COEN 175

Lecture 4: Top-Down Parsing of Expressions

Top-Down Parsing

- Top-down parsers construct parse trees from the root to the leaves.
- Top-down parsers are suited for LL(*k*) grammars.
  - LL = left-to-right scan, leftmost derivation
  - *k* = number of symbols of look-ahead
- Bottom-up parsers are suited for LR(*k*) grammars.
  - LR = left-to-right scan, rightmost derivation in reverse
- The easiest type of top-down parser to implement is a **recursive descent** parser.

Recursive Descent Parsing

- A recursive descent parser is implemented as a set of functions:
  - Each nonterminal has a corresponding function that is responsible for matching its right-hand side.
  - Each nonterminal on the right-hand side is replaced by a call to its corresponding function.
  - Each terminal on the right-hand side is matched with (i.e., compared against) the current token.
- A recursive descent parser is a **predictive parser** if it never has to backtrack. Thus, we can parse any program in linear time.

A Simple Example

- Implement a parser for the following grammar:

  $S \rightarrow A\ B$
  $A \rightarrow a$
  $B \rightarrow b$

- Sample implementation:

```
void S() {
    A();
    B();
}
```

```
void A() {
    match('a');
}

void B() {
    match('b');
}
```

```
int lookahead = lexan();

void match(int t) {
    if (lookahead == t)
        lookahead = lexan();
    else
        error();
}
```

Selecting Alternatives

- Implement a parser for the following grammar:

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

- Sample implementation:

```
void A() {
    match('a');
}

void B() {
    match('b');
}
```

```
void S() {
    if (lookahead == 'a') {
        match('a');
        A();
    } else {
        match('b');
        B();
    }
}
```

A Big Problem

- Consider the following grammar:

  $E \rightarrow E + T \mid T$

- In order to match *E*, we first may have to match *E*. This will result in infinite recursion.

- We must **eliminate left recursion** by rewriting the grammar to use only right recursion:

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

- In other words, an expression is a term followed by zero or more terms summed together.

Eliminating Left Recursion

- An LL(k) grammar cannot have left recursion.
- Consider the following grammar, where no α_i begins with A:

  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$

- The grammar without left recursion is:

  $A \rightarrow \beta\, A'$

  $A' \rightarrow \alpha_1\, A' \mid \alpha_2\, A' \mid \ldots \mid \alpha_n\, A' \mid \epsilon$

- When coding our parser, we can either explicitly eliminate the left recursion or implicitly do so by combining the functions for A and A'.

A First Implementation

- Implement a parser for the following grammar:

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

- A straightforward implementation:

```
void E() {
    T();
    E'();
}
```

```
void E'() {
    if (lookahead == '+') {
        match('+');
        T();
        E'();
    } else {
        /* nothing to do */
    }
}
```

Eliminating Tail Recursion

- Implement a parser for the following grammar:

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

- A better implementation:

```
void E() {
    T();
    E'();
}
```

```
void E'() {
    while (lookahead == '+') {
        match('+');
        T();
    }
}
```

With Function Inlining

- Implement a parser for the following grammar:

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

- An even better implementation:

```
void E() {
    T();

    while (lookahead == '+') {
        match('+');
        T();
    }
}
```

An Advanced Example

- Implement a parser for the following grammar:

  $E \rightarrow E + T \mid E - T \mid T$
  $T \rightarrow T * F \mid T / F \mid F$
  $F \rightarrow (E) \mid \textbf{id}$

- First, we eliminate left recursion:

  $E \rightarrow T\ E'$
  $E' \rightarrow + T\ E' \mid - T\ E' \mid \epsilon$
  $T \rightarrow F\ T'$
  $T' \rightarrow * F\ T' \mid / F\ T' \mid \epsilon$
  $F \rightarrow (E) \mid \textbf{id}$

- We now implement functions for the rules following the method we just established.

An Advanced Example

- Looking at the first two rules:

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E' \mid -\ T\ E' \mid \epsilon$$

- Our implementation:

```
void E() {
    T();

    while (lookahead == '+' || lookahead == '-') {
        match(lookahead);
        T();
    }
}
```

An Advanced Example

- Looking at the next two rules:

  $T \rightarrow F\ T'$

  $T' \rightarrow *\ F\ T' \mid /\ F\ T' \mid \epsilon$

- Our implementation:

```
void T() {
    F();

    while (lookahead == '*' || lookahead == '/') {
        match(lookahead);
        F();
    }
}
```

An Advanced Example

- Looking at the final rule:

  $F \rightarrow$ (E) | **id**

- Our implementation:

```
void F() {
    if (lookahead == '(') {
        match('(');
        E();
        match(')');
    } else
        match(ID);
}
```