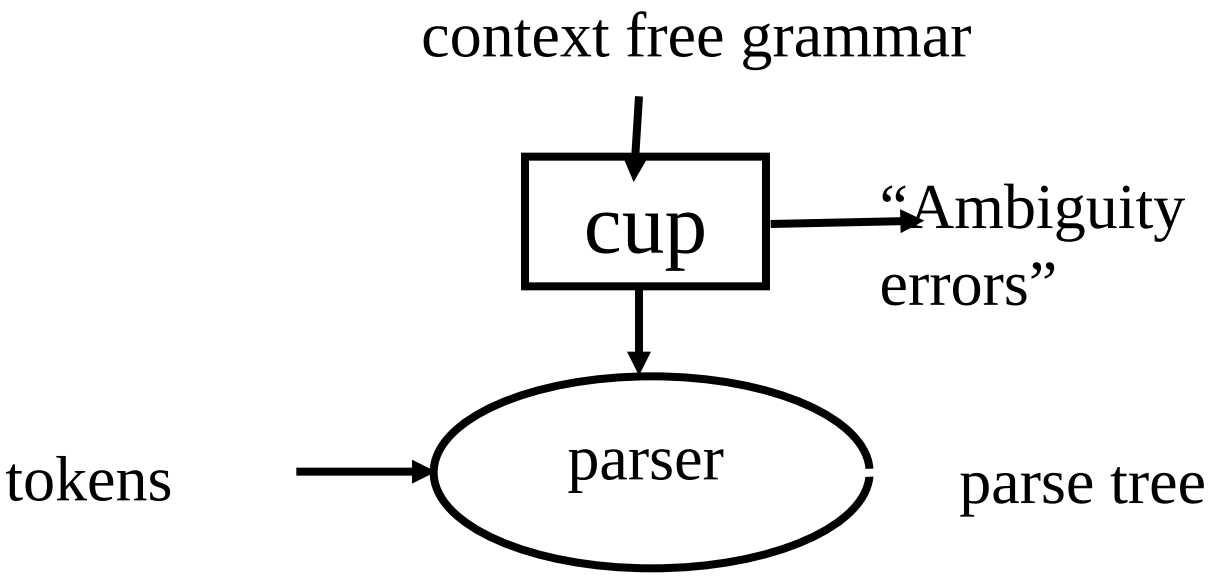
- Input: A context free grammar
- Output: A parser for this grammar
  - Reports syntax error
  - Generates syntax tree
- Example tools
  - yacc, bison, CUP

# Pushdown Automaton

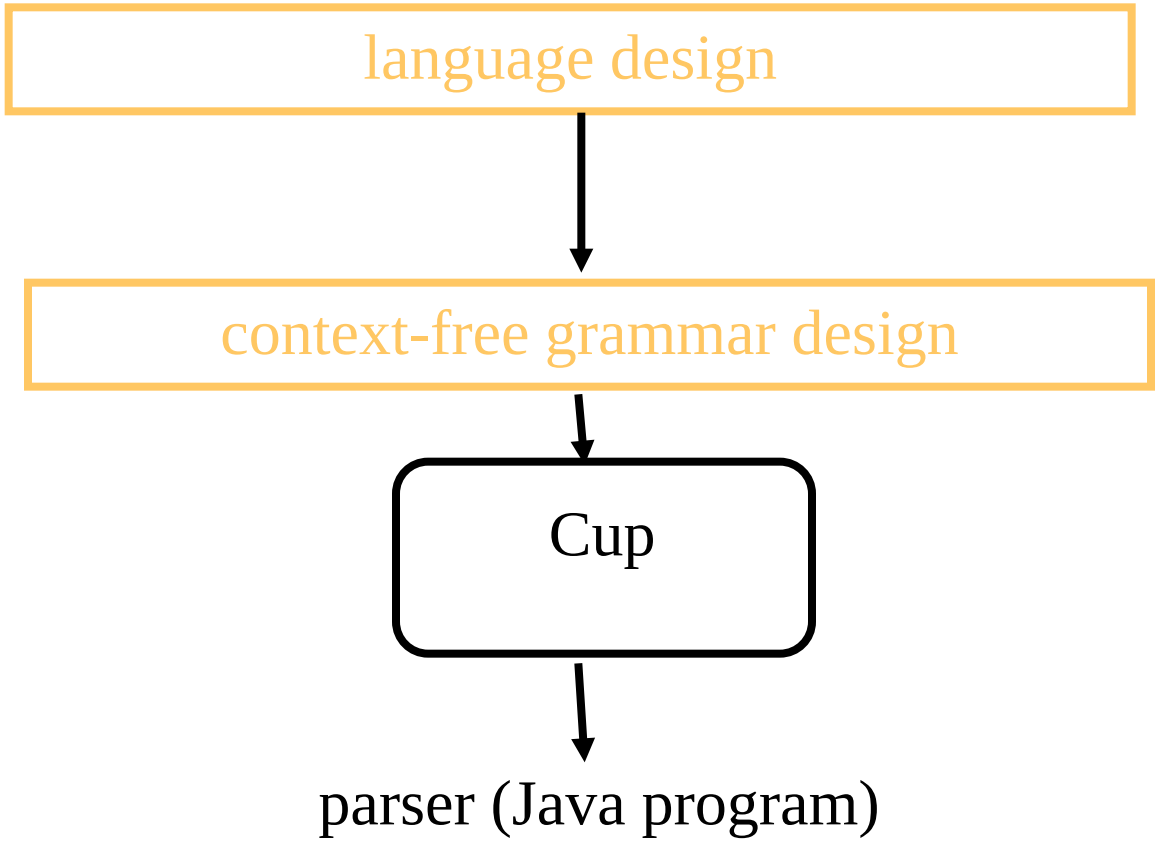


# Efficient Parsers

- 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



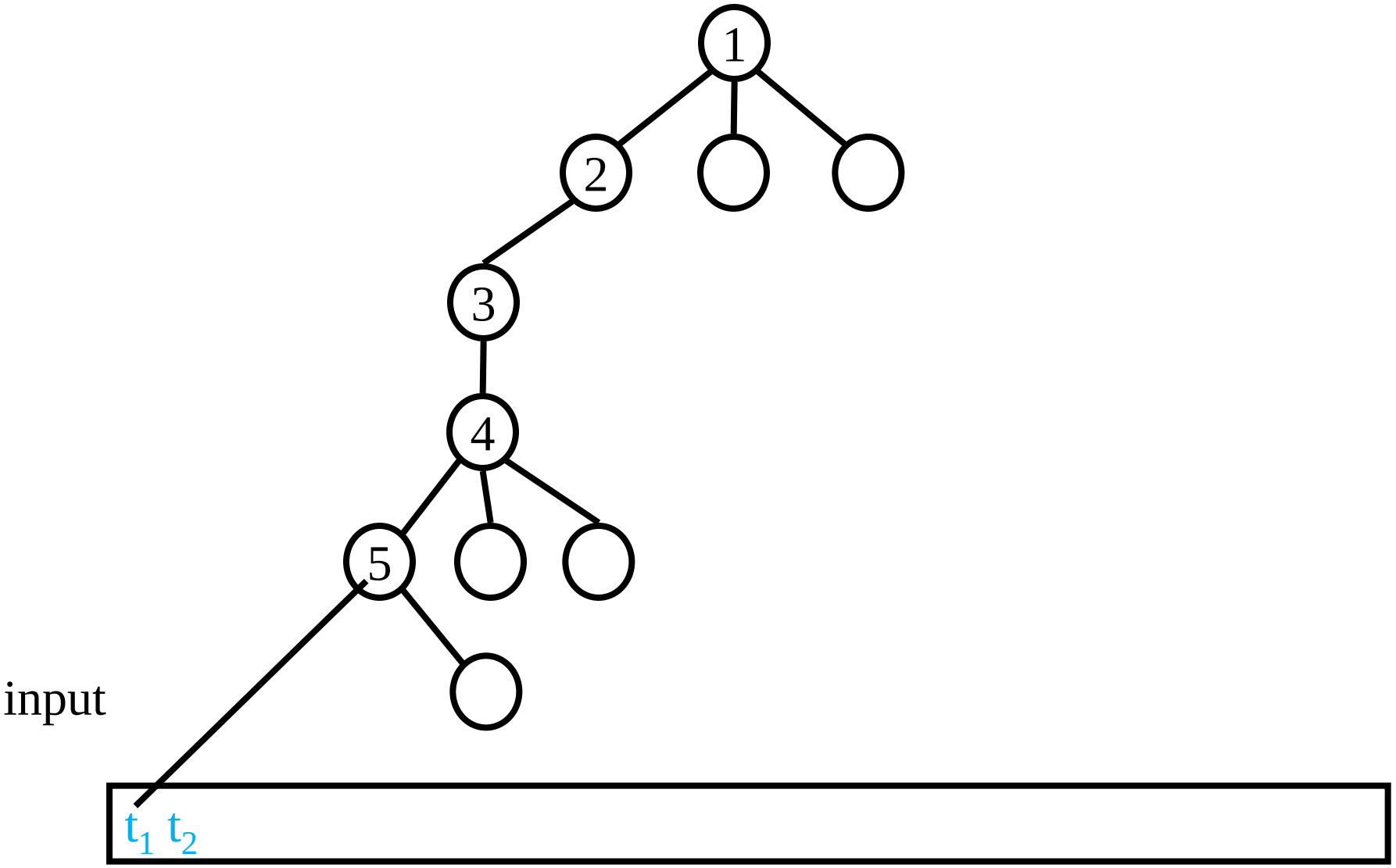
# Designing a parser



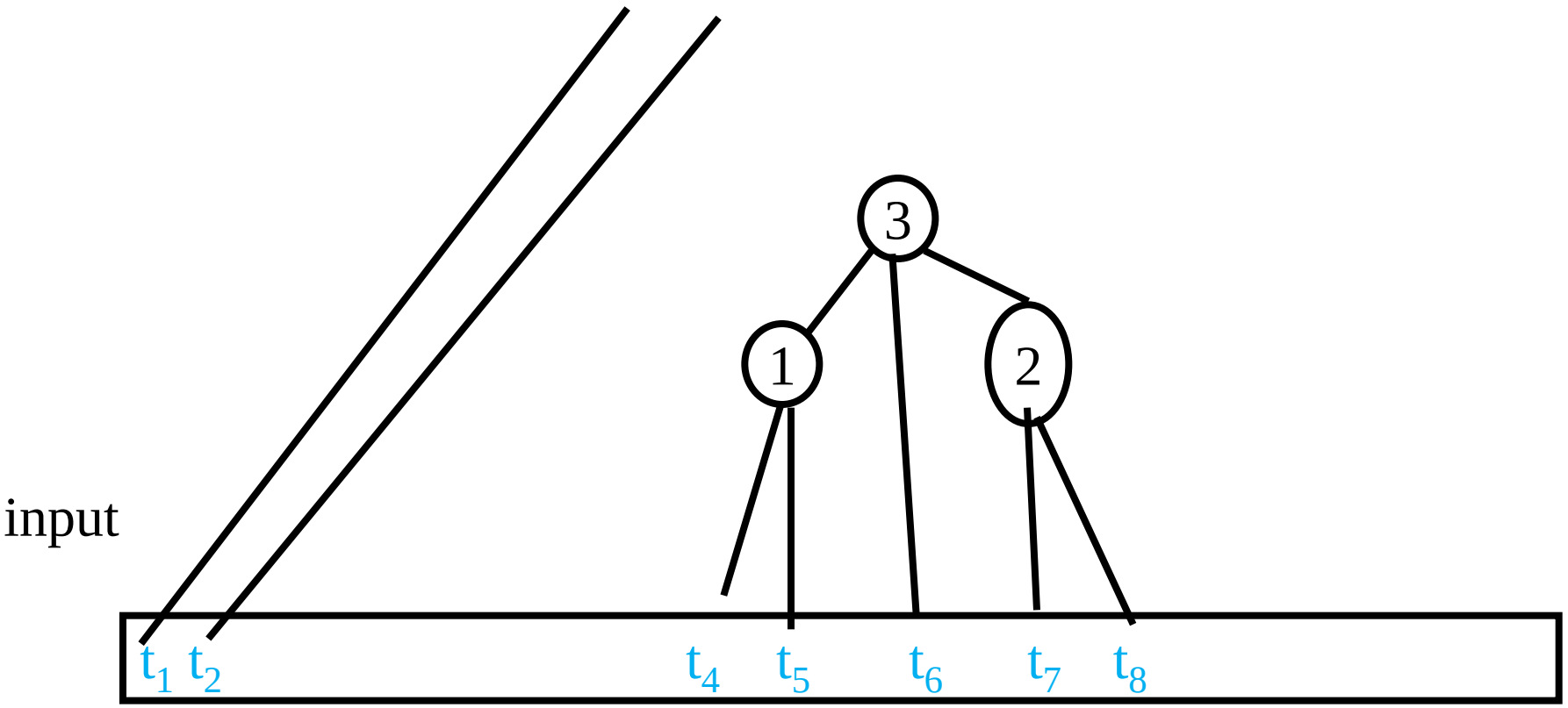
# Kinds of Parsers

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

<S>→ **id** := <E>

~~<S> → <S> **;** <S>~~

<S> → **if (**<E>**)** <S> **else** <S>

<E> → <T> <EP>

<T> → **id** | **(**<E>**)**

<EP> → ε | **+** <E>

# 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()
```

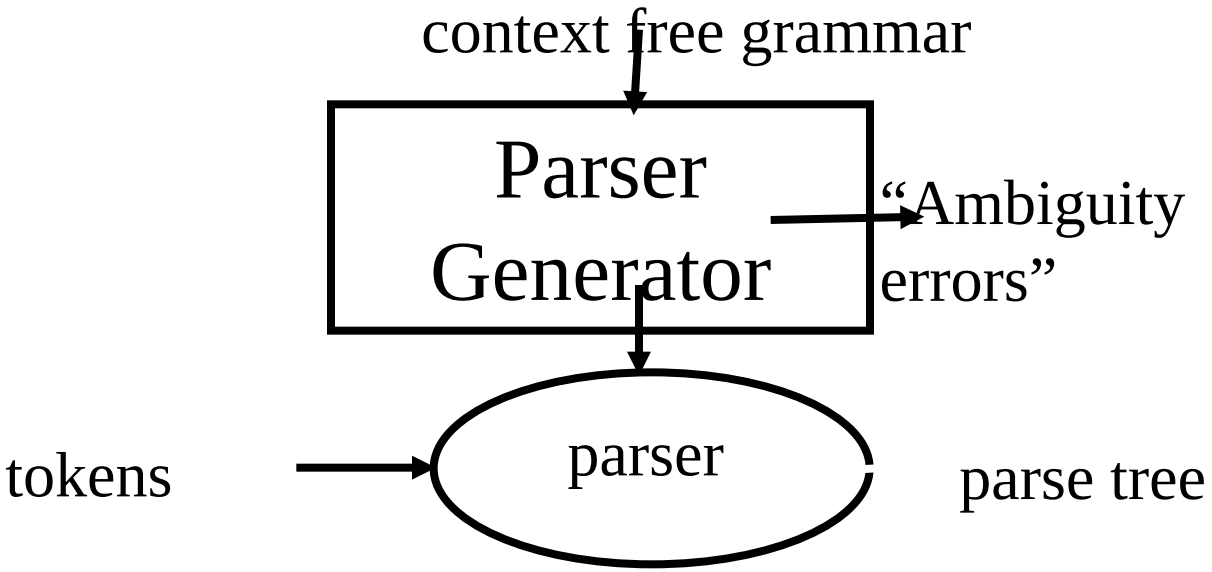
# Bad Example for Predictive Parsing

<S> → <A> **c** | <B> **d**
<A> → **a**
<B> → **a**

```
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 $\alpha$
  - $First(\alpha) = \{ t \mid \exists \beta: \alpha \rightarrow^* t\ \beta \}$
- The grammar is LL(1) if for every two grammar rules $\langle A \rangle \rightarrow \alpha$ and $\langle A \rangle \rightarrow \beta$
  - $First(\alpha) \cap First(\beta) = \varnothing$

# Bad Example for Predictive 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> → Vα then First(V) ⊆ 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 ∈ First(V) and
                t ∉ First(<A>)
     add t to First(<A>)

# A Simple Example

<E> → ( <E> ) | ID

| First(ID) | First(<E>) |
|---|---|
| {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 $\in$ 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 | $\exists \beta,\gamma$: <S> $\rightarrow^*$ $\beta$ <A> 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 <A> define
  - Follow(<A>) = { t | ∃β: <S> $\rightarrow^*$ <A>tβ }
- For a rule <A> → α
  - If α is nullable then
    select(<A>→ α) = First(α) ∪ Follow(<A>)
    - Otherwise select(<A>→ α) = First(α)
- The grammar is LL(1) if for every two grammar rules <A> → α and <A> → β
  - Select(A → α) ∩ Select(A → β) = ∅

# 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> → $V_1$ $V_2$ … $V_n$ α and
        $V_1$, $V_2$, …, $V_{n-1}$ are nullable
        a token t ∈ First($V_n$) and
        t ∉ First(<A>)
    add t to First(<A>)

# An Imperative View

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

# A Simple Example

<E> → ( <E> ) | ID

| | ( | ) | ID | $ |
|---|---|---|---|---|
| <E> | <E> → (<E>) | | <E>→id | |

# Left Factoring

- If a grammar contains two productions
  - <A> → a $\alpha$ | a $\beta$
    then it it is not LL(1)
- Left factor A → 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)
  - <A> → <A>a | 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

- Reach 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
  Exp ::= Exp BinOp Exp | ID | 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]
- Used beyond PL