Chapter 3 Syntax Analysis

Nai-Wei Lin

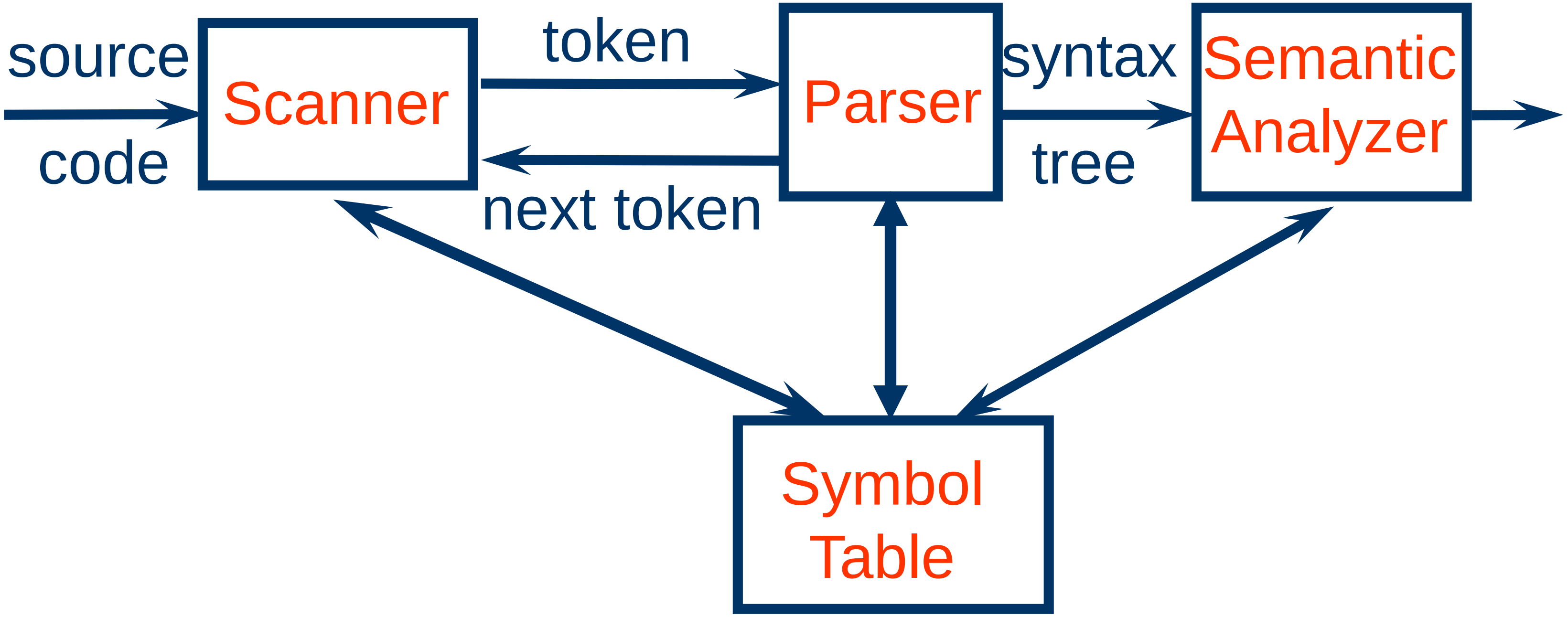
Syntax Analysis

- Syntax analysis recognizes the syntactic structure of the programming language and transforms a string of tokens into a tree of tokens and syntactic categories
- Parser is the program that performs syntax analysis

Outline

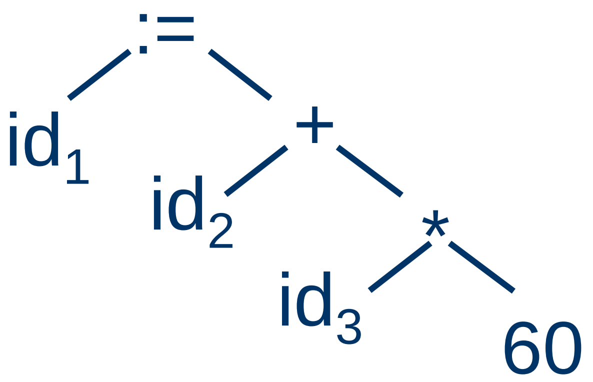
- Introduction to parsers
- Syntax trees
- Context-free grammars
- Push-down automata
- Top-down parsing
- A parser generator
- Bottom-up parsing

Introduction to Parsers



Syntax Trees

- A syntax tree represents the syntactic structure of tokens in a program defined by the grammar of the programming language



Context-Free Grammars (CFG)

- A set of terminals: basic symbols (token types) from which strings are formed
- A set of nonterminals: syntactic categories each of which denotes a set of strings
- A set of productions: rules specifying how the terminals and nonterminals can be combined to form strings
- The start symbol: a distinguished nonterminal that denotes the whole language

An Example: Arithmetic Expressions

- Terminals: **id**, ‘+’, ‘-’, ‘*’, ‘/’, ‘(’, ‘)’
- Nonterminals: *expr*, *op*
- Productions:

  $expr \rightarrow expr\ op\ expr$

  $expr \rightarrow$ ‘(’ $expr$ ‘)’

  $expr \rightarrow$ ‘-’ $expr$

  $expr \rightarrow \mathbf{id}$

  $op \rightarrow$ ‘+’ | ‘-’ | ‘*’ | ‘/’

- Start symbol: *expr*

An Example: Arithmetic Expressions

id $\Rightarrow$ { **id** },

'+' $\Rightarrow$ { + },

'-' $\Rightarrow$ { - },

'*' $\Rightarrow$ { * },

'/' $\Rightarrow$ { / },

'(' $\Rightarrow$ { (},

')' $\Rightarrow$ {) },

op $\Rightarrow$ { +, -, *, / },

expr $\Rightarrow$ { **id**, - **id**, (**id**), **id** + **id**, **id** - **id**, … }.

Derivations

- A derivation step is an application of a production as a rewriting rule, namely, replacing a nonterminal in the string by one of its right-hand sides, $N \to \alpha$

  $\ldots N \ldots \Rightarrow \ldots \alpha \ldots$

- Starting with the start symbol, a sequence of derivation steps is called a derivation

  $S \Rightarrow \ldots \Rightarrow \alpha$

  or $S \Rightarrow^{*} \alpha$

An Example

Grammar:

1. $expr \rightarrow expr\ op\ expr$
2. $expr \rightarrow$ ‘(’ $expr$ ‘)’
3. $expr \rightarrow$ ‘-’ $expr$
4. $expr \rightarrow$ **id**
5. $op \rightarrow$ ‘+’
6. $op \rightarrow$ ‘-’
7. $op \rightarrow$ ‘*’
8. $op \rightarrow$ ‘/’

Derivation:

$\underline{expr}$

$\Rightarrow -\ \underline{expr}$

$\Rightarrow -\ (\underline{expr}\ )$

$\Rightarrow -\ (\underline{expr}\ op\ expr\ )$

$\Rightarrow -\ (\ \mathbf{id}\ \underline{op}\ expr\ )$

$\Rightarrow -\ (\ \mathbf{id} + \underline{expr}\ )$

$\Rightarrow -\ (\ \mathbf{id} + \mathbf{id}\ )$

Left- & Right-Most Derivations

- If there are more than one nonterminal in the string, many choices are possible
- A leftmost derivation always chooses the leftmost nonterminal to rewrite
- A rightmost derivation always chooses the rightmost nonterminal to rewrite

An Example

Leftmost derivation:

$\underline{\mathit{expr}}$

$\Rightarrow - \underline{\mathit{expr}}$

$\Rightarrow - (\underline{\mathit{expr}}\ )$

$\Rightarrow - (\underline{\mathit{expr}}\ \mathit{op}\ \mathit{expr}\ )$

$\Rightarrow - (\mathbf{id}\ \underline{\mathit{op}}\ \mathit{expr}\ )$

$\Rightarrow - (\ \mathbf{id} + \underline{\mathit{expr}}\ )$

$\Rightarrow - (\ \mathbf{id} + \mathbf{id}\ )$

Rightmost derivation:

$\underline{\mathit{expr}}$

$\Rightarrow - \underline{\mathit{expr}}$

$\Rightarrow - (\underline{\mathit{expr}}\ )$

$\Rightarrow - (\mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}}\ )$

$\Rightarrow - (\mathit{expr}\ \underline{\mathit{op}}\ \mathbf{id})$

$\Rightarrow - (\underline{\mathit{expr}} + \mathbf{id}\ )$

$\Rightarrow - (\ \mathbf{id} + \mathbf{id}\ )$

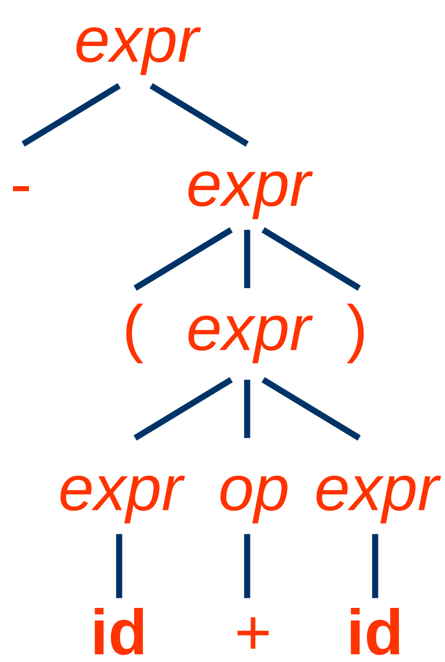
Parse Trees

- A parse tree is a graphical representation for a derivation that filters out the order of choosing nonterminals for rewriting
- Many derivations may correspond to the same parse tree, but every parse tree has associated with it a unique leftmost and a unique rightmost derivation

An Example

Leftmost derivation:

expr
⇒ - *expr*
⇒ - (*expr*)
⇒ - (*expr* *op* *expr*)
⇒ - (**id** *op* *expr*)
⇒ - (**id** + *expr*)
⇒ - (**id** + **id**)



Rightmost derivation:

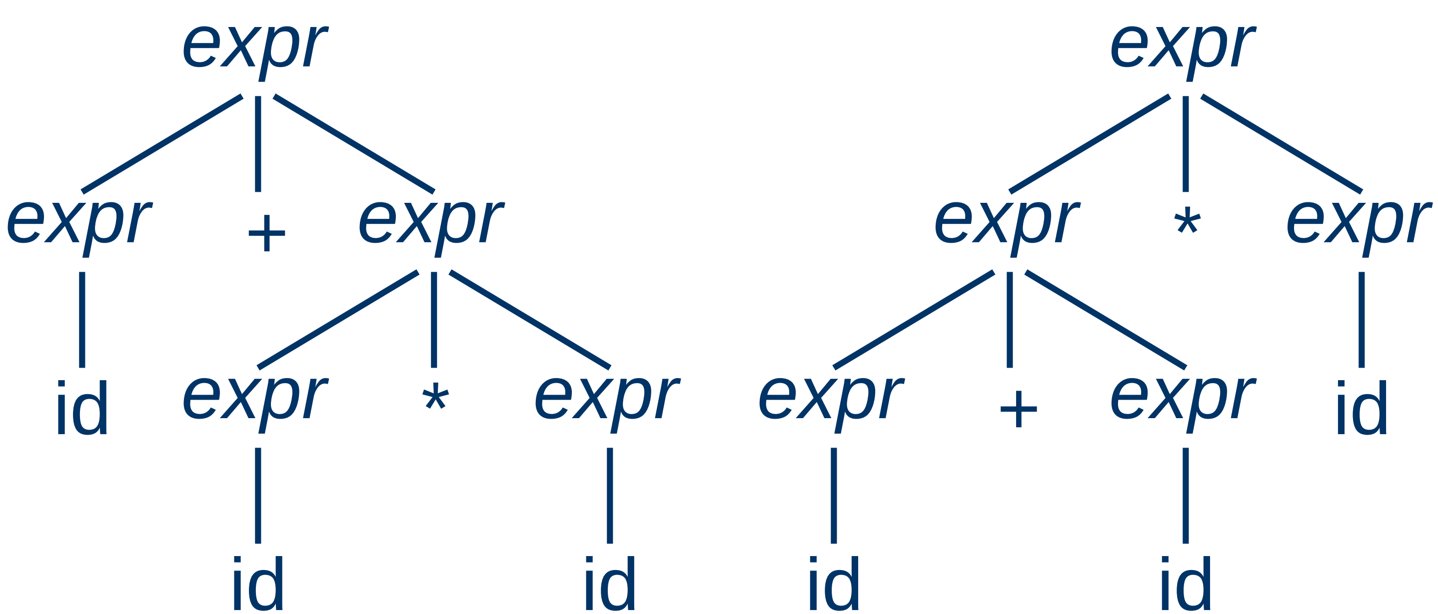
expr
⇒ - *expr*
⇒ - (*expr*)
⇒ - (*expr* *op* *expr*)
⇒ - (*expr* *op* **id**)
⇒ - (*expr* + **id**)
⇒ - (**id** + **id**)

Ambiguous Grammars

- A grammar is ambiguous if it can derive a string with two different parse trees
- If we use the syntactic structure of a parse tree to interpret the meaning of the string, the two parse trees have different meanings
- Since compilers do use parse trees to derive meaning, we would prefer to have unambiguous grammars

An Example

id + id * id



Transform Ambiguous Grammars

Ambiguous grammar:

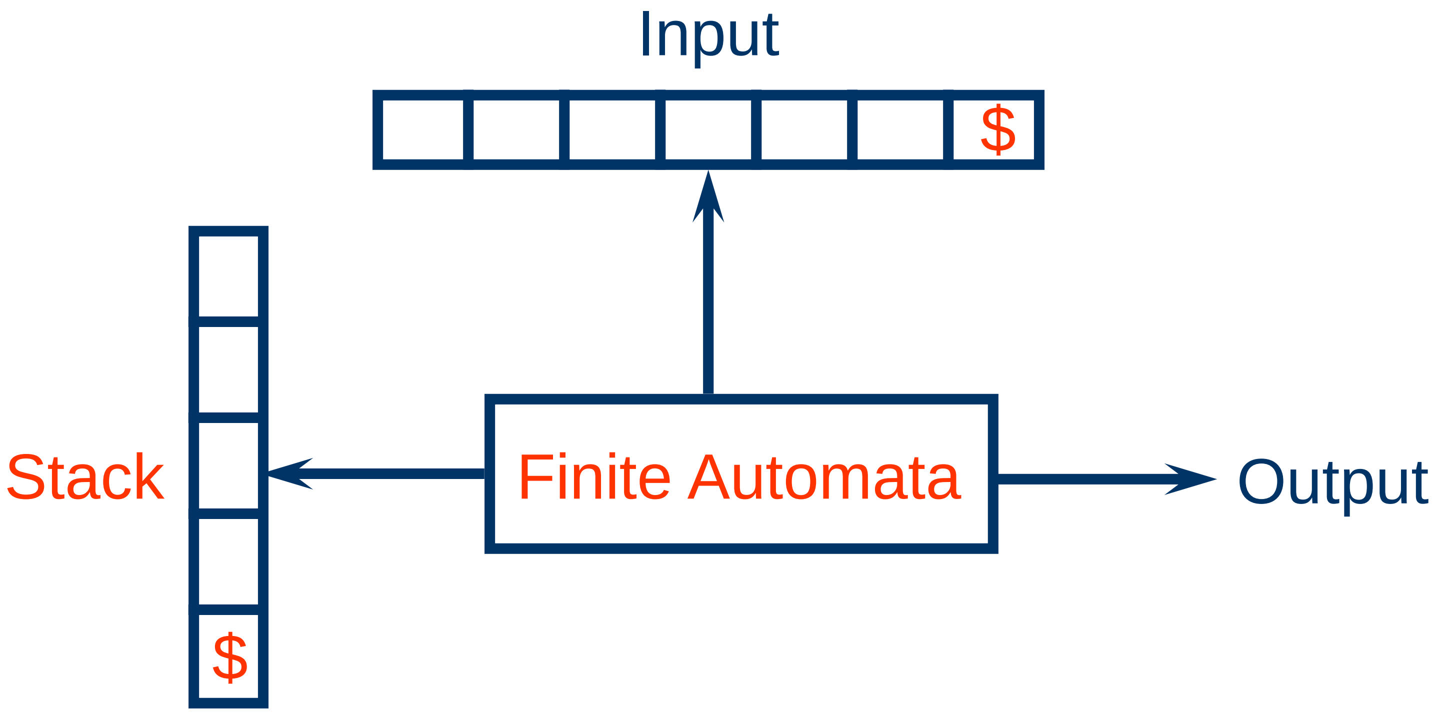
expr → *expr op expr*
expr → '(' *expr* ')'
expr → '-' *expr*
expr → id
op → '+' | '-' | '*' | '/'

Not every ambiguous grammar can be transformed to an unambiguous one!

Unambiguous grammar:

expr → *expr* '+' *term*
expr → *expr* '-' *term*
expr → *term*
term → *term* '*' *factor*
term → *term* '/' *factor*
term → *factor*
factor → '(' *expr* ')'
factor → '-' *expr*
factor → id

Push-Down Automata



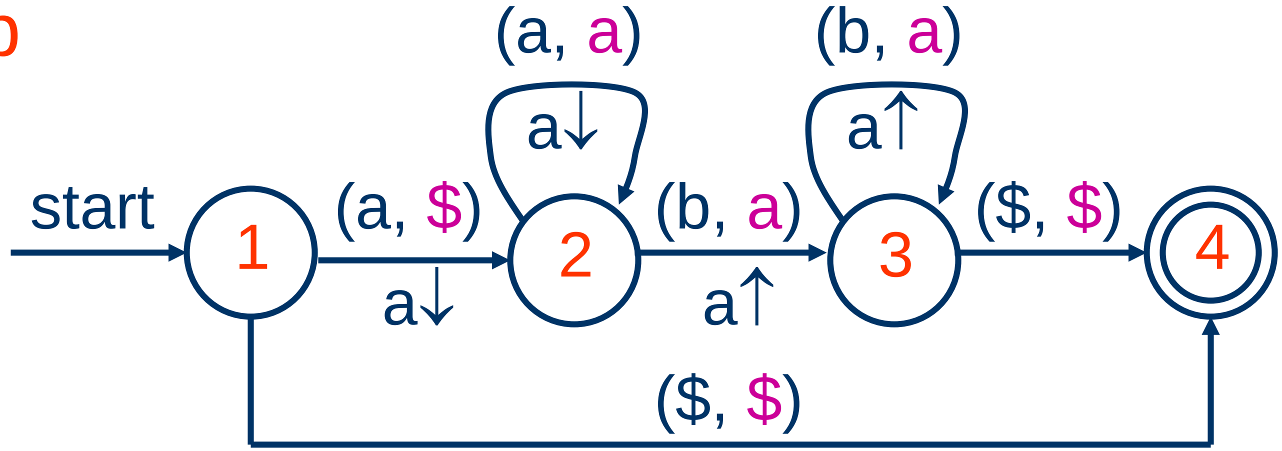
End-Of-File and Bottom-of-Stack Markers

- Parsers must read not only terminal symbols but also the end-of-file marker and the bottom-of-stack maker
- We will use $ to represent the end of file marker
- We will also use $ to represent the bottom-of-stack maker

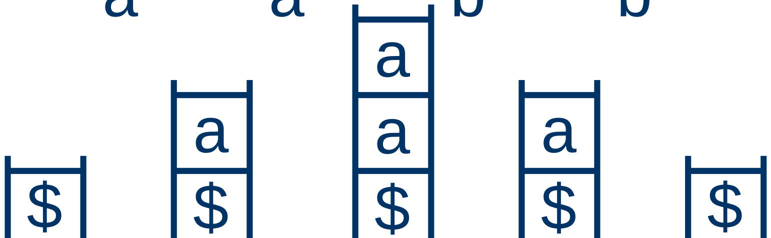
An Example

$S \rightarrow a\ S\ b$

$S \rightarrow \varepsilon$



$1 \xrightarrow[a]{} 2 \xrightarrow[a]{} 2 \xrightarrow[b]{} 3 \xrightarrow[b]{} 3 \xrightarrow[\$]{} 4$



CFG versus RE

- Every language defined by a RE can also be defined by a CFG
- Why use REs for lexical syntax?
  - Do not need a notation as powerful as CFGs
  - Are more concise and easier to understand than CFGs
  - More efficient lexical analyzers can be constructed from REs than from CFGs
  - Provide a way for modularizing the front end into two manageable-sized components

Nonregular Languages

- REs can denote only a fixed number of repetitions or an unspecified number of repetitions of one given construct

  a^n, a^*

- A nonregular language: $L = \{a^n b^n \mid n \geq 0\}$

  $S \rightarrow a\, S\, b$

  $S \rightarrow \varepsilon$

Top-Down Parsing

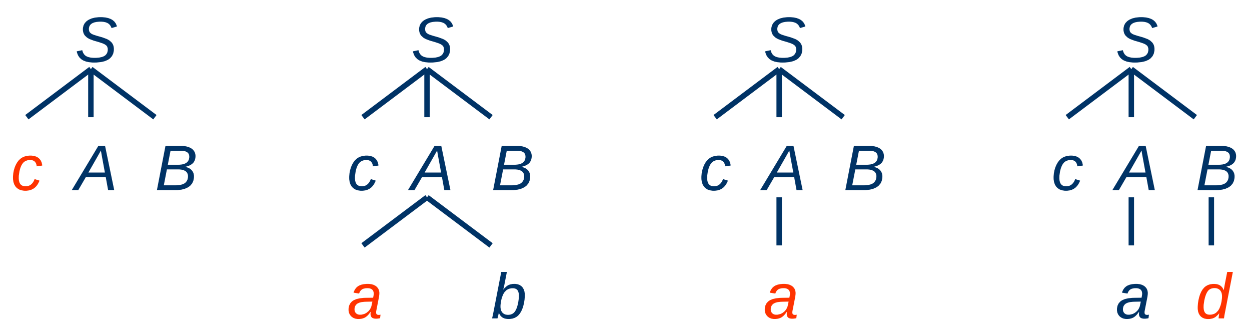
- Construct a parse tree from the root to the leaves using the leftmost derivation

$S \rightarrow c\,A\,B$

$A \rightarrow a\,b$ input: *cad*

$A \rightarrow a$

$B \rightarrow d$



Predictive Parsing

- Predictive parsing is a top-down parsing without backtracking
- Namely, according to the next token, there is only one production to choose at each derivation step

stmt → **if** *expr* **then** *stmt* **else** *stmt*
| **while** *expr* **do** *stmt*
| **begin** *stmt_list* **end**

LL(k) Parsing

- Predictive parsing is also called LL(k) parsing
- The first L stands for scanning the input from left to right
- The second L stands for producing a leftmost derivation
- The k stands for using k lookahead input symbol to choose alternative productions at each derivation step

LL(1) Parsing

- We will only describe LL(1) parsing from now on, namely, parsing using only one lookahead input symbol
- Recursive-descent parsing – hand written or tool (e.g. ANTLR and CoCo/R) generated
- Table-driven predictive parsing – tool (e.g. LISA and LLGEN) generated

Recursive Descent Parsing

- A procedure is associated with each nonterminal of the grammar
- An alternative case in the procedure is associated with each production of that nonterminal
- A match of a token is associated with each terminal in the right hand side of the production
- A procedure call is associated with each nonterminal in the right hand side of the production

Recursive Descent Parsing

$S \rightarrow$ **if** E **then** S **else** S

| **begin** L **end**

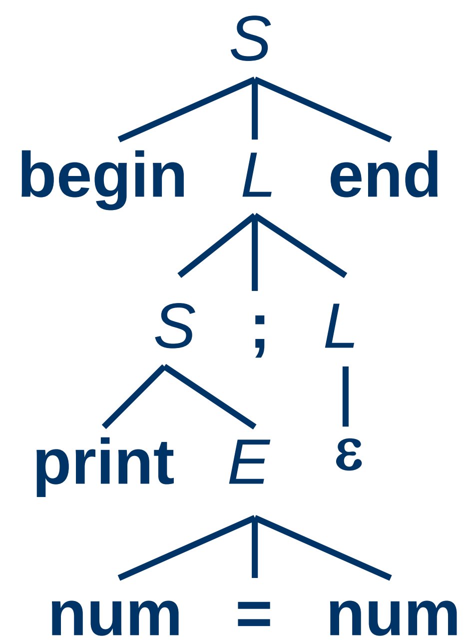
| **print** E

$L \rightarrow S$ **;** L

| ε

$E \rightarrow$ **num = num**

begin print num = num ; end



Choosing the Alternative Case

$S \to$ **if** E **then** S **else** S	FIRST(**if** E **then** ...) = {**if**}
| **begin** L **end**	FIRST(**begin** L **end**) = {**begin**}
| **print** E	FIRST(**print** E) = {**print**}
$L \to S$ **;** L	FIRST(S **;** L) = {**if**, **begin**, **print**}
| ε	FOLLOW(L) = {**end**}
$E \to$ **num = num**	FIRST(**num = num**) = {**num**}

An Example

```
const int
    IF = 1, THEN = 2, ELSE = 3, BEGIN = 4,
    END =5, PRINT = 6, SEMI = 7, NUM = 8,
    EQ = 9;
int token = lexer();

void match(int t)
{
    if (token == t) token = lexer(); else error();
}
```

An Example

```
void S() {
    switch (token) {
        case IF: match(IF); E(); match(THEN); S();
                    match(ELSE); S(); break;
        case BEGIN: match(BEGIN); L();
                    match(END); break;
        case PRINT: match(PRINT); E(); break;
        default: error();
    }
}
```

An Example

```
void L() {
   switch (token) {
      case IF:
      case BEGIN:
      case PRINT:
            S(); match(SEMI); L(); break;
      case END: break;
      default: error();
   }
}
```

An Example

```
void E() {
    switch (token) {
        case NUM:
            match(NUM); match(EQ); match(NUM);
            break;
        default: error();
    }
}
```

First and Follow Sets

- The first set of a string α, FIRST(α), is the set of terminals that can begin the strings derived from α. If $\alpha \Rightarrow^* \varepsilon$, then ε is also in FIRST(α)
- The follow set of a nonterminal X, FOLLOW(X), is the set of terminals that can immediately follow X

Computing First Sets

- If X is terminal, then FIRST(X) is {X}
- If X is nonterminal and $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X)
- If X is nonterminal and $X \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production, then add a to FIRST(X) if for some i, a is in FIRST(Y_i) and ε is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}). If ε is in FIRST(Y_j) for all j, then add ε to FIRST(X)

An Example

$S \to$ **if** E **then** S **else** S

$\quad$ | **begin** L **end**

$\quad$ | **print** E

$L \to S$ **;** $L \mid \varepsilon$

$E \to$ **num = num**

FIRST(E) = { **num** }

FIRST(L) = { **if**, **begin**, **print** , ε }

FIRST(S) = { **if**, **begin**, **print** }

Computing Follow Sets

- Place $ in FOLLOW(S), where S is the start symbol and $ is the end-of-file marker
- If there is a production $A \to \alpha B\beta$, then everything in FIRST(β) except for ε is placed in FOLLOW(B)
- If there is a production $A \to \alpha B$ or $A \to \alpha B\beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

An Example

$S \rightarrow$ **if** E **then** S **else** S

| **begin** L **end**

| **print** E

$L \rightarrow S$ **;** L | ε

$E \rightarrow$ **num = num**

FOLLOW(S) = { **$**, **else**, ; }

FOLLOW(L) = { **end** }

FOLLOW(E) = { **then**, **$**, **else**, ; }

Table-Driven Predictive Parsing

Input. Grammar G.   Output. Parsing Table M.

Method.

1. For each production $A \rightarrow \alpha$ of the grammar,

   do steps 2 and 3.

2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ε is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ε is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be error.

An Example

	S	*L*	*E*
if	$S \rightarrow$ **if** E **then** S **else** S	$L \rightarrow S$ **;** L	
then			
else			
begin	$S \rightarrow$ **begin** L **end**	$L \rightarrow S$ **;** L	
end		$L \rightarrow \varepsilon$	
print	$S \rightarrow$ **print** E	$L \rightarrow S$ **;** L	
num			$E \rightarrow$ **num = num**
;			
$			

An Example

Stack	Input
$ S	begin print num = num ; end $
$ end L begin	begin print num = num ; end $
$ end L	print num = num ; end $
$ end L ; S	print num = num ; end $
$ end L ; E print	print num = num ; end $
$ end L ; E	num = num ; end $
$ end L ; num = num	num = num ; end $
$ end L ;	; end $
$ end L	end $
$ end	end $
$	$

LL(1) Grammars

- A grammar is LL(1) iff its predictive parsing table has no multiply-defined entries
- A grammar G is LL(1) iff whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  (1) $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \emptyset$,
  (2) If $\varepsilon \in \mathrm{FIRST}(\alpha)$, $\mathrm{FOLLOW}(A) \cap \mathrm{FIRST}(\beta) = \emptyset$,
  (3) If $\varepsilon \in \mathrm{FIRST}(\beta)$, $\mathrm{FOLLOW}(A) \cap \mathrm{FIRST}(\alpha) = \emptyset$.

A Counter Example

$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S' \mid \mathbf{a}$
$S' \rightarrow \mathbf{e}\ S \mid \varepsilon$
$E \rightarrow \mathbf{b}$

	a	**b**	**e**	**i**	**t**	**$**
S	$S \rightarrow \mathbf{a}$			$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S'$		
S'			$S' \rightarrow \varepsilon$ $S' \rightarrow \mathbf{e}\ S$			$S' \rightarrow \varepsilon$
E		$E \rightarrow \mathbf{b}$				

$\varepsilon \in \mathrm{FIRST}(\varepsilon) \wedge \mathrm{FOLLOW}(S') \cap \mathrm{FIRST}(\mathbf{e}\ S) = \{\mathbf{e}\} \neq \varnothing$

Left Recursive Grammars

- A grammar is left recursive if it has a nonterminal A such that $A \Rightarrow^{*} A\ \alpha$
- Left recursive grammars are not LL(1) because

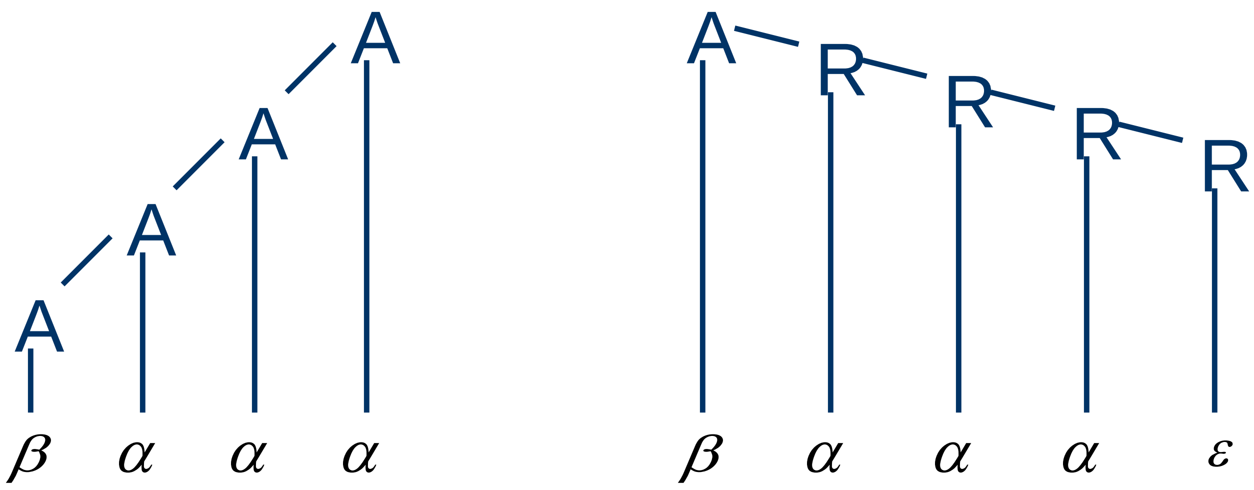
  $A \rightarrow A\ \alpha$

  $A \rightarrow \beta$

  will cause $\text{FIRST}(A\ \alpha) \cap \text{FIRST}(\beta) \neq \emptyset$
- We can transform them into LL(1) by eliminating left recursion

Eliminating Left Recursion

$$A \rightarrow A\,\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{l} A \rightarrow \beta\, R \\ R \rightarrow \alpha\, R \mid \varepsilon \end{array}$$



Direct Left Recursion

$$A \rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

$$\Downarrow$$

$$A \rightarrow \beta_1\,A' \mid \beta_2\,A' \mid \ldots \mid \beta_n\,A'$$

$$A' \rightarrow \alpha_1\,A' \mid \alpha_2\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$$

An Example

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

$\Downarrow$

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$

Indirect Left Recursion

$S \rightarrow A\,a \mid b$

$A \rightarrow A\,c \mid S\,d \mid \varepsilon$

$S \Rightarrow A\,a \Rightarrow S\,d\,a$

$A \rightarrow A\,c \mid A\,a\,d \mid b\,d \mid \varepsilon$

$\Downarrow$

$S \rightarrow A\,a \mid b$

$A \rightarrow b\,d\,A' \mid A'$

$A' \rightarrow c\,A' \mid a\,d\,A' \mid \varepsilon$

Left factoring

- A grammar is not LL(1) if two productions of a nonterminal A have a nontrivial common prefix. For example, if $\alpha \neq \varepsilon$, and $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2$, then $\mathrm{FIRST}(\alpha\ \beta_1) \cap \mathrm{FIRST}(\alpha\ \beta_2) \neq \varnothing$
- We can transform them into LL(1) by performing left factoring

$$A \rightarrow \alpha\ A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

An Example

$$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S \mid \mathbf{i}\ E\ \mathbf{t}\ S\ \mathbf{e}\ S \mid \mathbf{a}$$
$$E \rightarrow \mathbf{b}$$

⇩

$$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S' \mid \mathbf{a}$$
$$S' \rightarrow \mathbf{e}\ S \mid \varepsilon$$
$$E \rightarrow \mathbf{b}$$

Parser Rules

- Parser rule names must begin with a lowercase letter.

```
parserRuleName :
    alternative1 | ... | alternativeN ;
```

Parser Rule Elements

- T: Match token T at the current input position.
- ’literal’: Match the string literal at the current input position.
- r: Match rule r at current input position, which amounts to invoking the rule just like a function call.

An Example

```
program : MAIN '(' ')' '{' declarations statements '}' ;
declarations : INT ID SEMI declarations
        |
        ;
statements : statement statements
        |
        ;
statement : READ ID SEMI
        | RETURN SEMI
        ;
```

Parser Rule Elements

- {«action»}: Execute an action immediately after the preceding rule element and immediately before the following rule element.
- The action conforms to the syntax of the target language.
- ANTLR copies the action code to the generated class verbatim .

Bottom-Up Parsing

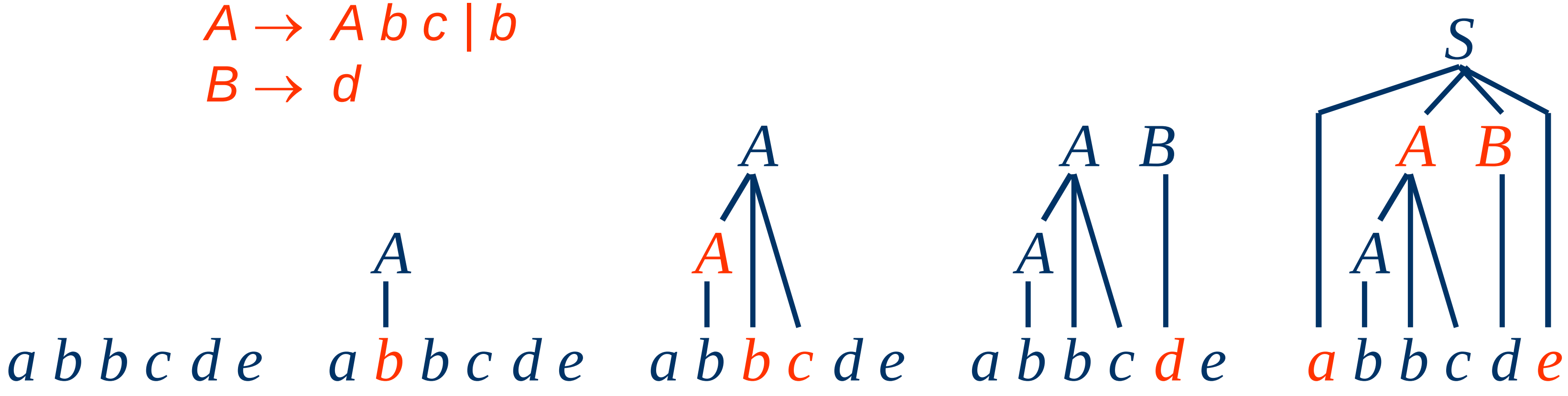
- Construct a parse tree from the leaves to the root using rightmost derivation in reverse

$S \rightarrow a\ A\ B\ e$

$A \rightarrow A\ b\ c \mid b$

$B \rightarrow d$

input: *abbcde*



$$abbcde \Leftarrow aAbcde \Leftarrow aAde \Leftarrow aABe \Leftarrow S$$

Hierarchy of Grammar Classes