Lecture 8: Recursive-descent parsing

- **Recursive-descent formalized**
  - **FIRST sets**
  - **LL(1) condition**
  - **Transformations to LL(1) form**
- **Grammars for expressions - a difficult case**

(Next week: LR(1) parsing, ocamlyacc)

Top-down parsing

- **For each non-terminal with productions:**

$$A \rightarrow \vec{X} \mid \vec{Y} \mid \ldots \mid \vec{Z}$$

define parseA:

`parseA toklis` = choose production based on hd toklis:
 if $A \rightarrow \vec{X}$ chosen: handle $\vec{X}$
 else if $A \rightarrow \vec{Y}$ chosen: handle $\vec{Y}$,
 else if *etc.*

handle $X_1 X_2 \ldots X_n$: handle X_1; handle X_2; $\ldots$; handle X_n

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 “$\Rightarrow$”: $X_1 \ldots X_n \Rightarrow X_1 \ldots X_{i-1} \alpha X_{i+1} \ldots X_n$ (for any $1 \leq i \leq n$) if the grammar has production $X_i \rightarrow \alpha$.**
- **$\Rightarrow^+$ and $\Rightarrow^*$ are the transitive and reflexive-transive closures of $\Rightarrow$. (Say $\vec{X}$ *derives* α if $\vec{X} \Rightarrow^* \alpha$.)**
- **α is a *sentential form* of G if the start symbol of G derives α. If, furthermore, α consists solely of tokens, then it is a *sentence*. (These notions correspond to being the “frontier” of a syntax tree; some care is needed in defining “frontier” to account for ϵ-productions.)**

"choose production based on hd toklis" (cont.)

- $\vec{X}$ **is *nullable* if it can derive** ϵ**.**
- **Define: FIRST(**$\vec{X}$**) =** $\{t \in T \,|\, \vec{X} \Rightarrow^* t\alpha$ **for some** $\alpha\} \cup \{\bullet \,|\, \vec{X}\ nullable\}$**.**
- **Define: FOLLOW(**A**) =** $\{t \in T \,|\, \exists$ **a sentential form** $\alpha A t \beta\}$

There are well-known algorithms for calculating FIRST and FOLLOW sets, but we will consider only simple cases where they can be calculated by inspection.

"choose production based on hd toklis" (cont.)

- **Define: G is *left-recursive* if $\exists A:\ A \Rightarrow^{+} A\alpha$ for some α.**
- **Define: G is *LL(1)* if**

1. **G is not left-recursive, and**
2. **For all non-terminals A, if the productions of A are $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$:**
   (a) **The sets FIRST(α_1), ..., FIRST(α_n) are pairwise disjoint. ($\forall i, j. i \neq j \Rightarrow FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$.)**
   (b) **If A is nullable, then suppose α_i is the unique right-hand side such that $\bullet \in$ FIRST(α_i). Then, for all $j \neq i$, FIRST(α_j)$\cap$ FOLLOW(A) $= \emptyset$.**

Top-down parsing revisited

If G is LL(1), then for each non-terminal A with productions

$$A \rightarrow \vec{X} \mid \vec{Y} \mid \ldots \mid \vec{Z}$$

construct `parseA`:

`parseA toklis = let t = hd toklis in`
 `if t` $\in$ FIRST($\vec{X}$) `then` handle $\vec{X}$
 `else if t` $\in$ FIRST($\vec{Y}$) `then` handle $\vec{Y}$
 ...`else if t` $\in$ FIRST($\vec{Z}$) or ($\bullet \in$ FIRST($\vec{Z}$) and t $\in$ FOLLOW(A))
  `then` handle $\vec{Z}$ ($\vec{Z}$ the unique nullable right-hand side of A, if any)
  `else` *error*

handle $X_1, X_2, \ldots, X_n$: handle X_1; handle X_2; $\ldots$; handle X_n

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

handle B : `parseB toklis`

Transformation to LL(1)

- **Left refactoring:**

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

$$\Rightarrow \quad \begin{array}{l} A \rightarrow \alpha B \\ B \rightarrow \beta \mid \gamma \end{array}$$

- **Left-recursion removal:**

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

$$\Rightarrow \quad \begin{array}{l} A \rightarrow \beta B \\ B \rightarrow \epsilon \mid \alpha B \end{array}$$

Example

- **Consider non-LL(1) grammar 3 from the previous class:**

  A $\rightarrow$ **id** | **'(' B ')'**
  B $\rightarrow$ **A** | **A '+' B**

- **Grammar 3 transformed to LL(1) form:**

  A $\rightarrow$ **id** | **'(' B ')'**
  B $\rightarrow$ **A C**
  C $\rightarrow$ **'+' B** | ϵ

Ambiguity

- **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:**
  - **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

Enforcing associativity

Some expression grammars

G_A: E $\rightarrow$ id | E - E | E * E

G_B: E $\rightarrow$ id | id - E | id * E

G_C: E $\rightarrow$ id | E - id | E * id

G_D: E $\rightarrow$ T - E | T
T $\rightarrow$ id | id * T

G_E: E $\rightarrow$ E - T | T
T $\rightarrow$ id | T * id

G_F: E $\rightarrow$ T E$'$
E$'$ $\rightarrow$ ϵ | - E
T $\rightarrow$ id T$'$
T$'$ $\rightarrow$ ϵ | * T

- G_A: **E** $\rightarrow$ **id** $|$ **E - E** $|$ **E * E**

  - **x - y * z** **x - y - z**

- G_B: **E** $\rightarrow$ **id** | **id - E** | **id * E**

  - **x - y * z**   **x - y * z - w**

  - **x * y - z**

- G_C: **E** $\rightarrow$ **id** | **E - id** | **E * id**

  - **x - y * z** **x - y * z - w**

  - **x * y - z**

- G_D: **E** $\longrightarrow$ **T - E** $|$ **T**
  T $\longrightarrow$ **id** $|$ **id * T**

- **x - y * z** **x * y - z** **x - y - z**

- G_E: **E** $\longrightarrow$ **E - T** $|$ **T**
  T $\longrightarrow$ **id** $|$ **T * id**

- **x - y * z** **x * y - z** **x - y - z**

- G_F: $\mathbf{E} \longrightarrow \mathbf{T}\ \mathbf{E}'$

  $\mathbf{E}' \longrightarrow \epsilon \mid \textbf{-}\ \mathbf{E}$

  $\mathbf{T} \longrightarrow \textbf{id}\ \mathbf{T}'$

  $\mathbf{T}' \longrightarrow \epsilon \mid \textbf{*}\ \mathbf{T}$

- **x - y * z** **x * y - z** **x - y - z**