Announcements

- HW1 due on Monday February 8th
  - Name and date your submission
  - Submit electronically in Homework Server AND on paper in the beginning of class
  - You may submit late on Thursday February 11th for 50% credit
  - No other submissions accepted

- Email questions to csci4430@lists.cs.rpi.edu

Last Class

- Formal languages describe and recognize Programming language syntax
- Regular languages specify tokens (e.g., keywords, identifiers, numeric literals, etc.)
  - Generated by Regular expressions
  - Recognized by DFAs (in a compiler, the scanner)
- Context-free languages describe more complex constructs (e.g., expressions and statements)
  - Generated by Context-free grammars
  - Recognized by PDAs (in a compiler, called parser)
- We reviewed regular expressions, CFGs, derivation, parse, ambiguity. Scanning

Today’s Lecture Outline

- Top-down parsing vs. bottom-up parsing
- Top-down parsing
  - Introduction
  - A backtracking parser
  - Recursive descent predictive parser
  - Table-driven predictive parser
  - LL(1) parsing table
    - FIRST, FOLLOW and PREDICT sets
    - Constructing LL(1) parsing tables

Programming Language Syntax, Parsing

Read: Scott, Chapter 2.3.1 and 2.3.2

A Simple Calculator Language

```
asst_stmt -> id = expr ; // asst_stmt is the start symbol
expr -> expr + expr | expr * expr | id
```

Character stream: `position = initial + rate * time;`

Scanner

Parser

Parse tree:

```
 asst_stmt
  | id = expr
  | id + id * id ;
```

(Parse tree simplified to fit on slide.)
Parsing

- Objective: build a parse tree for an input string of tokens, from a single scan of input!
  - Only special subclasses of context-free grammars can do this

Two approaches
- Top-down: builds parse tree from the root to the leaves
- Bottom-up: builds parse tree from the leaves to the top
- Both are easily automated

Grammar for Comma-separated Lists

\[
\text{list} \rightarrow \text{id list_tail} \quad // \text{list is the start symbol}
\text{list_tail} \rightarrow , \text{id list_tail} | ;
\]

Generates comma-separated lists of id's.

E.g., id; id, id, id;

For example:

\[
\text{list} \Rightarrow \text{id list_tail} \\
\text{list_tail} \Rightarrow \text{id}, \text{id list_tail} \\
\Rightarrow \text{id}, \text{id ;}
\]

Top-down Parsing

- Terminals are seen in the order of appearance in the token stream
  - id, id, id ;
- The parse tree is constructed
  - From the top to the leaves
  - Corresponds to a left-most derivation
  - Look at left-most nonterminal in current sentential form, and lookahead terminal and “predict”, which production to apply

Bottom-up Parsing

- Terminals are seen in the order of appearance in the token stream
  - id, id, id ;
- The parse tree is constructed
  - From the leaves to the top
  - A right-most derivation in reverse

Top-down Predictive Parsing

- “Predicts” production to apply based on one or more lookahead token(s)
- Predictive parsers work with LL(k) grammars
  - First L stands for “left-to-right” scan of input
  - Second L stands for “left-most” derivation
  - Parse corresponds to left-most derivation
  - k stands for “need k tokens of lookahead to predict”
- We are interested in LL(1)

Question

- Can we always predict what production to apply based on one token of lookahead?
  - id, id, id ;
  - Yes, there is at most one choice (i.e., at most one production that applies)
  - This grammar is an LL(1) grammar
Question

- A new grammar
- What language does it generate?
  - Same, comma-separated lists
- Can we predict based on one token of lookahead?
  - id, id, id;

Aside: Top-down Depth-first Parsing

- For each nonterminal, exhaustively try productions in order, backtracking if necessary

- Consider the grammar. $S$ is the start symbol
  - $S \rightarrow cAd$
  - $A \rightarrow ab | a$
- Consider string
  - cad

Aside: Top-down Depth-first Parsing

- Grammar $S \rightarrow aSbS | bSaS | \epsilon$

- Input string abba

Aside: Backtracking, more generally

- A general search technique
  - Searches for a solution in (large) space
  - 1. We have partial solution
    - Make next choice and expand solution
      - If solution, then done
      - If (partial) solution invalid, backtrack to the last point $p$ where there is untried choice (undoing all work since that point). Repeat 1.
        - If there is no such $p$, then there is no solution
      - If partial solution valid, repeat 1.

Aside: Depth-first Parsing

- A string of tokens to parse: $t_1 t_2 \ldots t_n$
- Begin with start symbol sentential form

- Let $A$ be leftmost nonterminal in current sentential form. Exhaustively try each production for $A$ backtracking if necessary
- E.g., input cad

  - $S \Rightarrow cAd$ (try $A \rightarrow ab$, get $cabd$, no match, backtrack) $cAd \Rightarrow (try A \rightarrow a$, get $cad$, DONE!)

  - $S \rightarrow cAd$
  - $A \rightarrow ab | a$

Aside: Top-down Depth-first Parsing

- Input string cad
  - Start with start symbol
  - Try $S \rightarrow cAd$
    - Leaf $c$ matches input $c$
  - Try a rule for $A$: $A \rightarrow ab$
    - Leaf $a$ matches input $a$
    - But $b \neq d$. Backtrack to $A$
  - Try second rule for $A$: $A \rightarrow a$
    - Leaf $a$ matches $a$.
    - $d$ matches $d$.
    - Done: $S \Rightarrow cAd \Rightarrow cad$

  - $S \rightarrow cAd$
  - $A \rightarrow ab | a$
Aside: Depth-first Parsing

- Sentential form is $t_1 t_2 \ldots t_k A \ldots$
  - Initially $k = 0$ and $A \ldots$ is the start symbol
- Try a production for $A$ (leftmost nonterminal)
  - Say, $A \to t_{i1} t_{i2} B \ldots$ to get $t_1 t_2 \ldots t_{i1} t_{i2} B \ldots$
  - Backtrack if necessary
- Accept when there are no more nonterminals and all terminals match, or reject when there are no more productions left
- A problematic strategy…

Top-down Predictive Parsing

- Back to predictive parsing!
- “Predicts” production to apply based on one or more lookahead token(s)
  - No backtracking! Parser always gets it right
- Predictive parsers work with LL(k) grammars

Top-down Predictive Parsing

- Expression grammar:
  - Not LL(1)
- Unambiguous version:
  - Still not LL(1). Why?
- LL(1) version:

```
expr  →  expr + expr
expr  →  expr * expr
expr  →  id

expr  →  term term_tail
term_tail →  + term term_tail | ε
term  →  id factor_tail
factor_tail →  * id factor_tail | ε
term  →  id factor_tail
factor_tail →  * id factor_tail | ε
```

Exercise

- Draw the parse tree for expression $id + id * id + id$

Lecture Outline

- Top-down parsing vs. bottom-up parsing
- Top-down parsing
  - Introduction
  - A backtracking parser
  - Recursive descent predictive parsing
  - Table-driven top-down predictive parsing
  - LL(1) parsing table
    - FIRST, FOLLOW and PREDICT sets
    - Constructing LL(1) parsing tables

Recursive Descent

- Each nonterminal has a procedure
- The right-hand-sides (rhs) for the nonterminal form the body of its procedure

- `lookahead()`
  - Peeks at current token in input stream
- `match(t)`
  - if `lookahead() == t` then consume current token, else PARSE_ERROR
Recursive Descent

\[
\begin{align*}
\text{start} & \rightarrow \text{expr} \\
\text{expr} & \rightarrow \text{term} \text{ term\_tail} \\
\text{term\_tail} & \rightarrow + \text{ term\_tail} | \varepsilon \\
\text{term} & \rightarrow \text{id} \text{ factor\_tail} \\
\text{factor\_tail} & \rightarrow \varepsilon \\
\end{align*}
\]

\[
\begin{align*}
\text{case \ lookahead() of} \\
\text{id:} & \text{ match(\$\$)} \\
\text{otherwise \ PARSE\_ERROR} \\
\text{case \ lookahead() of} \\
\text{id:} & \text{ \text{term\_tail}(\$\$)} \\
\text{otherwise \ PARSE\_ERROR} \\
\text{case \ lookahead() of} \\
\text{+ :} & \text{ \text{match(\$\$)} \text{ term\_tail}(\$\$)} \\
\text{otherwise \ PARSE\_ERROR} \\
\end{align*}
\]

LL(1) Parsing Table

- But how does the parser "predict"?
- It uses the LL(1) parsing table
  - One dimension is nonterminal to expand
  - Other dimension is look-ahead token
    - We are interested in one token of look-ahead
  - Entry "nonterminal on token" contains the production to apply or contains nothing

Table-driven Top-down Parsing

Uses \text{parse\_stack}, \text{parse\_table}

\text{parse\_stack.push}(\text{start}\_symbol)

\text{loop}

\text{expected\_sym} := \text{parse\_stack}.pop

\text{if expected\_sym is terminal or \$\$ then}

\text{match(expected\_sym)}

\text{if expected\_sym = \$\$ then return \text{SUCCESS!}}

\text{else if parse\_table[expected\_sym, \text{look-ahead}] = \text{ERROR then return \text{PARSE\_ERROR}}}

\text{else foreach \ sym in \text{reverse} from production}

\text{parse\_stack.push(\sym)}
Lecture Outline

- Top-down parsing vs. bottom-up parsing
- Top-down parsing
  - Introduction
  - A backtracking parser
  - Recursive descent predictive parsing
  - Table-driven top-down predictive parsing
- LL(1) parsing table
  - FIRST, FOLLOW and PREDICT sets
  - Constructing LL(1) parsing tables

Intuition

**Top-down parsing**

- Parse tree is built from the top to the leaves
- Always expand the leftmost nonterminal

```
expr  →  term term_tail
term  →  id + id*id
id    →  factor_tail
factor_tail → * id factor_tail
factor_tail → ε
```

What production applies for `factor_tail` on `*`?
+ does not belong to an expansion of `factor_tail`.
However, `factor_tail` has an epsilon production and +
belongs to an expansion of `term_tail` which follows `factor_tail`.
Thus, predict the epsilon production.

Constructing LL(1) Parsing Tables

- We can construct an LL(1) parsing table for any context-free grammar
  - In general, the table will have multiply-defined entries.
    That is, for some nonterminal and lookahead token, more
    than one productions apply
  - A grammar whose LL(1) parsing table has no
    multiply-defined entries is said to be LL(1) grammar
  - LL(1) grammars are a very special subset of context-free grammars

FIRST and FOLLOW sets

- Let $α$ be any sequence of nonterminals and terminals
  - $\text{FIRST}(α)$ contains the set of terminals $a$ that begin
    the strings derived from $α$
  - If there is a derivation $α \Rightarrow * \ varepsilon$, then $varepsilon$ is in $\text{FIRST}(α)$
- Let $A$ be a nonterminal
  - $\text{FOLLOW}(A)$ contains the set of terminals $b$ that can
    appear immediately to the right of $A$ in some sentential
    form. In other words, there is a derivation $S \Rightarrow * ... Ab ... \Rightarrow * ...$

Computing FIRST

- Given a grammar, apply these rules until no more terminals or $\varepsilon$ can be added to any $\text{FIRST}(α)$ set
  1. If $α$ starts with a terminal $a$, then $\text{FIRST}(α) = \{ a \}$
  2. If $α$ is a nonterminal $X$ and $X \rightarrow ε$, then add $ε$ to $\text{FIRST}(X)$
  3. If $α$ is nonterminal $X \rightarrow Y_1 Y_2 ... Y_k$ then place $a$ in
     $\text{FIRST}(X)$ if for some $i$, $a$ is in $\text{FIRST}(Y_i)$ and $ε$ is in all of
     $\text{FIRST}(Y_1), ... \text{FIRST}(Y_{i-1})$. If $ε$ is in all of $\text{FIRST}(Y_1), ... \text{FIRST}(Y_{i-1})$, add $ε$ to $\text{FIRST}(X)$.
    - Everything in $\text{FIRST}(Y_i)$ is surely in $\text{FIRST}(X)$
    - If $Y_i$ does not derive $ε$, then we add nothing more;
      Otherwise, we add $\text{FIRST}(Y_i)$, and so on
Example

\[
\text{start} \rightarrow \text{expr} $$
\text{expr} \rightarrow \text{term} \text{term_tail}
\text{term} \rightarrow \text{id} \text{factor_tail}
\text{term_tail} \rightarrow + \text{term} \text{term_tail} | \epsilon
\text{factor_tail} \rightarrow * \text{id} \text{factor_tail} | \epsilon
\]

\[
\begin{align*}
\text{FIRST}(\text{start}) &= \{ \text{id} \} \\
\text{FIRST}(\text{expr}) &= \{ \text{id} \} \\
\text{FIRST}(\text{term}) &= \{ \text{id} \} \\
\text{FIRST}(\text{term_tail}) &= \{ +, \epsilon \} \\
\text{FIRST}(+ \text{term} \text{term_tail}) &= \{ + \} \\
\text{FIRST}(\text{factor_tail}) &= \{ \epsilon \}
\end{align*}
\]

Question

\[
\text{start} \rightarrow \text{list} $$
\text{list} \rightarrow \text{id} \text{list_tail}
\text{list_tail} \rightarrow \text{id} \text{list_tail} | ;
\]

- Compute FIRST sets:
  \[
  \begin{align*}
  \text{FIRST}(\text{start}) &= \\
  \text{FIRST}(\text{list}) &= \\
  \text{FIRST}(\text{list_tail}) &= \\
  \text{FIRST}(\text{list} $$) &= \\
  \text{FIRST}(, \text{id} \text{list_tail}) &=
  \end{align*}
  \]

Computing FOLLOW

- Given a grammar, apply these rules until nothing can be added to any FOLLOW(A) set
  1. If there is a production \( A \rightarrow \alpha B \beta \), then everything in FIRST(\( B \)) except for \( \epsilon \) is in FOLLOW(B)
  2. If there is a production \( A \rightarrow \alpha B \) or a production \( A \rightarrow \alpha B \beta \) where FIRST(\( B \)) contains \( \epsilon \), then everything in FOLLOW(A) is in FOLLOW(B)

\[
\begin{align*}
\text{FOLLOW}(\text{expr}) &= \{ $$ \} \\
\text{FOLLOW}(\text{term}) &= \{ +, $$ \} \\
\text{start} \Rightarrow \text{expr} $$ \Rightarrow \text{term} \text{term_tail} $$ \Rightarrow \text{term} + \text{term} $$ \Rightarrow + \text{follows} \text{term} \quad $$ \text{follows} \text{term}
\end{align*}
\]