

Lecture 7

Syntax Analysis III

Top Down Parsing

Top Down Parsing

- A top-down parser starts with the root of the parse tree, labeled with the start or goal symbol of the grammar.
- To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string
 1. At a node labeled A , select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
 2. When a terminal is added to the fringe that doesn't match the input string, backtrack (Some grammars are backtrack free (predictive))
 3. Find the next node to be expanded
- The key is selecting the right production in step 1
 - should be guided by input string

Recursive Descent Parsing

- Parse tree is constructed
 - From the top level non-terminal
 - Try productions in order from left to right
- Terminals are seen in order of appearance in the token stream.
- When productions fail, backtrack to try other alternatives
- Example:
 - Consider the parse of the string: $(int5)$
 - The grammar is :
$$E \rightarrow T \mid T + E$$
$$T \rightarrow int \mid int * T \mid (E)$$

Recursive Descent Parsing Algorithm

- TOKEN – type of tokens
 - In our case, let the tokens be: INT, OPEN, CLOSE, PLUS, TIMES
 - *next – points to the next input token
- Define boolean functions that check for a match of:
 - A given token terminal
`bool term(TOKEN tok) { return *next++ == tok; }`
- The nth production of a particular non-terminal S:
`bool Sn() { ... }`
- Try all productions of S:
`bool S() { ... }`

Recursive Descent Parsing Algorithm

- For production $E \rightarrow T$ **Functions for non-terminal 'E'**
[E T | T + E]
 `bool E1() { return T(); }`
- For production $E \rightarrow T + E$
 `bool E2() { return T() && term(PLUS) && E(); }`
- For all productions of E (with backtracking)
 `bool E() {
 TOKEN *save = next;
 return (next = save, E1())
 || (next = save, E2());
 }`

Recursive Descent Parsing Algorithm

- **Functions for non-terminal $T : [T \text{ int} \mid \text{int} * T \mid (E)]$**
 - `bool T1() { return term(INT); }`
 - `bool T2() { return term(INT) && term(TIMES) && T(); }`
 - `bool T3() { return term(OPEN) && E() && term(CLOSE); }`
- `bool T() {`
 - `TOKEN *save = next;`
 - `return (next = save, T1())`
 - `|| (next = save, T2())`
 - `|| (next = save, T3());`
 - `}`

Recursive Descent Parsing Algorithm

- To start the parser
 - Initialize *next* to point to first token
 - Invoke *E()*
 - Try parsing by hand:
 - (int)
- ```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }
bool E() {
 TOKEN *save = next;
 return (next = save, E1()) || (next = save, E2());
}

bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() {
 TOKEN *save = next;
 return (next = save, T1())
 || (next = save, T2())
 || (next = save, T3());
}
```

# Limitations of RD Parser

## Grammar

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

## Input String:

$int * int$

```
bool term(TOKEN tok) { return *next++ == tok; }
```

```
bool E1() { return T(); }
```

```
bool E2() { return T() && term(PLUS) && E(); }
```

```
bool E() {
```

```
 TOKEN *save = next;
```

```
 return (next = save, E1()) || (next = save, E2());
```

```
}
```

```
bool T1() { return term(INT); }
```

```
bool T2() { return term(INT) && term(TIMES) && T(); }
```

```
bool T3() { return term(OPEN) && E() && term(CLOSE); }
```

```
bool T() {
```

```
 TOKEN *save = next;
```

```
 return (next = save, T1())
```

```
 || (next = save, T2())
```

```
 || (next = save, T3());
```

```
}
```



# Limitations

- If a production for non-terminal  $X$  succeeds
  - Can't backtrack to try a different production for  $X$  later
- General recursive-descent algorithms supports such “full” backtracking
  - Can implement any grammar

# Countermeasures

- Discussed RD algorithm is not general
  - But easy to implement by hand
- Sufficient for the grammars where for any non-terminal at most one production can succeed.
- The example grammar can be rewritten to work with the presented algorithm
  - Left factoring

# Left Recursive Grammar

- Grammar:  $S \rightarrow Sa$ 
  - `bool S1() {return S() && terminal (a);}`
  - `bool S() { return S1();}`
- S( ) goes into an infinite loop
- A **left recursive grammar** has a non-terminal S such that

$$S \rightarrow S\alpha \quad \text{for some } \alpha$$

$$S \rightarrow Sa \rightarrow Saa \rightarrow Saaa \dots \dots \rightarrow Sa\dots \dots a$$

- Recursive Descent does not work in such cases.
- Consider the grammar:  $A \rightarrow A\alpha \mid \beta$
- A generates all string starting with a  $\beta$  and followed by any number of  $\alpha$ 's.

# Eliminating Left-Recursion

- Direct Left-Recursion:

$$A \rightarrow A\alpha \mid \beta$$



$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

A generates all the strings with a  $\beta$  and followed by any number of  $\alpha$ 's

$$A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \dots \mid \beta_n$$



$$A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$$

All strings derived from A start with one of  $\beta_1 \dots \beta_n$  and continue with several instances of  $\alpha_1 \dots \alpha_n$

# Eliminating Left-Recursion

- Indirect Left-Recursion

$$S \rightarrow A \alpha \mid \delta$$

$$A \rightarrow S \beta$$

The grammar is also left recursive  
because  $S \rightarrow S \beta \alpha$

- **Algorithm:**

1. *Arrange the non-terminals in some order  $A_1, \dots, A_n$*
2. *for ( $i$  in  $1..n$ ) {*
3. *for ( $j$  in  $1..i-1$ ) {*
4. *replace each production of the form  $A_i \rightarrow A_j \gamma$  by the*  
*productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$*
5. *}*
6. *eliminate the immediate left recursion among  $A_i$  productions*
7. *}*

The above algorithm guaranteed to work if the grammar has no cycle [derivation of the form  $A \rightarrow A$  or  $\epsilon$  production  $A \rightarrow \epsilon$ ]. Cycles can be eliminated systematically from a grammar as can  $\epsilon$  productions.

# Eliminating Left-Recursion

- $S \rightarrow Aa \mid b$
  - $A \rightarrow Ac \mid Sd \mid \epsilon$
- The grammar is also left recursive because  $S \rightarrow Sda$
- Out loop (2 to 7) eliminates any left recursion among  $A_1$  productions. Any remaining  $A_1$  productions of the form  $A_1 \rightarrow A_1\alpha$  must therefore have  $l > 1$ .
  - After  $i-1^{\text{st}}$  iteration of the outer for loop, all non terminal  $A_k$ ,  $k < i$ , is cleaned i.e. any production  $A_k \rightarrow A_l\alpha$ , must have  $l > k$
  - At the  $i^{\text{th}}$  iteration, inner loop 3to5, progressively raises the lower limit in any productions  $A_i \rightarrow A_m\alpha$ , until we have  $m > i$ .
  - Line 6, eliminating left recursion for  $A_i$  forces  $m$  to be greater than  $i$

**Thanks**