

# Recursive Descent Parsers

## Lecture 5

ROKAN UDDIN FARUQUI

Associate Professor  
Dept of Computer Science and Engineering  
University of Chittagong, Bangladesh  
Email: *rokan@cu.ac.bd*

- 1 Parsing
- 2 LL Parsers and LR Parsers
- 3 Recursive Descent Parser
- 4 Example



# Outline

- 1 Parsing
- 2 LL Parsers and LR Parsers
- 3 Recursive Descent Parser
- 4 Example



# Parsing

## Definition (Parser)

A **parser** is a program that matches a sequence of tokens to the grammar rules of a context-free grammar, thereby building the **syntax tree** representing those tokens.



# Parsing Algorithms

**(id + id) \* id**

**$E \Rightarrow E * E \Rightarrow \dots \Rightarrow (id + id) * id$**

## Definition

A **top-down parser** begins with the start symbol and applies productions *to be matched*, until all the tokens have been processed.

## Definition

A **bottom-up parser** begins by matching productions to tokens *as they are read* and continues until the sequence of all tokens has been reduced to the start symbol.

**id + id**



# Parsing Algorithms

- Top-down parsers traverse the parse tree from the top down.
- They tend to be simpler but less powerful.
- Bottom-up parsers traverse the parse tree from the bottom up.
- They tend to be more complex, but more powerful.



# Parsing Algorithms

## Bison

- There are two basic methods of implementing top-down parsers.
  - Recursive descent parsers
  - Table-driven parsers, also called LL parsers
- Bottom-up parsers, also called LR parsers, are table-driven



# Outline

- 1 Parsing
- 2 LL Parsers and LR Parsers
- 3 Recursive Descent Parser
- 4 Example



# LL Parsers

$(id + id) * id$

Handwritten annotations: A red arrow points to the first 'id' in the expression. Another red arrow points to the '+' sign. The word 'leftmost' is written in red cursive to the right of the expression.

## Definition (LL Parser)

An **LL** **parser** parses the input from left to right (L) and uses a leftmost derivation (L).

## Definition (LR Parser)

An **LR** **parser** parses the input from left to right (L) and uses a rightmost derivation (R).



# Outline

- 1 Parsing
- 2 LL Parsers and LR Parsers
- 3 Recursive Descent Parser**
- 4 Example



# Recursive Descent Parser

**id + id**

**id - id**

**E  $\rightarrow$  E + E**

```

S() {
  A();
  B();
  C();
}

E() {
  E();
  match('+');
  E();
}

```

Definition (Recursive Descent Parser)

In a **recursive descent parser**,

- Each nonterminal in the grammar is implemented as a function.
- Parsing begins with the start symbol *S* by calling the function *S()*.
- Based on the first token received, an *S*-production is selected and executed. For example, if *S*  $\rightarrow$  A B C is selected, then the functions *A()*, *B()*, and *C()* are called, in that order.
- Continue in this manner until *S* is “satisfied.” That is, all tokens have been matched with the bodies of the productions.



# Recursive Descent Parsers

- The first Pascal compiler used a recursive descent parser.
- Recursive descent parsers have the benefit of being very simple to implement.
- However,
  - Error-recovery is difficult.
  - They are not able to handle as large a set of grammars as other parsing methods.



# Error Recovery

- When a syntax error occurs, in order for the compiler to recover, it usually has to discard the last few tokens, move to the end of the line, and resume.
- In a recursive descent parser, discarding tokens involves returning from several nested function calls.
- In a table-driven parser, discarding tokens requires simply clearing part of the stack.



# Outline

- 1 Parsing
- 2 LL Parsers and LR Parsers
- 3 Recursive Descent Parser
- 4 **Example**



# Example

## Example (Recursive Descent)

- Write a parser for the following grammar.

```

CO{
    case 1:
    'id' == 'num'

```

```

S → if C then S
      | while C do S
      | id = num ;
      | id ++ ;

```

```

C → id == num | id != num

```

```

SO{
    case 1:
    'if' CO 'then' SO;
    case 2:
    'while' CO 'do' SO;
    case 3:
    'id' '=' 'num' ';'
    case 4:
    'id' '+' '+' ';'
    default:
    syntax error;
}

```

where  $S$  represents a statement and  $C$  represents a condition.

# Example

## Example (Recursive Descent)

- Modify the previous example by adding the productions

$$S \rightarrow \text{do } S \text{ while } C ;$$
$$C \rightarrow \text{id} < \text{num}$$


# Example

## Example (Recursive Descent)

- Modify the previous example by adding the production

$$S' \rightarrow S S' \mid \epsilon$$

where  $S'$  represents a sequence of statements.

