Describing Syntax of Programming Languages

Two Aspects of Programming Languages

- **Syntax**
  - The form or structure of the expressions, statements, and program units
- **Semantics**
  - The meaning of the expressions, statements, and program units

Describing Syntax and Semantics of Programming Languages

- How to describe Syntax and Semantics of programming languages?
  - Precisely
  - Unambiguously
- Need a formal notation other than natural languages!

Who Must Use Programming Language Definitions?

- Language designers
- Language implementors
- Programmers (the users of the language)

A Formal Language

- A language is defined over **an alphabet**.
  - An alphabet is a finite set of characters (symbols).
- **A language** is **a set of strings** of characters (symbols) from some alphabet.
  - The string is called a sentence.
- Two types of languages:
  - **Finite languages**
  - **Infinite languages**

Structure of Programming Languages

- **Lexical structure**
- **Syntactic structure**

Lexemes and Tokens in A Programming Language

- A **lexeme** is the lowest level syntactic unit of a programming language.
- A lexical **token** of a programming language is a category of its lexemes.
  - A sequence of characters that can be treated as a unit in the grammar of a programming language
  - A basic syntactic unit

Example: Lexemes and Tokens

- `foo n14 last ….`
  - **Token ID**
- `73 0 515 …`
  - **Token NUM**
- `66.1 .5 10. 1e67 5.5e-10`
  - **Token REAL**
- `if`
  - **Token IF**
- `,`
  - **Token COMMA**
- `(`
  - **Token LPREN**
- `)`
  - **Token RPREN**

A Programming Language

- A *programming language* is a set of strings of lexemes from some alphabet.
- A *programming language* has a finite set of tokens.

Example: Tokens

```
float match0(char *s) /* find a zero */
{ if (!strncmp(s, "0.0", 3))
   return 0.;
}
```

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN LBRACE IF LPAREN BANG ID(strncmp) LPAREN ID(s) COMMA STRING(0.0) COMMA NUM(3) RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE

How to Describe Syntax

- An informal way.
- A formal method
  - Need a formal notation other than natural languages!
    - A **metalanguage** is a language used to describe another language.

Approaches to Describing Syntax Formally

- Two Approaches:
  - Define by **Language Generators**
    - How to **generate** all valid strings (sentences)?
  - Define by **Language Recognizers**
    - How to **recognize** all valid strings (sentences)?

Language Generator

Methods for Language Generator

- Formal language generation mechanisms:
  1. **Context-Free Grammars**
  2. **BNF (Backus-Naur Form)**
  3. **EBNF(Extended BNF)**
  4. **Syntax Graphs**

1. Context-free Grammars

- Developed by Noam Chomsky in the mid-1950s
  - To describe the syntax of natural languages.

Languages and Grammars

- Four classes of languages
  - **Regular languages**
  - **context-free languages**
  - **context-sensitive languages**
  - **Recursively enumerable languages**
- Four classes of grammars
  - **Regular grammars**
  - **Context-free grammars**
  - **Context-sensitive grammars**
  - **Unrestricted grammars**

Grammars

- Grammar G = (T, N, S, P)
  - T (Terminals)
  - N (Nonterminals)
  - S (Starting symbol)
  - P (Production rules): LHS $\rightarrow$ RHS

Context-Free Languages and Context-Free Grammars

- The syntax of a programming language is
  - **Syntactic structure: Context-free language**
  - **Lexical structure: Regular language**
- We use to describe the syntax of a programming language
  - **Context-free grammars**
  - **Regular grammars**

Context-Free Grammars

- Context free grammar G = (T, N, S, P)
  - T (Terminals)
  - N (Nonterminals)
  - S (Starting symbol)
  - P (Production rules): **LHS** → **RHS**

A single nonterminal!

2. Backus Normal Form (BNF)

- Invented by John Backus to describe Algol 58.
- Modified by Peter Naur to describe Algol 60.
  - **B**ackus-**N**aur **F**orm (**BNF**)
- BNF is equivalent to Context-Free Grammars!

CFG and BNF & PLs

- They are sufficiently powerful to describe the majority of the syntax of all programming languages!

Example: A CFG for a Small Language

T = {**begin**, **end**, **;**, **:=**, **A**, **B**, **C**, **+**, **-**}
N = {<program>, <stmt_list>, <stmt>, <var>, <expression>}
S = <program>
P = {<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> **;** <stmt_list>
<stmt> → <var> **:=** <expression>
<var> → **A** | **B** | **C**
<expression> → <var> **+** <var>
| <var> **-** <var>
| <var>
}

Example: CFGs

- A Simple C Assignment Statement:

<assign> → <var> = <expression>

Example: CFGs

- A Pascal if Statement:

<if_stmt> → if <logic_expr> then <stmt>
<if_stmt> → if <logic_expr> then <stmt> else <stmt>
or
<if_stmt> → if <logic_expr> then <stmt>
| if <logic_expr> then <stmt> else <stmt>

Example: CFGs

- An Identifier List:

<iden_list> → identifier
| identifier , <ident_list>

How to Generate From CFG

- Starting from the starting symbol.
- Apply the production rules repeatedly.
- Until a sentence (all terminal symbols).

– **Derivation**!

Derivation

- Every string of symbols in the derivation is a *sentential form.*
- A *sentence* is a sentential form that has only terminal symbols.
- A **leftmost derivation** is one in which the **leftmost nonterminal** in each sentential form is the one that is expanded.

Example: Leftmost derivation of
begin A := B + C; B := C end

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> **;** <stmt_list>
<stmt> → <var> **:=** <expression>
<var> → **A** | **B** | **C**
<expression> → <var> + <var>
| <var> - <var>
| <var>

<program>
⇒ **begin** <stmt_list> **end**
⇒ **begin** <stmt> **;** <stmt_list> **end**
⇒ **begin** <var> := <expression> **;** <stmt_list> **end**
⇒ **begin** A := <expression> **;** <stmt_list> **end**
⇒ **begin** A := <var> + <var> **;** <stmt_list> **end**
⇒ **begin** A := B + <var> **;** <stmt_list> **end**

Example: Leftmost derivation of
begin A := B + C; B := C end

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> **;** <stmt_list>
<stmt> → <var> **:=** <expression>
<var> → **A** | **B** | **C**
<expression> → <var> + <var>
| <var> - <var>
| <var>

⇒ **begin** A := B + C **;** <stmt_list> **end**
⇒ **begin** A := B + C **;** <stmt> **end**
⇒ **begin** A := B + C **;** <var> := <expression> **end**
⇒ **begin** A := B + C **;** B := <expression> **end**
⇒ **begin** A := B + C **;** B := <var> **end**
⇒ **begin** A := B + C **;** B := C **end**

Example: A CFG for Simple Assignment Statements

T = {**:=**, **A**, **B**, **C**, +, *, **(**, **)**}
N = {<assign>, <id>, <expr> }
S = <assign>
P = { <assign> → <id> := <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

QUIZ: Leftmost derivation of
A := B * (A + C)

<assign>
⇒ <id> **:=** <expr>
⇒ **A :=** <expr>
⇒ **A :=** <id> * <expr>
⇒ **A := B** * <expr>
⇒ **A := B** * (<expr>)
⇒ **A := B** * (<id> + <expr>)
⇒ **A := B** * (**A** + <expr>)
⇒ **A := B** * (**A** + <id>)
⇒ **A := B * (A + C)**

<assign> → <id> := <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

The Derivation Order

- **Left-most derivation**
  - Choose the left-most nonterminal symbol for replacement.
- **Right-most derivation**
  - Choose the right-most nonterminal symbol for replacement.
- **Derivation order has no effect on the language generated by the grammar!**

Example: Rightmost derivation of
begin A := B + C; B := C end

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> **;** <stmt_list>
<stmt> → <var> **:=** <expression>
<var> → **A | B | C**
<expression> → <var> + <var>
| <var> - <var>
| <var>

<program>
⇒ **begin** <stmt_list> **end**
⇒ **begin** <stmt> **;** <stmt_list> **end**
⇒ **Begin** <stmt> **;** <stmt> **end**
⇒ **begin** <stmt> **;** <var> := <expression> **end**
⇒ **begin** <stmt> **;**<var> := <expression> **end**
⇒ **begin** <stmt> **;**<var> := <var> **end**

Example: Rightmost derivation of
begin A := B + C; B := C end

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> **;** <stmt_list>
<stmt> → <var> **:=** <expression>
<var> → **A | B | C**
<expression> → <var> + <var>
| <var> - <var>
| <var>

⇒ **begin** <stmt> **;** <var> := C **end**
⇒ **begin** <var> := <expression> **;** B := C **end**
⇒ **begin** <var> := <var> + <var> **;** B := C **end**
⇒ **begin** <var> := <var> + C **;** B := C **end**
⇒ **begin** <var> := B + C **;** B := C **end**
⇒ **begin** A := B + C **;** B := C **end**

QUIZ: Right-most derivation of
A := B * (A + C)

<assign>
⇒<id> **:=** <expr>
⇒.
⇒.
⇒.
⇒.
⇒**A := B * (A + C)**

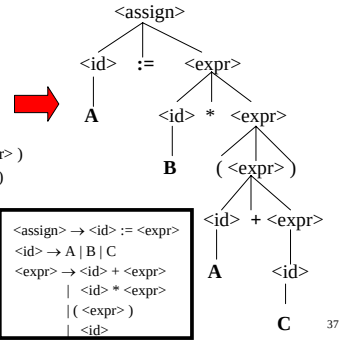
<assign> → <id> := <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Parse Tree

- A **parse tree** is
  - a hierarchical representation of a derivation.
- Leftmost and rightmost derivations for the same string produce the same parse tree.
  - If the grammar is **unambiguous**.

Example: Parse Tree of A := B * (A + C)

<assign>
⇒ <id> := <expr>
⇒ **A** := <expr>
⇒ **A** := <id> * <expr>
⇒ **A** := **B** * <expr>
⇒ **A** := **B** * (<expr>)
⇒ **A** := **B** * (<id> + <expr>)
⇒ **A** := **B** * (**A** + <expr>)
⇒ **A** := **B** * (**A** + <id>)
⇒ **A** := **B** * (**A** + **C**)



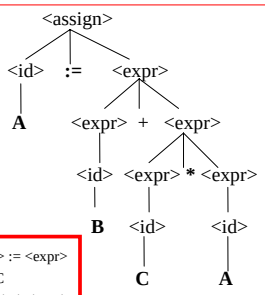
<assign> → <id> := <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Example: A CFG for Simple Assignment Statements

T = {**:=**, **A**, **B**, **C**, **+**, *****, **(**, **)**}
N = {<assign>, <id>, <expr> }
S = <assign>
P = { <assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Example: Leftmost derivation I & Parse Tree of A := B + C * A

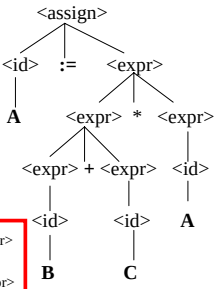
<assign>
⇒ <id> := <expr>
⇒ **A** := <expr>
⇒ **A** := <expr> + <expr>
⇒ **A** := <id> + <expr>
⇒ **A** := **B** + <expr>
⇒ **A** := **B** + <expr> * <expr>
⇒ **A** := **B** + <id> * <expr>
⇒ **A** := **B** + **C** * <expr>
⇒ **A** := **B** + **C** * <id>
⇒ **A** := **B** + **C** * **A**



<assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Example: Leftmost derivation II & Parse Tree of A := B + C * A

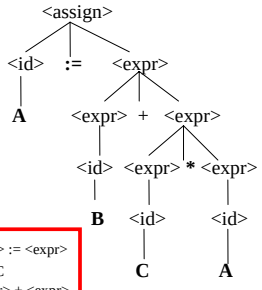
<assign>
⇒ <id> := <expr>
⇒ **A** := <expr>
⇒ **A** := <expr> * <expr>
⇒ **A** := <expr> + <expr> * <expr>
⇒ **A** := <id> + <expr> * <expr>
⇒ **A** := **B** + <expr> * <expr>
⇒ **A** := **B** + <id> * <expr>
⇒ **A** := **B** + **C** * <expr>
⇒ **A** := **B** + **C** * <id>
⇒ **A** := **B** + **C** * **A**



<assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

QUIZ: Rightmost derivation I & Parse Tree of A := B + C * A

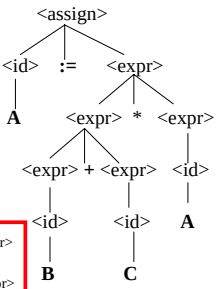
<assign>
⇒ <id> := <expr>
⇒ .
⇒ .
⇒ .
⇒ **A** := **B** + **C** * **A**



<assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

QUIZ: Rightmost derivation II & Parse Tree of A := B + C * A

<assign>
⇒ <id> := <expr>
⇒ .
⇒ .
⇒ .
⇒ **A** := **B** + **C** * **A**



<assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Example: A CFG for Simple Assignment Statements

T = {:=, A, B, C, +, *, (,)}
N = {<assign>, <id>, <expr> }
S = <assign>
P = { <assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Two different parse trees for the same string **A := B + C * A!!!**

Ambiguous Grammar!

Ambiguous Grammar

- A grammar is **ambiguous** iff it generates a string (sentence) that has two or more distinct parse trees.
- **Ambiguity is a problem!**
  - Because the meaning of the string cannot be determined uniquely.

Example: Another CFG for Simple Assignment Statements

T = {:=, A, B, C, +, *, (,)}
N = {<assign>, <id>, <expr> }
S = <assign>
P = { <assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Ambiguous Grammar!

T = {:=, A, B, C, +, *, (,)}
N = {<assign>, <id>, <expr> }
S = <assign>
P = { <assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

Unambiguous Grammar!

QUIZ: Leftmost Derivation of A := B + C * A

<assign>
⇒<id> := <expr>
⇒.
⇒.
⇒.
⇒**A := B * (A + C)**

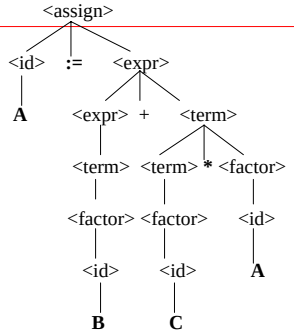
T = {:=, A, B, C, +, *, (,)}
N = {<assign>, <id>, <expr> }
S = <assign>
P = { <assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

QUIZ: Rightmost Derivation of A := B + C * A

<assign>
⇒<id> := <expr>
⇒.
⇒.
⇒.
⇒**A := B * (A + C)**

T = {:=, A, B, C, +, *, (,)}
N = {<assign>, <id>, <expr> }
S = <assign>
P = { <assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

Example: Parse Tree of A := B + C * A



Only one parse tree for the same string A := B + C * A!!!

Example: An ambiguous CFG for if Statements

P = { <stmt> → <if_stmt>
<if_stmt> → if <logic_expr> then <stmt>
| if <logic_expr> then <stmt> else <stmt>
}

Ambiguous Grammar!

Example: An unambiguous CFG for if Statements

P = { <stmt> → <matched> | <unmatched>

<matched> → if <logic_expr> then <matched> else <matched>

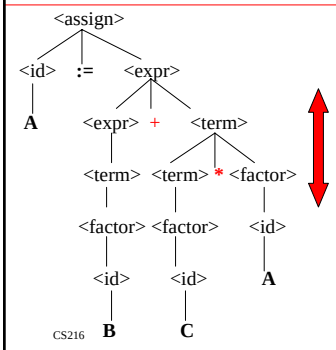
<unmatched> → if <logic_expr> then <stmt>
| if <logic_expr> then <matched> else <unmatched>
}

Unambiguous Grammar!

Operator Precedence

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity.
- Operator with **higher precedence**
  - **Down** in the parse tree
- Operator with **lower precedence**
  - **Up** in the parse tree

Example: Parse Tree of A := B + C * A

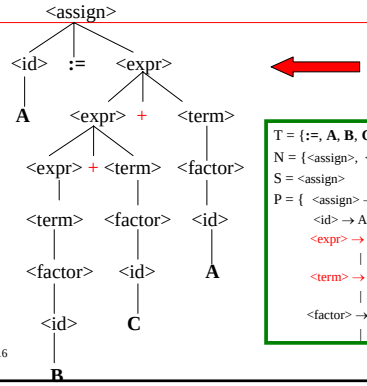


T = {:=, **A**, **B**, **C**, +, *, (,)}
N = {<assign>, <id>, <expr> }
S = <assign>
P = { <assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

Operator Associativity

- **Left associative**
  - **Left recursive** production rule
- **Right associative**
  - **Right recursive** production rule

Example: Parse Tree of A := B + C + A



T = {:=, **A**, **B**, **C**, +, *, (,)}
N = {<assign>, <id>, <expr> }
S = <assign>
P = { <assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

QUIZ: Syntax of Prolog?

Syntax of Prolog

```
<term>  ::= <constant>
          | <variable>
          | <compound-term>

<constant>  ::= <integer>
              | <real number>
              | <atom>

<compound-term>  ::= <atom> ( <termlist> )

<termlist> ::= <term> | <term> , <termlist>
```


Facts

```
<fact>  ::=  <term>  .
```


Rules

```
<rule>  ::=  <term>  :-  <termlist>  .

<termlist>  ::= <term>  |  <term>  ,  <termlist>
```


A Prolog Program

```
<clause>  ::= <fact>
            | <rule>
```


3. Extended BNF (EBNF)

- BNF + Some extensions:
  - **Optional** parts are placed in brackets.
    - […]
  - **Alternative** parts are put in parentheses and separated with vertical bars.
    - (… | …)
  - **Repetitive** (0 or more) parts are put in braces.
    - { … }

EBNF

- Optional parts:

  <proc_call> → <identifier> [(<expr_list>)]

  <selection> → if (<exp>) <statement> [else <statement>] ;

EBNF

- Alternative parts:

  <for_stmt> → for <var> := <exp> (to | downto) <exp> do <stmt>

  <term> → <term> (+ | -) const

EBNF

- Repetitive (0 or more) parts:

  <ident_list> → <identifier> { , <identifier> }

Extended BNF (EBNF)

- These extensions do not enhance the descriptive power of BNF.
- These extensions only increase readability and writability!

Example: BNF to EBNF

BNF:

<expr> → <expr> + <term>
| <expr> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>

EBNF:

<expr> → <term> { (+ | -) <term> }
<term> → <factor> { (* | /) <factor> }

4. Syntax Graphs/Diagrams/Charts

- Use a directed graph:
  - Put the terminals in circles or ellipses.
  - Put the nonterminals in rectangles.
  - Connect with lines with arrowheads for production rules.
- A separate syntax graph for each syntactic unit.
- Increases readability.

Language Recognizer

How to Recognize the Syntax of Programming Languages?

- Recognizing the **lexical structure** of a programming language.
  - **Lexical analysis**
  - **Lexer** or **scanner**
- Recognizing the **syntactic structure** of a programming language.
  - **Syntax analysis**
  - **Parser**

Recognizing the Structure of the Program

- Scanner (Lexical Analyzer)
  - Input: A stream of characters
  - Output: A stream of tokens
- Parser (Syntax Analyzer)
  - Input: A stream of tokens
  - Output: Valid string or not + (A parse tree)

Scanning

Lexical Analyzer (Scanner)

- Input:
  - A stream of characters (symbols)
- Output:
  - A stream of tokens
- Discards white space, blanks, tabs, newlines and comments.

Example: Lexical Analyzer

- float match0(char *s) /* find a zero */ { if (!strncmp(s, “0.0”, 3)) return 0.;}

- FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN LBRACE IF LPAREN BANG ID(strncmp) LPAREN ID(s) COMMA STRING(0.0) COMMA NUM(3) RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE

Tokens

- How to describe (specify) tokens precisely?
  - Informal specification – Imprecise & ambiguous!
  - Specify the syntax of tokens formally
- How to recognize tokens?
  - Recognize the syntax of tokens

Tokens are Regular Languages

- A **token** is a simple language called a **regular language**.
  - How to the describe (specify) regular languages precisely?
    - Specify the syntax of regular languages
  - How to recognize regular languages?
    - Recognize syntax of regular languages

Parsing

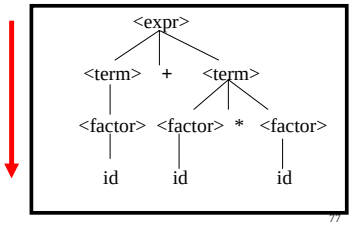
Approach to Parsing

- For a given input string:
- Discover the parse tree in a top-down fashion.
  - **Top-down parsing**
- Discover the parse tree in a bottom-up fashion.
  - **Bottom-up parsing**

Top –Down Parsing

<expr> -> <term> {(+ | -) <term>}
<term> -> <factor> {(* | /) <factor>}
<factor> -> id | (<expr>)

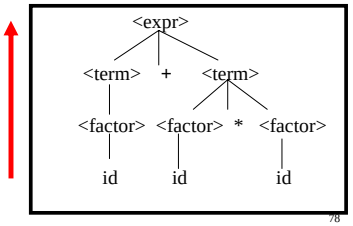
id + id * id



Bottom-Up Parsing

<expr> -> <term> {(+ | -) <term>}
<term> -> <factor> {(* | /) <factor>}
<factor> -> id | (<expr>)

id + id * id



Two Approaches to Parsing

- **Top-Down Parsing**
  - Try to construct the parse tree from top to bottom.
- **Bottom-Up Parsing**
  - Try to construct the parse tree from bottom to top.

Top-Down Parsing

- **LL(1) parsing**

LL(1) **:** Scanning direction of input string L→R
L**L**(1) **:** Derivation Rule – Left most derivation
LL**(1)** **:** Symbols at a time

- **Recursive Descent parsing**

Bottom-Up Parsing

powerful but difficult

- **LR(1) parsing**

LR(1): Scanning direction of input string L→R
L**R**(1): Derivation Rule – Right most derivation
LR**(1)**: Symbols at a time – one symbol at a time

- **SLR(1)**: Simple LR(1)
- **LALR(1)**: Look Ahead LR(1)

Top-Down Parsing

Recursive Descent Parsing: A Simple Top-Down Parsing

- A recursive descent parsing method traces out a parse tree in top-down order.
- A recursive descent parsing method is a simple top-down parsing method.

Recursive Descent Parser

- Each non-terminal in the grammar has a subprogram associated with it.
- The subprogram parses all sentential forms that the non-terminal can generate.
- The parsing subprograms are built directly from the grammar rules.
- The parsing subprograms are often recursive!

Example: Recursive Descent Parser

<expr> -> <term> {(+ | -) <term>}
<term> -> <factor> {(* | /) <factor>}
<factor> -> id | (<expr>)

```
void expr()
void term()
void factor()

Main:
  lexical();
  expr();
```


<expr> -> <term> {(+ | -) <term>}

```
void expr() {
    term();  /* parse the first term*/
    while (next_token == plus_code ||
           next_token == minus_code) {
      lexical();  /* get next token */
      term();  /* parse the next term */
    }
}
```


<term> -> <factor> {(* | /) <factor>}

```
void term() {
    factor();  /* parse the first factor*/
    while (next_token == ast_code ||
           next_token == slash_code) {
      lexical();  /* get next token */
      factor();  /* parse the next factor */
    }
}
```


<factor> -> id | (<expr>)

```
void factor() {
    if (next_token == id_code) {
    lexical();
    return;
    }
    else if (next_token == left_paren_code) {
     lexical();
     expr();
     if (next_token == right_paren_code) {
      lexical();
      return;
      }
     else error(); /* expecting right ) */
    }
    else error(); /* neither id nor ( */
}
```


Recursive Descent Parser

- Recursive descent parsers cannot be built from **left-recursive** grammars!

<expr> → <expr> + <term>
| <expr> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>

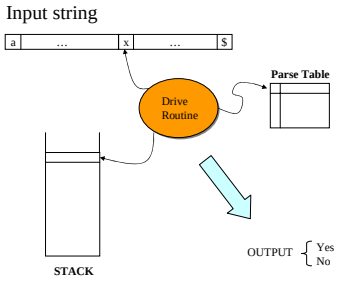
Recursive Descent Parser

<expr> → <expr> + <term>
| <expr> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>

?

```
void expr() {
    expr();
    if (next_token == +) {
      lexical();
      term();
    }
…
}
```

LL(1) Parser



LL(1) Parser

Drive Routine

Let a = the current_input_symbol
X=the symbol on top of the STACK

1. Push $
2. Push S // S is the Starting Symbol
3. While (not_empty(STACK))
   i) X== A terminal or $
      if X==a then pop(X);
      Move to next_input_symbol;
   ii) X==A non-Terminal
      if LL(1)Table[X,a]=X → α then pop(X);
      push(α) in reverse order;
      else error();
   end While
4. Output "YES_ACCEPT"

Example: Removal of Left-Recursive Productions

E → E + T | T
T→ T * F | F
F → id | (E)

E → TQ
Q → +TQ | ε
T → FR
R → *FR | ε
F → id | (E)

E → TE'
E' → ε | +TE'
T → FT'
T' → ε | *FT'
F → id | (E)

Example: The LL(1) Parse Table

E → TQ
Q → +TQ | ε
T → FR
R → *FR | ε
F → id | (E)

	id	(	*	+	)	$
E	E →TQ	E →TQ				
Q				Q →+TQ	Q → ε	Q → ε
T	T →FR	T →FR				
R			R →*FR	R → ε	R → ε	R → ε
F	F → id	F → (E)				

Example: LL(1) Parser

: ERROR

	id	(	*	+	)	$
E	E →TQ	E →TQ				
Q				Q →+TQ	Q → ε	Q → ε
T	T →FR	T →FR				
R			R →*FR	R → ε	R → ε	R → ε
F	F → id	F → (E)				

Example: LL(1) Parser

	id	(	*	+	)	$
E	E →TQ	E →TQ				
Q				Q →+TQ	Q → ε	Q → ε
T	T →FR	T →FR				
R			R →*FR	R → ε	R → ε	R → ε
F	F → id	F → (E)				

The input Symbol: id + id $

Parse State	input	Production rule	derivation
$E	id + id $	E →TQ	E
$QT	id + id $	T →FR	E → TQ
$QRF	id + id $	F →id	E → FRQ
$QRid	id + id $		

Example: LL(1) Parser

	id	(	*	+	)	$
E	E →TQ	E →TQ				
Q				Q →+TQ	Q → ε	Q → ε
T	T →FR	T →FR				
R			R →*FR	R → ε	R → ε	R → ε
F	F → id	F → (E)				

The input Symbol: id + id $

Parse State	input	Production rule	derivation
$QR	+ id $	R → ε	E → id RQ
$Q	+ id $	Q →+TQ	E → id Q
$QT+	+ id $		E → id+TQ

Example: LL(1) Parser

	id	(	*	+	)	$
E	E →TQ	E →TQ				
Q				Q →+TQ	Q → ε	Q → ε
T	T →FR	T →FR				
R			R →*FR	R → ε	R → ε	R → ε
F	F → id	F → (E)				

The input Symbol: id + id $

Parse State	input	Production rule	derivation
$QT	id $	T →FR	
$QRF	id $	F →id	E → id+FRQ
$QRid	id $		E → id+idRQ

Example: LL(1) Parser

	id	(	*	+	)	$
E	E →TQ	E →TQ				
Q				Q →+TQ	Q → ε	Q → ε
T	T →FR	T →FR				
R			R →*FR	R → ε	R → ε	R → ε
F	F → id	F → (E)				

The input Symbol: id + id $

Parse State	input	Production rule	derivation
$QR	$	R → ε	
$Q	$	Q → ε	E → id+idQ
$	$		E → id+id

Empty-stack Yes- Accept!

QUIZ: LL(1) Parser

	id	(	*	+	)	$
E	E →TQ	E →TQ				
Q				Q →+TQ	Q → ε	Q → ε
T	T →FR	T →FR				
R			R →*FR	R → ε	R → ε	R → ε
F	F → id	F → (E)				

The input Symbol: (id + id $

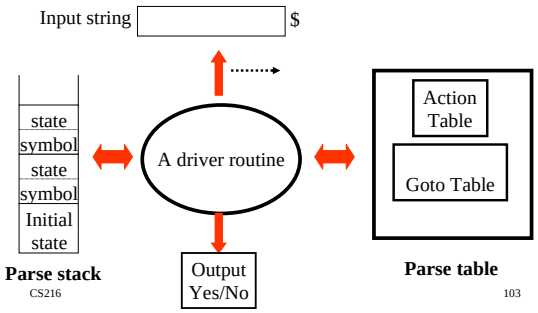
Parse State	input	Production rule	derivation

Bottom-Up Parsing

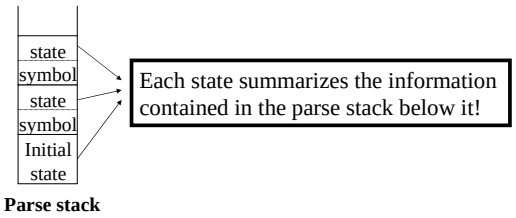
Bottom-Up Parsing

- Given an input string s and a CFG G=(T,N,S,P),
  - Scan the input string from left to right.
  - Look at one symbol at a time.
- Discover the parse tree T for the input string.
  - From leaves to the root (backward)
  - By following the right-most derivation (in reverse order).

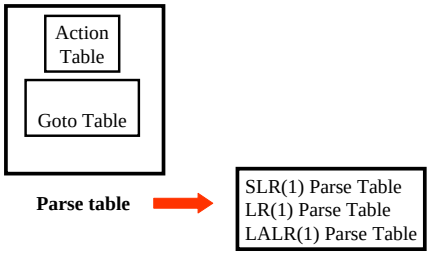
Bottom-Up Parsing



Parse Stack



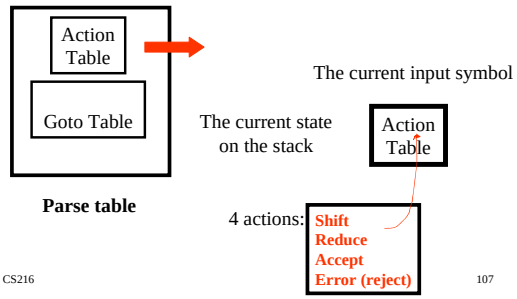
Parse Table



SLR(1), LR(1) and LALR(1) Parsing

- Use SLR(1) parse table
  - SLR(1) Parsing
- Use LR(1) parse table
  - LR(1) Parsing
- Use LALR(1) parse table
  - LALR(1) Parsing

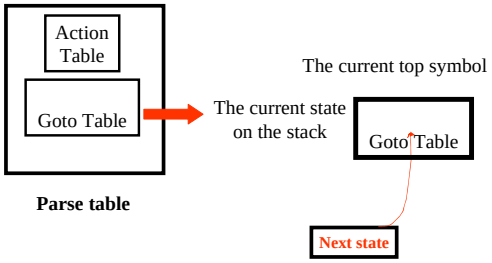
Action Table



Actions

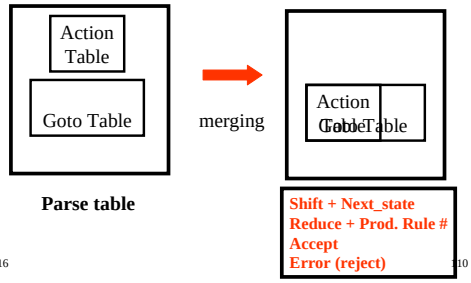
- Shift
  - Push the current input symbol on the stack
- Reduce with $A \rightarrow \beta$
  - Reduce by the production $A \rightarrow \beta$
  - Pop β and push A
- Accept
- Error (Reject)

Goto Table



Push the new state on the stack

Parse Table



Drive Routine for Bottom-Up Parsing

- Push the starting state (say 0) onto the parse stack.
  - The current state at the top of the stack is 0.
- Set the current input symbol to point to the first symbol of the input string.

Drive Routine for Bottom-Up Parsing

- Repeat
  - q = the current state (the top of the stack) ;
  - a = the current input symbol ;
  - Do ●●●●●●●
- Until (accept or reject) ;

Drive Routine for Bottom-Up Parsing

- If Parse_Table[q,a] = shift q' then
  - Push a onto the stack ;
  - Push q' onto the stack ;
  - Advance the current input symbol to the next;

Drive Routine for Bottom-Up Parsing

- If Parse_Table[q,a] = reduce #n then
  - Production rule #n: n : $A \rightarrow \beta$
  - Pop 2* $|\beta|$ symbols off the stack;
  - **if $\beta = \varepsilon$, then nothing is popped !**
  - Let q' be the state now on top of the stack;
  - Push A;
  - Let q'' be Parse_Table[q', A];
  - Push q'';

Drive Routine for Bottom-Up Parsing

- If Parse_Table[q,a] = accept then
  - Accept – Yes!
- Else
  - Reject – No!

Example: Bottom-Up Parsing

(0) E' → E
(1) E → E +T
(2) E → E –T
(3) E → T
(4) T→ T * F
(5) T→ T / F
(6) T→ F
(7) F → (E)
(8) F → id

Example: Action Part

	i	+	-	*	/	(	)	$
0	s					s		
1		s	s					acc
2		r3	r3	s	s		r3	r3
3		r6	r6	r6	r6		r6	r6
4	s					s		
5		r8	r8	r8	r8		r8	r8
6	s					s		
7	s					s		
8	s					s		
9	s					s		
10		s	s				s	
11		r1	r1	s	s		r1	r1
12		r2	r2	s	s		r2	r2
13		r4	r4	r4	r4		r4	r4
14		r5	r5	r5	r5		r5	r5
15		r7	r7	r7	r7		r7	r7

Example: Goto Part

	i	+	-	*	/	(	)	$	E	T	F
0	5					4			1	2	3
1		6	7								
2				8	9						
3											
4	5					4			10	2	3
5											
6	5					4				11	3
7	5					4				12	3
8	5					4					13
9	5					4					14
10		6	7				15				
11				8	9						
12				8	9						
13											
14											
15											

Example: Parse Table

	i	+	-	*	/	(	)	$	E	T	F
0	s5					s 4			1	2	3
1		s6	s7					acc			
2		r3	r3	s8	s 9		r3	r3			
3		r6	r6	r6	r6		r6	r6			
4	s5					s4			10	2	3
5		r8	r8	r8	r8		r8	r8			
6	s5					s 4				11	3
7	s5					s 4				12	3
8	s5					s 4					13
9	s5					s 4					14
10		s 6	s 7				s15				
11		r1	r1	s8	s 9		r1	r1			
12		r2	r2	s8	s 9		r2	r2			
13		r4	r4	r4	r4		r4	r4			
14		r5	r5	r5	r5		r5	r5			
15		r7	r7	r7	r7		r7	r7			

Example: Bottom-Up Parsing

Stack	Input	Entry	Action
0	**(i + i) / i $**	**s4**	**shift and enter state 4**
0(4	**(i + i) / i $**	**s5**	**shift and enter state 5**
0(4i5	**(i + i) / i $**	**r8**	**F → i**
0(4F3	**(i + i) / i $**	**r6**	**T → F**
0(4T2	**(i + i) / i $**	**r3**	**E → T**
0(4E10	**(i + i) / i $**	**s6**	**shift and enter state 6**
0(4E10+6	**(i + i) / i $**	**s5**	**shift and enter state 5**
0(4E10+6i5	**(i + i) / i $**	**r8**	**F → i**
0(4E10+6F3	**(i + i) / i $**	**r6**	**T → F**

Example: Bottom-Up Parsing

Stack	Input	Entry	Action
0(4E10+6F3	(i + i) / i $	r6	T → F
0(4E10+6T11	(i + i) / i $	r1	E → E+T
0(4E10	(i + i) / i $	s15	shift and enter state 15
0(4E10)15	(i + i) / i $	r7	F → (E)
0F3	(i + i) / i $	r6	T → F
0T2	(i + i) / i $	s9	shift and enter state 9
0T2/9	(i + i) / i $	s5	shift and enter state 5
0T2/9i5	(i + i) / i $	r8	F → i
0T2/9F14	(i + i) / i $	r4	T → T/F

Example: Bottom-Up Parsing

Stack	Input	Entry	Action
0T2/9F14	(i + i) / i $	r4	T → T/F
0T2	(i + i) / i $	r3	E → T
0E1	(i + i) / i $	acc	Accept!

Input string:
(i + i) / i $
Accept!

QUIZ: Bottom-Up Parsing

Input string:
i * (i – i $

Reject!

QUIZ: Bottom-Up Parsing

Stack	Input	Entry	Action
0	i * (i – i $	s5	shift and enter state 4
0i5	i * (i – i $	r8	F → i
0F3	i * (i – i $	r6	T → F
0T2	i * (i – i $	s8	
0T2*8	i * (i – i $	s4	
0T2*8(4	i * (i – i $	s5	
0T2*8(4i5	i * (i – i $	r8	F → i
0T2*8(4F3	i * (i – i $	r6	T → F
0T2*8(4T2	i * (i – i $	r3	E → T
0T2*8(4E10	i * (i – i $	s7	

QUIZ: Bottom-Up Parsing

Stack	Input	Entry	Action
0T2*8(4E10	i * (i – i $	s7	
0T2*8(4E10-7i5	i * (i – i $	s5	
0T2*8(4E10-7i5	i * (i – i $	r8	F → i
0T2*8(4E10-7F3	i * (i – i $	r6	T → F
0T2*8(4E10-7T12	i * (i – i $	r2	E → E-T
0T2*8(4E10	i * (i – i $	error	Reject!

Input string:
i * (i – i $
Reject!