Syntax Analysis

Mooly Sagiv

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
- Simple Top-Down Parsing
- Bottom-Up Parsing (next lesson)

Benefits of formal definitions

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

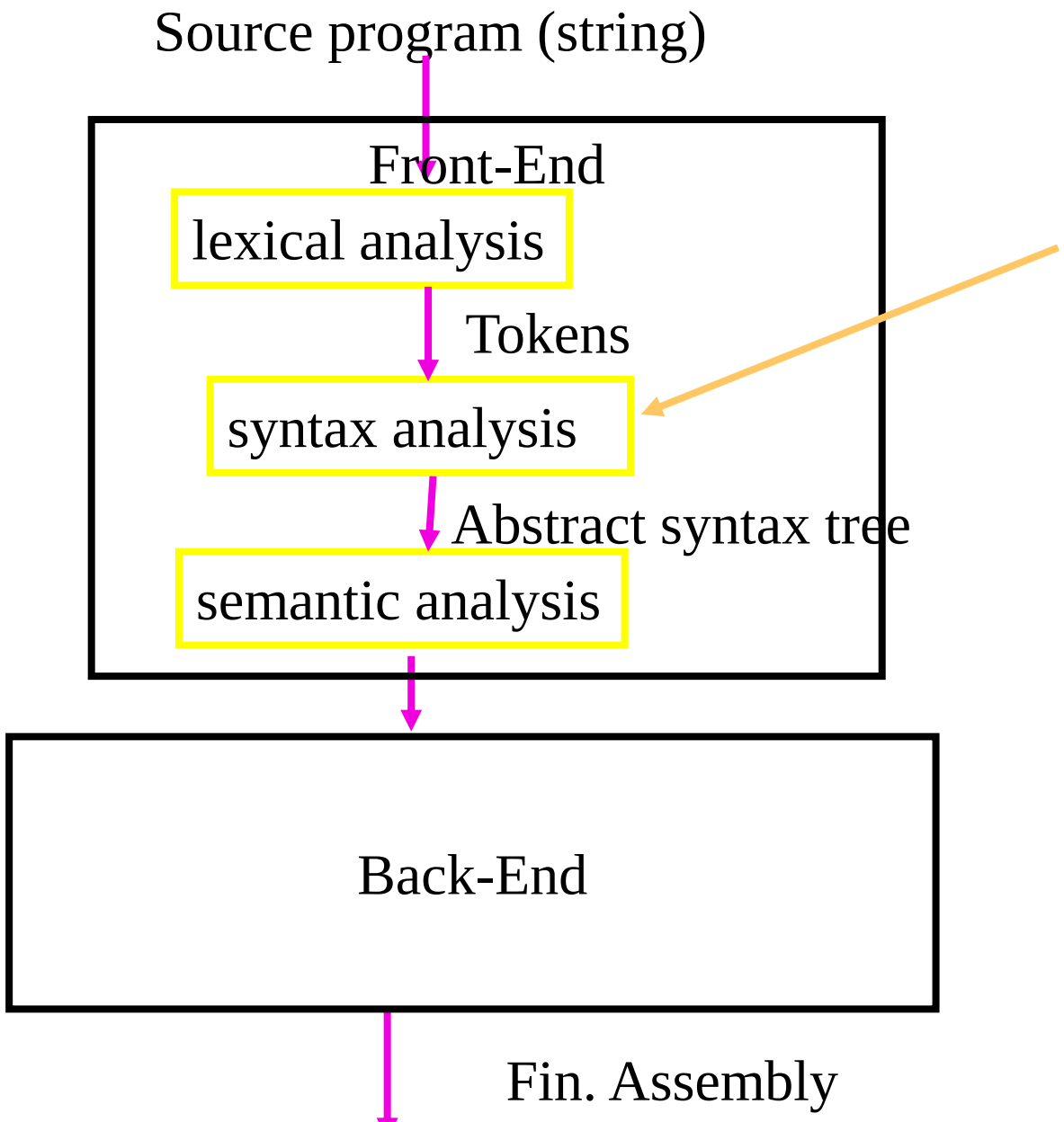
What is a good formal definition?

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

Benefits of formal syntax for programming language

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

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

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> → + | - | * | /

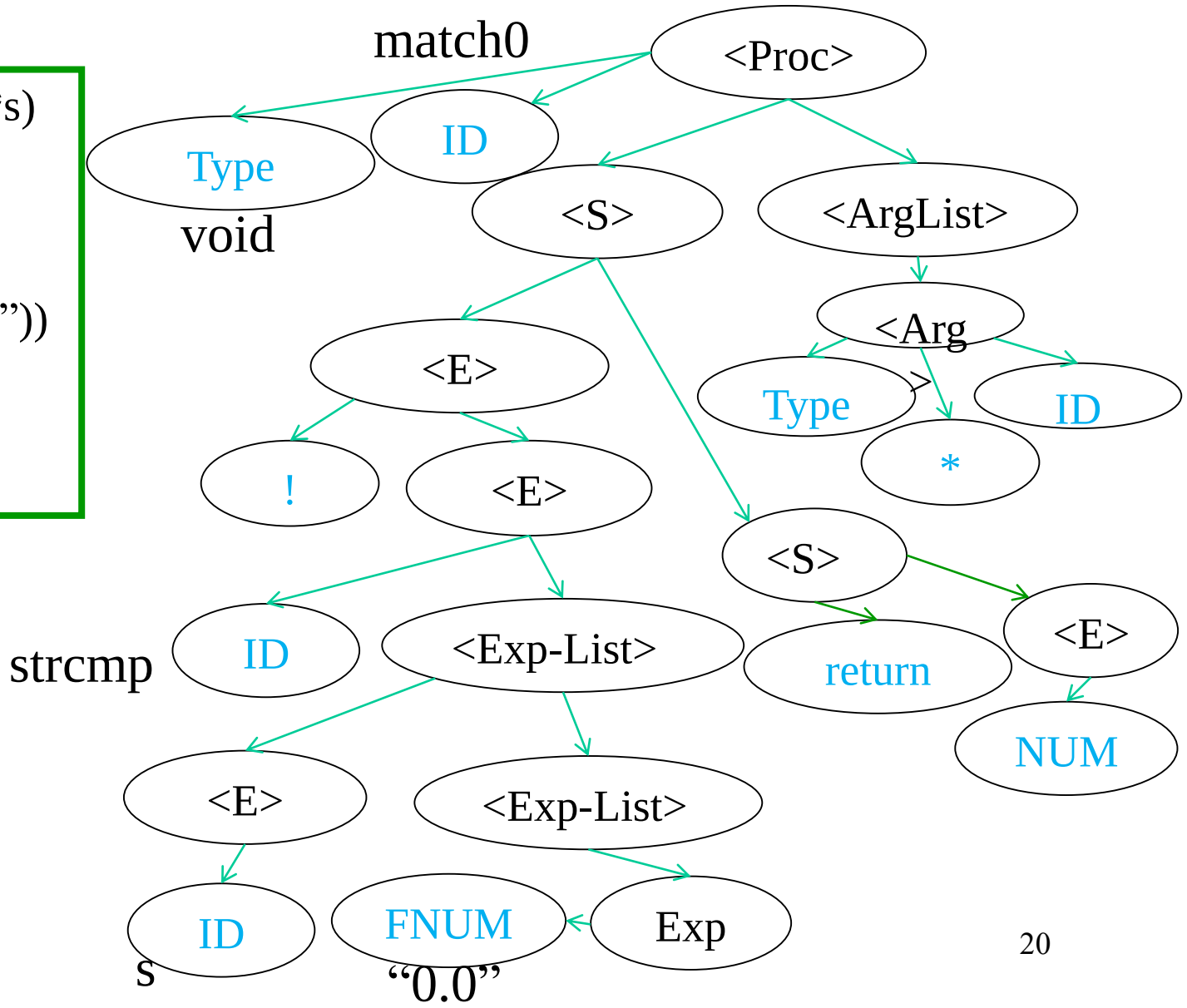
Statement Definitions

- 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

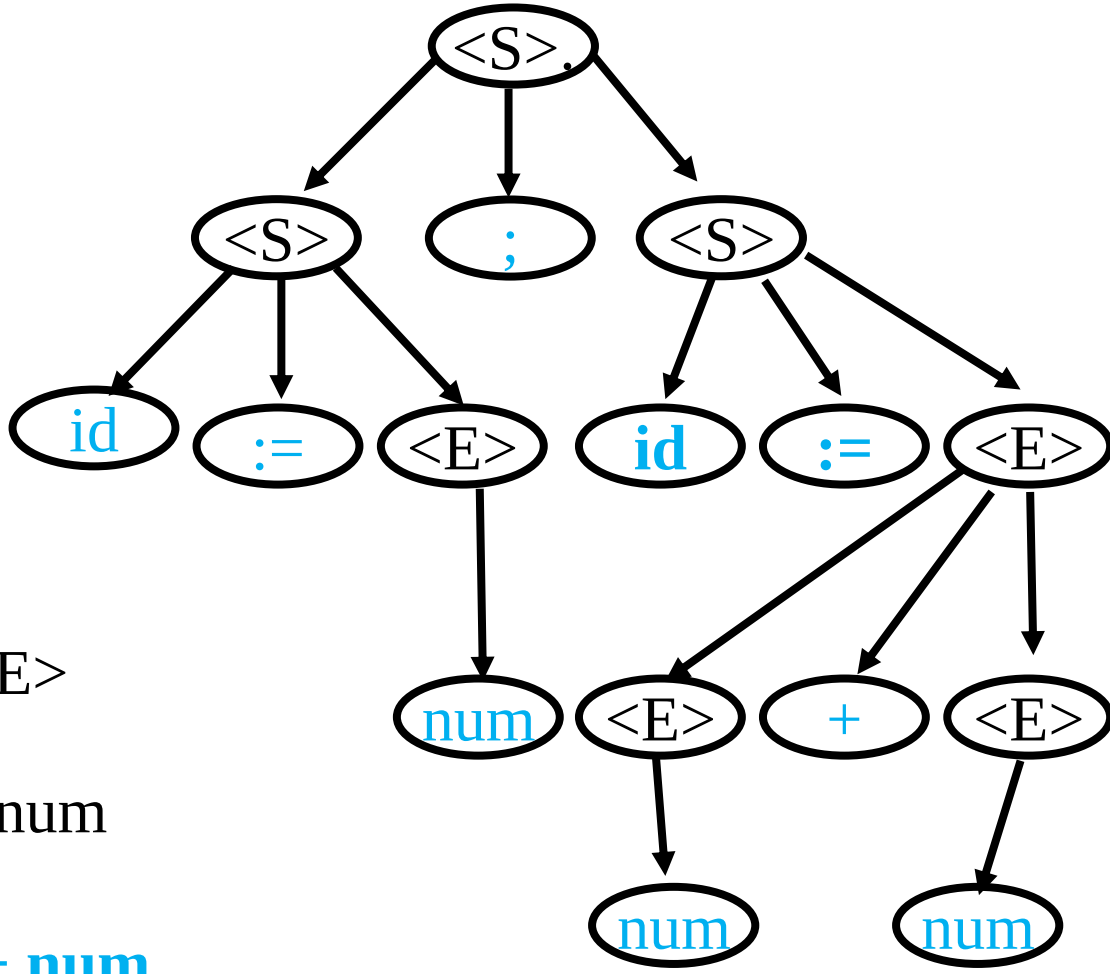
id := <E> **; id :=** <E>

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>)

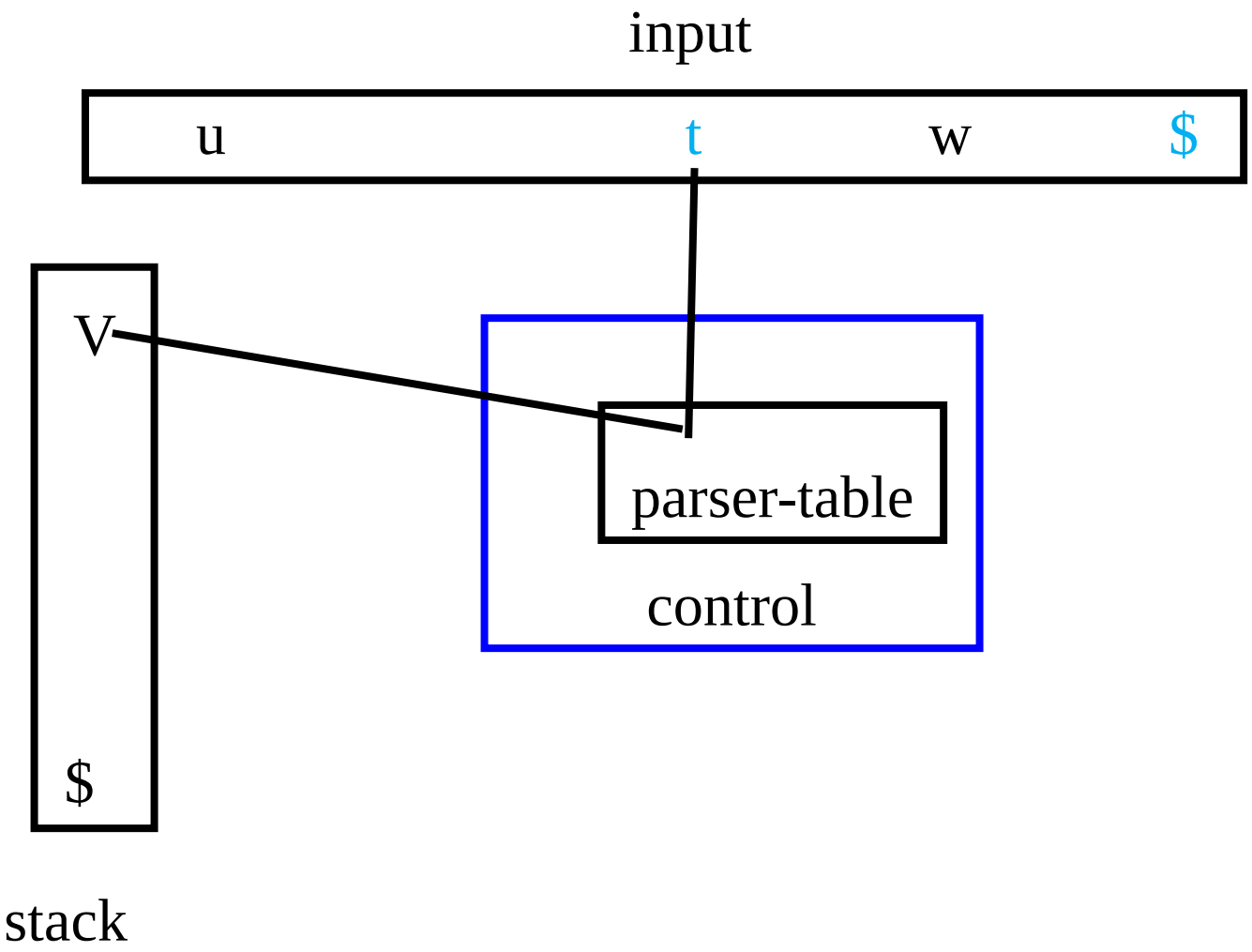
The Parsing Problem

- Given a context free grammar G
- An input program P as a sequence of tokens
- Does there exists a tree in G which drives W
  - Yes – Produce an output tree
  - No – Produce a prefix P’ of P which cannot be extended into a valid word

Parser Generators

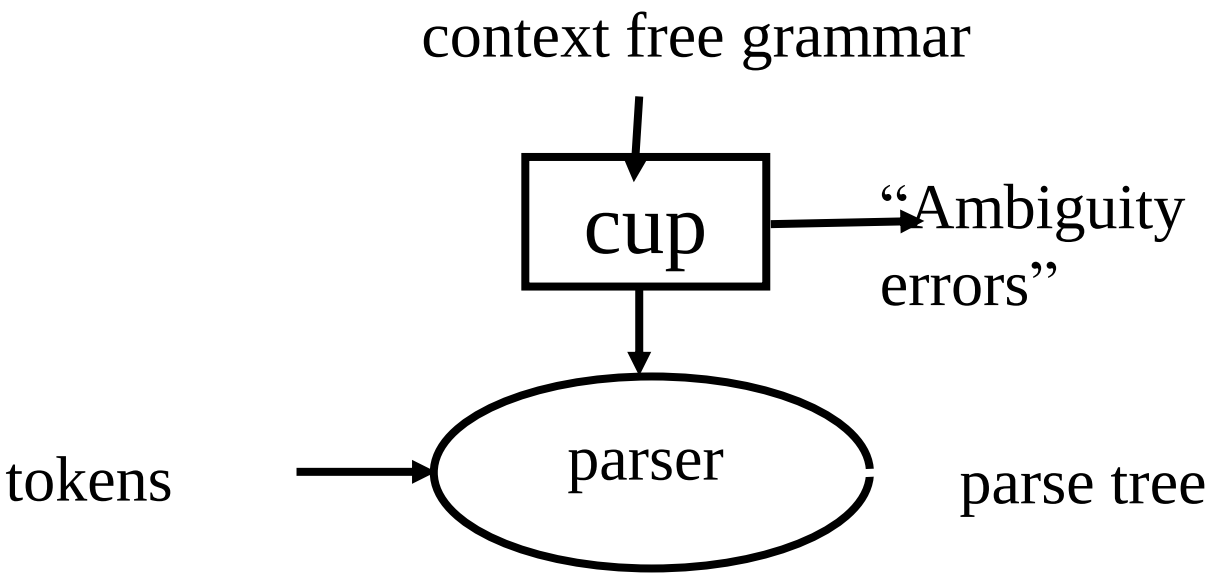
- Input: A context free grammar
- Output: A parser for this grammar
  - Reports syntax error
  - Generates syntax tree
- General algorithms
  - Earley $O(n^3)$
- Tools for special grammars
  - yacc, bison, CUP, ANTLR

Pushdown Automaton

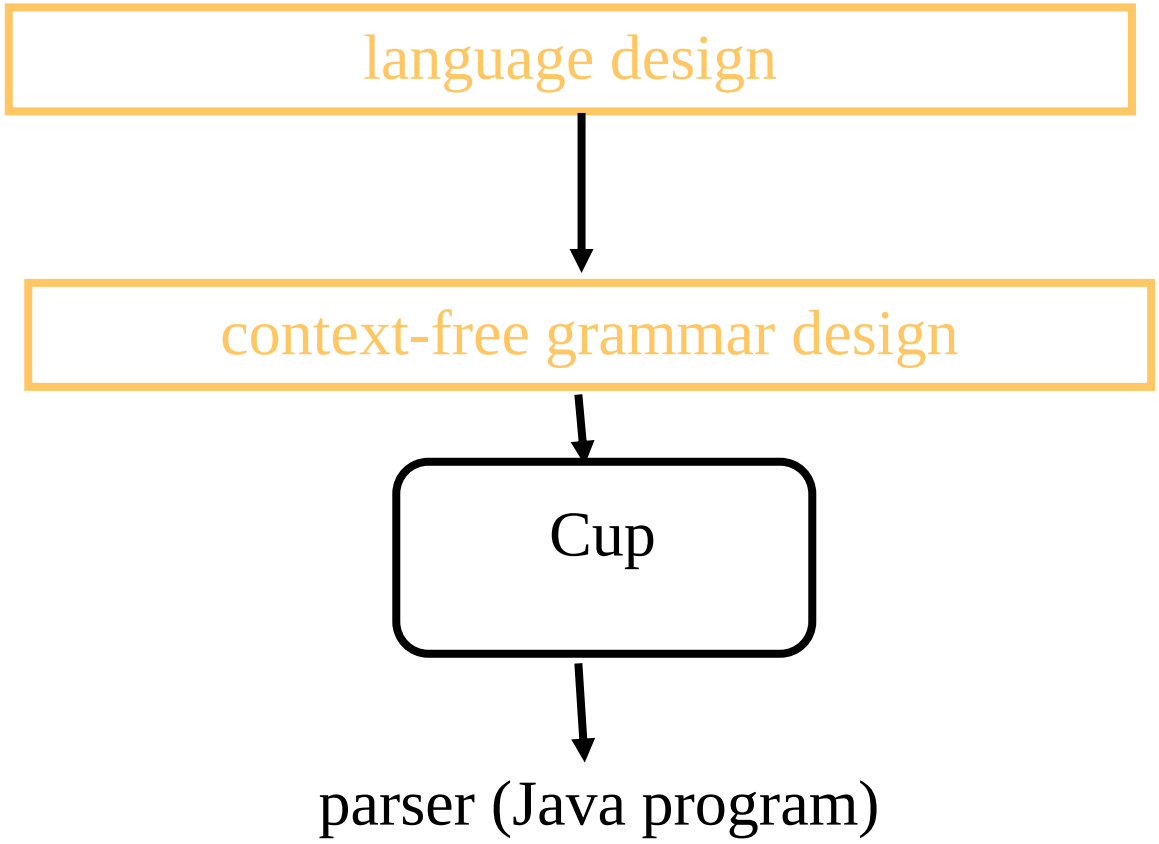


Efficient Parsers

- 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



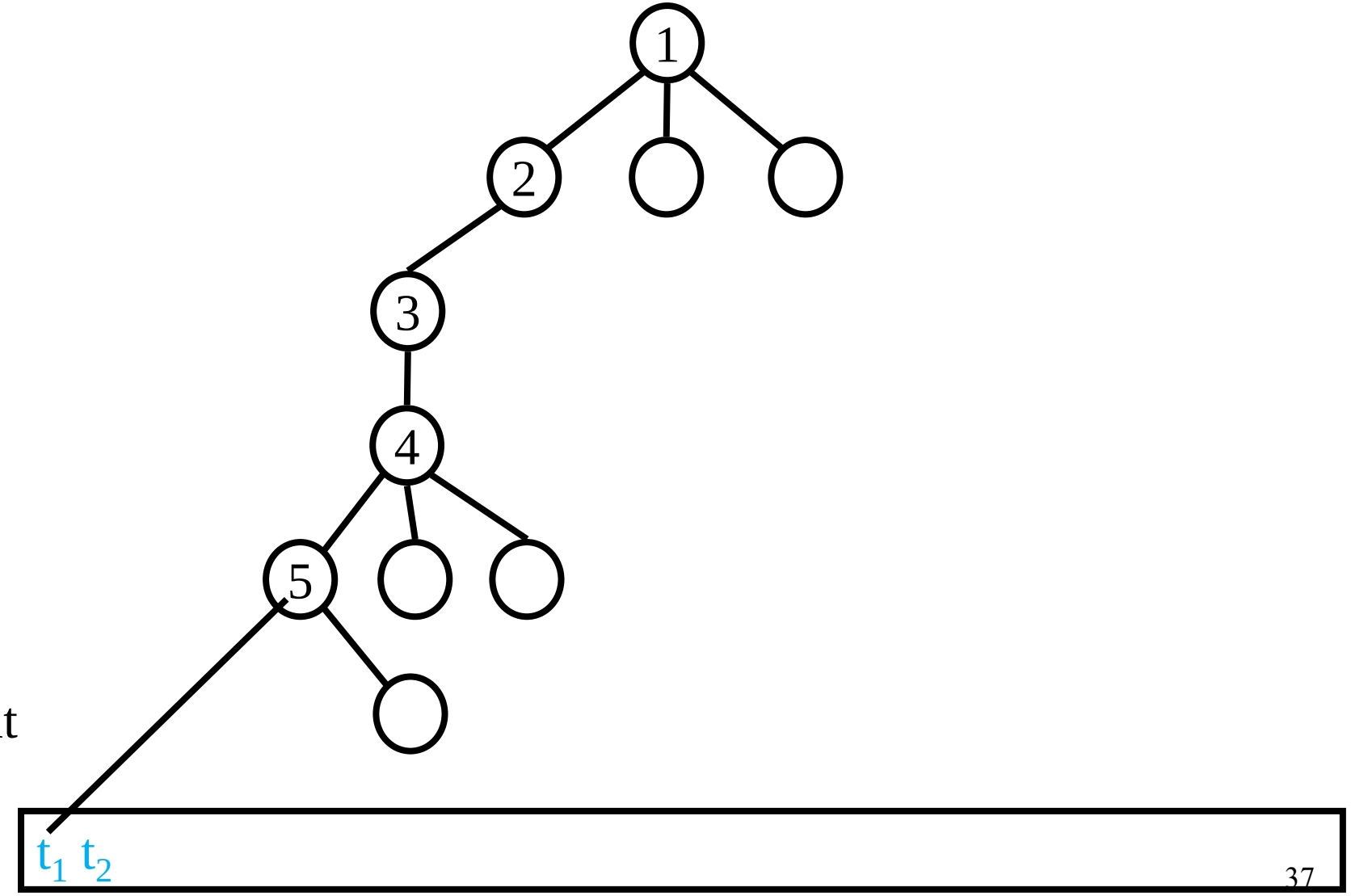
Designing a parser



Kinds of Parsers

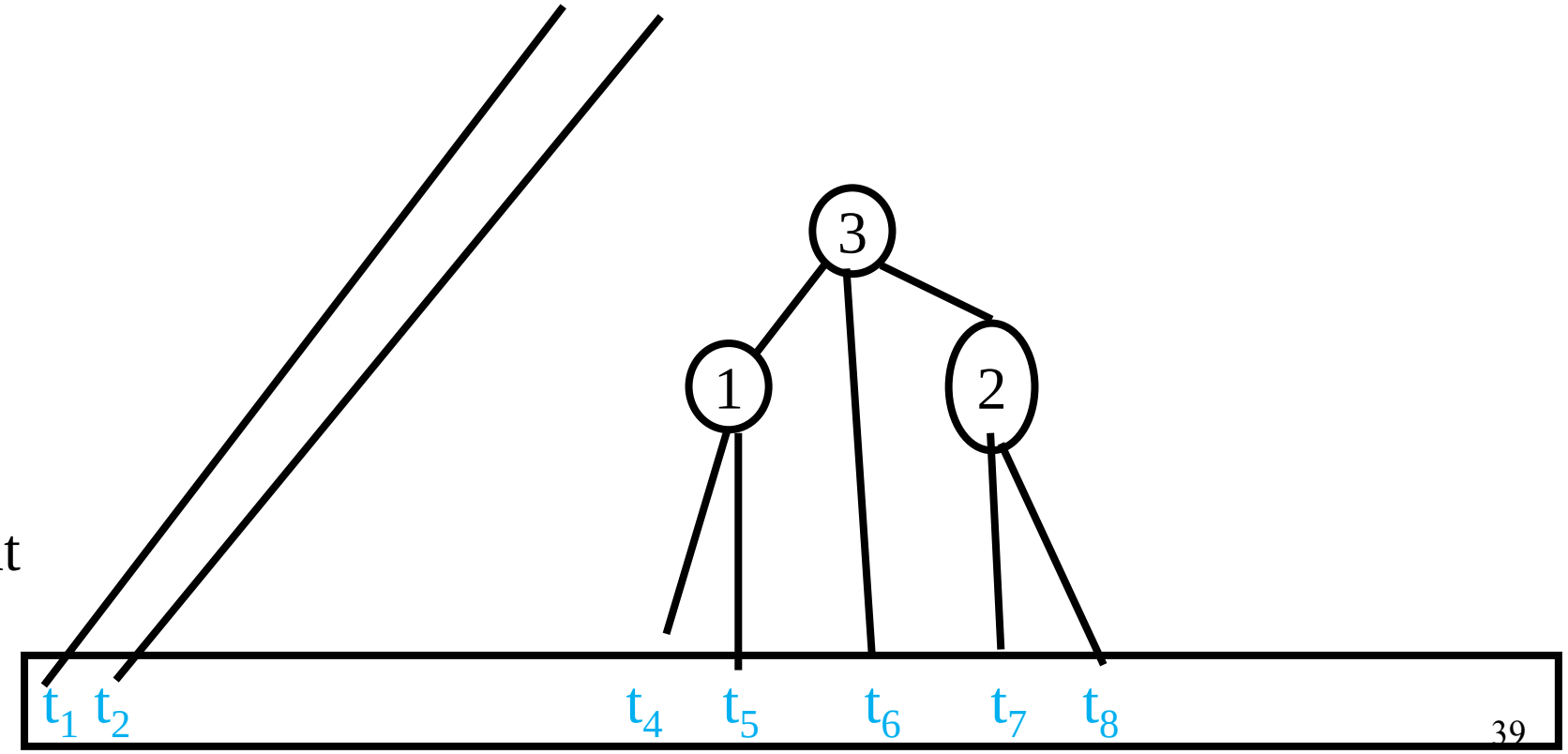
- Top-Down (Predictive Parsing) LL
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
  - For every non-terminal and token **predict** the next production
  - Preorder tree traversal
- Bottom-Up LR
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation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

Top-Down Parsing



```
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;
```

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()
```

```
def main:
  parse_S;
   if not match(input,
EOF) …
```

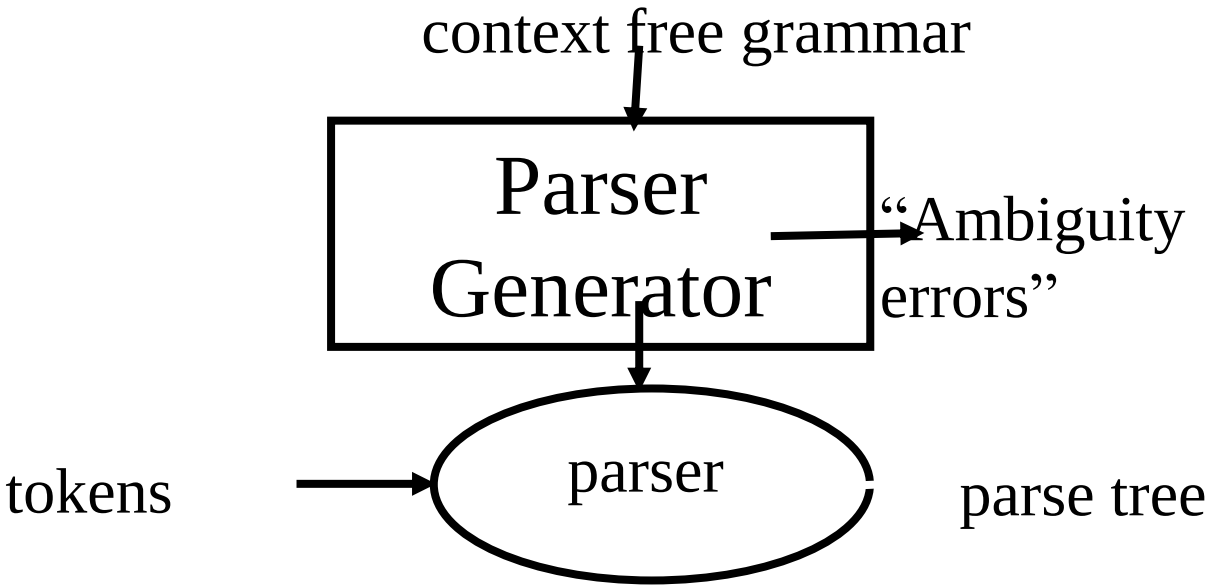
Bad Example for Predictive Parsing

<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



Automatically Constructing Predictive Parser

- Input: A context free grammar
- Output:
  - An indication that the grammar is not good
    - Ambiguous for predictive parsing
  - Parsing tables
    - Python/Java/C code

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 α
  - $First(\alpha) = \{ t \mid \exists \beta: \alpha \rightarrow^* t\ \beta \}$
- 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

L(S)={ac, ad}

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

Good Example for Predictive Parsing

<S> → <A> <CD>
<A> → a
<CD> → c
<CD> → d

L(S)={ac, ad}

α	First(α)
a	{a}
c	{c}
d	{d}
A	{a}
CD	{c,d}
S	{a}
ACD	{a}

Ambiguous Grammar

<E>→ <E> + <E> | num

String	First
num	
<E> + <E>	

Ambiguous Grammar

<E>→ <E> + <E> | num

String	First
num	{num}
<E> + <E>	{num}

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, (}

Ambiguous Grammar

<E>→ <E> + <E> | num

String	First
num	{num}
<E> + <E>	{num}

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(α) apply the rule <A> $\rightarrow$ α
  - 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> $\rightarrow^*$ β <A> t γ}
- 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

- The First set can change
  - <S>→ <A> | b
  - <A> → a | <X> b
  - <X> → ε
- First may not be enough

Handling Nullable Non-Terminals

- The First set can change
- First may not be enough
  - <B> → <S> A e
  - <S> → a <S> b | ε
  - A → d

Handling Nullable Non-Terminals

- For a non-terminal <A> define
  - Follow(<A>) = { t | ∃β: <S> →* <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>)

Computing Follow Iteratively

For each non-terminal A, Follow(A) := {}
Follow(S) = {$}
while changes occur do
    for each rule V → α A β
        for each token t ∈ First(β) and
            add t to Follow(A)
        if β is nullable
            for each token t ∈ Follow(V)
                add t to Follow(A)

Iteratively Computing Follow

1 $E \rightarrow T\ E'$
2 $E' \rightarrow \varepsilon$
3 $E' \rightarrow +\ T\ E'$
4 $T \rightarrow F\ T'$
5 $T' \rightarrow \varepsilon$
6 $T' \rightarrow *\ F\ T'$
7 $F \rightarrow id$
8 $F \rightarrow (E)$

E	T	F	E'	T'
$				
			$	
	+, $			
		*		
				+
$,)				
		+	$,)	

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

Summary thus far

- Predictive parsers are intuitive
- Good error detection & recovery
- But limited in power
- Solutions:
  - Limited programming languages: Pascal, Oberon
  - Look-ahead
  - Transformations

A Simple Example

<E> → (<E>) | ID

	(	)	ID	$
<E>	<E> → (<E>)		<E>→id	

Left Factoring

- If a grammar contains two productions
  - <A> → a α | a β
    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
  - <A> → b C
  - <C> -> epsilon | a <C>
- 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 $E \rightarrow T\ E'$
2 $E' \rightarrow \varepsilon$
3 $E' \rightarrow +\ T\ E'$
4 $T \rightarrow F\ T'$
5 $T' \rightarrow \varepsilon$
6 $T' \rightarrow *\ F\ T'$
7 $F \rightarrow id$
8 $F \rightarrow (E)$

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	{(}

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"

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