CMSC 330: Organization of Programming Languages

Recursive Descent Parser Example

Recursive Descent Parser

- For terminals, create function match(a)
  - If lookahead is a it consumes the lookahead by advancing the lookahead to the next token, and returns
- For each nonterminal N, create a function parse_N
  - Let $N \rightarrow \beta_1 \mid \ldots \mid \beta_k$ be the productions of N
  - For each production $N \rightarrow \beta_i$ (where $\beta_i = \alpha_1 \ldots \alpha_n$)
  - If the lookahead is in First(β_i)
    - Call parse_α_1( ); ... ; parse_α_n( )

First Sets

- Definition
  - First(γ), for any terminal or nonterminal γ, is the set of initial terminals of all strings that γ may expand to
- For a terminal a
  - First(a) = { a }
- For a nonterminal N
  - If $N \rightarrow \varepsilon$, then add ε to First(N)
  - If $N \rightarrow \alpha_1\ \alpha_2 \ldots \alpha_n$, then add
    - First(α_1) if $\varepsilon \notin$ First(α_1)
    - Else add { First(α_1) – ε} ∪ First($\alpha_2 \ldots \alpha_n$)

Left Recursion Elimination Algorithm

- Given grammar
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$
- Rewrite grammar as
  - $A \rightarrow \beta L$
  - $L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \varepsilon$
- Repeat as necessary

Left Factoring Algorithm

- Given grammar
  - $A \rightarrow x\alpha_1 \mid x\alpha_2 \mid \ldots \mid x\alpha_n \mid \beta$
- Rewrite grammar as
  - $A \rightarrow xL \mid \beta$
  - $L \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
- Repeat as necessary

Example 1 – Left Factoring

Consider the grammar

$S \rightarrow T + S \mid T$
$T \rightarrow U * T \mid U$
$U \rightarrow (S) \mid V$
$V \rightarrow 0 \mid 1 \mid \ldots \mid 9$

Common prefix requires applying left factoring

$S \rightarrow T + S \mid T$
$T \rightarrow U * T \mid U$

Left factor S

$S \rightarrow T + S \mid T$
↓
$S \rightarrow T\ L$
$L \rightarrow + S \mid \varepsilon$

Left factor T

$T \rightarrow U * T \mid U$
↓
$T \rightarrow U\ M$
$M \rightarrow * T \mid \varepsilon$

Example 1 – Computing First Sets

Grammar
- $S \to T\ L$
- $L \to +\ S \mid \varepsilon$
- $T \to U\ M$
- $M \to *\ T \mid \varepsilon$
- $U \to (S) \mid V$
- $V \to 0 \mid 1 \mid \ldots \mid 9$

Compute First Sets
- First(V) = { 0…9 }
- First(U) = First((S)) ∪ First(V)
  - = { (} ∪ { (, 0…9 } = { (, 0…9 }
- First(M) = First(*T) ∪ First(ε)
  - = { * } ∪ { ε } = { *, ε }
- First(T) = First(UM) = First(U)
  - = { (, 0…9 }
- First(L) = First(+S) ∪ First(ε)
  - = { + } ∪ { ε } = { +, ε }
- First(S) = First(TL) = First(T)
  - = { (, 0…9 }

Example 1 – Recursive Descent Parser

Grammar
- $S \to T\ L$
- $L \to +\ S \mid \varepsilon$
- $T \to U\ M$
- $M \to *\ T \mid \varepsilon$
- $U \to (S) \mid V$
- $V \to 0 \mid 1 \mid \ldots \mid 9$

Recursive descent parser

```
parse_S( ) {
    // S → TL
    parse_T(); parse_L();
}

parse_L( ) {
    // L → + S
    if (lookahead == "+") {
        match("+"); parse_S( );
    }
    // L → ε
    else ;
}

parse_T( ) {
    // T → UM
    parse_U(); parse_M();
}
```


Example 1 – Recursive Descent Parser

Grammar
- $S \to T\ L$
- $L \to +\ S \mid \varepsilon$
- $T \to U\ M$
- $M \to *\ T \mid \varepsilon$
- $U \to (S) \mid V$
- $V \to 0 \mid 1 \mid \ldots \mid 9$

Recursive descent parser

```
parse_M( ) {
    // M → * T
    if (lookahead == "*") {
        match("*"); parse_T( );
    }
    else ;    // M → ε
}

parse_U( ) {
    if (lookahead == "(") {           // U → (S)
        match("("); parse_S( ); match(")");
    }
    else parse_V( );                  // U → V
}

parse_V( ) {
    if (lookahead == "0") {           // V → 0
        match("0");
    } else if (lookahead == "1") {    // V → 1
        match("1");
    } else
        …
    } else if (lookahead == "9") {    // V → 9
        match("9");
    } else error( );
}
```


Example 1 – Parsing Input

Grammar
- $S \to T\ L$
- $L \to +\ S \mid \varepsilon$
- $T \to U\ M$
- $M \to *\ T \mid \varepsilon$
- $U \to (S) \mid V$
- $V \to 0 \mid 1 \mid \ldots \mid 9$

Lookahead in red

Parse "1+2"	Remaining
parse_S( )	"1+2"
parse_T( )	"1+2"
parse_U( )	"1+2"
parse_V( )	"1+2"
match("1")	"1+2"
parse_M( )	"+2"
parse_L( )	"+2"
match("+")	"+2"
parse_S( )	"2"
parse_T( )	"2"
parse_U( )	"2"
parse_V( )	"2"
match("2")	"2"
	""

Example 2 – Computing First Sets

Consider the grammar
- $S \to ABd \mid aBc$
- $A \to \varepsilon$
- $B \to b \mid c$

Compute First Sets
- First(b) = {b}
- First(c) = {c}
- First(B) = First(b) ∪ First(c)
  - = {b,c}

- First(ε) = {ε}
- First(A) = First(ε) = {ε}

- First(ABd) = {First(A) – ε} ∪ First(Bd)
  - = {ε– ε} ∪ First(Bd)
  - = First(Bd) = First(B) = {b, c}
- First(aBc) = { a }

- First(S) = First(ABd) ∪ First(aBc)
  - = {b, c} ∪ {a}
  - = {a,b,c}

Example 2 – Using First Sets

Grammar
- $S \to ABd \mid aBc$
- $A \to \varepsilon$
- $B \to b \mid c$

First sets for RHS
- First(b) = {b}
- First(c) = {c}

- First(ABd) = {b, c}
- First(aBc) = { a }

Grammar is predictive if…
- First(ABd) ∩ First(aBc) = ∅
- First(b) ∩ First(c) = ∅
- And not left recursive…

Verify…
- First(ABd) ∩ First(aBc)
  - = {b, c} ∩{ a }
  - = ∅
- First(b) ∩ First(c)
  - = {b} ∩{c}
  - = ∅

No overlap, grammar is predictive

Example 2 – Recursive Descent Parser

Grammar

S → ABd | aBc
A → ε
B → b | c

Recursive descent parser

```
parse_B( ) {
    if (lookahead == “b”)
        match(“b”);   // B → b
    else if (lookahead == “c”)
        match(“c”);   // B → c
    else error( );
}
```

```
parse_A( ) { return; }   // A → ε

parse_S( ) {
    // S → ABd
    if ((lookahead == “b”) ||
      (lookahead == “c”)) {
      parse_A( ); parse_B( ); match(“d”);
    }
    // S → aBc
    else if (lookahead == “a”) {
      match(“a”); parse_B( ); match(“c”);
    }
    else error( );
}
```


Example 2 – Parsing Input

Grammar

S → ABd | aBc
A → ε
B → b | c

Lookahead in red

Parse “bd”	Remaining
parse_S( )	“bd”
parse_A( )	“bd”
parse_B( )	“bd”
match(“b”)	“bd”
match(“d”)	“d”
	“”

Parse “acc”	Remaining
parse_S( )	“acc”
match(“a”)	“acc”
parse_B( )	“cc”
match(“c”)	“cc”
match(“c”)	“c”
	“”