CS 421 Lecture 8: Top-down parsing

- Lecture outline
  - Recursive-descent formalized
    - FIRST sets
    - LL(1) condition
    - Transformations to LL(1) form
  - Grammars for expressions

Review: context-free grammar

- Given:
  - Set of terminals (tokens) T
  - Set of non-terminals (variables) V
- A cfg G is a set of *productions* of the form
  - $A \rightarrow X_1 \ldots X_n \quad (n \geq 0)$

  where
  - $A \in V, X_1 \ldots X_n \in G = V \cup T$
  - One symbol designated as "start symbol"

Top-down parsing: outline

- Top-down parsing
  - Start parsing with start symbol
  - Apply production rules one by one

- More than one production for rule *A*
  - Look at the next token to decide which production to apply

Top-down parsing: pseudocode

- For each non-terminal with productions
  - $A \rightarrow X_1 \ldots X_n \mid Y_1 \ldots Y_n \mid \ldots \mid Z_1 \ldots Z_n$
- Define `parseA`:

```
parseA toklis = choose production based on hd toklis:
      if A → X1 … Xn: handle X1 … Xn
      else if A → Y1 … Yn: handle Y1 … Yn
      else if …

handle X1 … Xn: handle X1; handle X2; … ; handle Xn
      where handle t : if hd toklis = t
                       then remove t and continue
                       else error

handle B : parseB toklis
```

“choose production based on hd toklis”

- Need to formalize some things…
- Define $\Rightarrow$
  - “Derives in one step”
  - $X_1 \ldots X_n \Rightarrow w_1 \ldots w_n$, where $X_i \in G$ and $w_i \in G^*$ if there exists j such that $X_j \rightarrow w_j$ is a production in G, and for all $i \neq j$, $X_i = w_i$
- $\Rightarrow^+$ and $\Rightarrow^*$ are the transitive and reflexive-transitive closures of $\Rightarrow$.
  - Say $X_1 \ldots X_n$ *derives* α *if* $X_1 \ldots X_n \Rightarrow^* \alpha$.
  - *E.g.*, α is a sequence of G if the start symbol of G derives α and α consists solely of tokens.

More Definitions

- $X_1 \ldots X_n$ is *nullable* if it can derive ε.
- FIRST($X_1 \ldots X_n$) = $\{ t \in T \mid X_1 \ldots X_n \Rightarrow^* t\alpha$ for some $\alpha\}$

  $\cup \{ \_ \mid X_1 \ldots X_n$ *nullable* $\}$
- G is *left-recursive* if there exists A: $A \Rightarrow^+ A\alpha$ for some α.
- G is *LL(1)* if
  - G is not left-recursive, and
  - $\forall A$, if the productions of A are: $A \rightarrow X \mid Y \mid \ldots \mid Z$ then the sets

    FIRST(X), …, FIRST(Z) are pairwise disjoint.

Top-down parsing: revisited

- If G is LL(1), then for each non-terminal *A* with productions
  - $A \rightarrow X_1 \ldots X_n \mid Y_1 \ldots Y_n \mid \ldots \mid Z_1 \ldots Z_n$

- Define `parseA`:

```
parseA toklis = let t = hd toklis in
      if t ∈ FIRST(X1 … Xn) then handle X1 … Xn
      else if t ∈ FIRST(Y1 … Yn) then handle Y1 … Yn
      else if …
      else if t ∈ FIRST(Z1 … Zn) or _ ∈ FIRST(Z1 … Zn)
           handle Z1 … Zn
           else error
handle X1 … Xn: handle X1; handle X2; … ; handle Xn

handle t : if hd toklis = t
              then remove t and continue
              else error

handle B : parseB toklis
```

Transformation to LL(1)

- Left refactoring:
  - $A \rightarrow \alpha\beta \;|\; \alpha\gamma$
  - $\Rightarrow$
    - $A \rightarrow \alpha \; B$
    - $B \rightarrow \beta \;|\; \gamma$

- Left-recursion removal:
  - $A \rightarrow A\alpha \;|\; \beta$
  - $\Rightarrow$
    - $A \rightarrow \beta \; B$
    - $B \rightarrow \varepsilon \;|\; \alpha \; B$

Example

- Consider non-LL(1) grammar 3 from previous class:
  - A → id | '(' B ')'
  - B → A | A '+' B

- Grammar 3 transformed to LL(1) form:
  - A → id | '(' B ')'
  - B → A C
  - C → '+' A C | ε

Ambiguity

- More than one valid parse tree for one input
- No test for ambiguity
- Recursive descent and LR(1) parsing not applicable to ambiguous grammar
  - Possible to “cheat” with LR parser – will see how next week

Expression grammars

- Expressions are challenging for several reasons
  - Should be LL(1) and LR(1)
  - Grammar should enforce precedence, if possible
  - Grammar should enforce associativity, if possible
  - Grammar shouldn’t be ambiguous
  - Should be easy to construct *abstract* syntax tree
- Especially hard to write LL(1) parser for expressions
  - Not so hard for LR(1)

Enforcing precedence

- Consider:
  - $x + y * z$ $\qquad\qquad$ $x * y + z$
- How should we parse?

Enforcing associativity

- Consider:
  - x - y - z   x = y = z + 1

- How should we parse?

Example: expression grammars

- Some expression grammars:
  - G_A: $E \rightarrow id \mid E - E \mid E * E$
  - G_B: $E \rightarrow id \mid id - E \mid id * E$
  - G_C: $E \rightarrow id \mid E - id \mid E * id$

Example: G_A

- G_A: $E \rightarrow id \mid E - E \mid E * E$
  - Ambiguity?
  - LR(1)/LL(1)?
  - Precedence?
  - Associativity?

  - $x - y * z$

Example: G_B

- G_B: $E \rightarrow id \mid id - E \mid id * E$
  - Ambiguity?
  - LR(1)/LL(1)?
  - Precedence?
  - Associativity?

  - $x - y * z$

  - $x * y - z$

  - $x - y - z$

Example: G_C

- G_C: $E \rightarrow id \mid E - id \mid E * id$
  - Ambiguity?
  - LR(1)/LL(1)?
  - Precedence?
  - Associativity?

  - $x - y * z$

  - $x * y - z$

  - $x - y - z$

Example: more expression grammars

- Some more expression grammars:
  - G_D: $E \rightarrow T - E \mid T$
    $T \rightarrow id \mid id * T$
  - G_E: $E \rightarrow E - T \mid T$
    $T \rightarrow id \mid T * id$
  - G_F: $E \rightarrow T\ E'$
    $E' \rightarrow \varepsilon \mid - E$
    $T \rightarrow id\ T'$
    $T' \rightarrow \varepsilon \mid * T$

Example: G_D

- G_D: $E \rightarrow T - E \mid T$
  $T \rightarrow id \mid id * T$
  - Ambiguity?
  - LR(1)/LL(1)?
  - Precedence?
  - Associativity?

  - $x - y * z$

  - $x * y - z$

  - $x - y - z$

Example: G_E

- G_E: $E \rightarrow E - T \mid T$
  $T \rightarrow id \mid T * id$
  - Ambiguity?
  - LR(1)/LL(1)?
  - Precedence?
  - Associativity?

  - $x - y * z$

  - $x * y - z$

  - $x - y - z$

Example: G_F

- G_F: $E \rightarrow T\ E'$

  $E' \rightarrow \varepsilon \mid -\ E$

  $T \rightarrow id\ T'$

  $T' \rightarrow \varepsilon \mid *\ T$

  - Ambiguity?
  - LR(1)/LL(1)?
  - Precedence?
  - Associativity?

  - $x - y * z$

  - $x * y - z$

  - $x - y - z$

Next class

- More parsing (yay!)
  - Bottom-up parsing
  - LR(1)