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@cs.lists.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
    Token stream: id + id + id
    Parser
        Parse tree:

        (Parse tree simplified to fit on slide.)
```

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
    Token stream: id + id = id + id
    Parser
        Parse tree:

        Token stream is ill-formed according to our grammar, parse tree construction fails, therefore Syntax error!
        Most compiler errors occur in the parser.
```
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

```
list → id list_tail  // list is the start symbol
list_tail → , id list_tail | ;
```

Generates comma-separated lists of id's.
E.g., id; id, id, id;

For example:
```
list ⇒ id list_tail
⇒ id, id list_tail
⇒ id, 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
  - 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;
```
- 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?
  - \[ \text{id, id, id;} \]

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
  - \( S \rightarrow \text{cAd} \)
  - \( A \rightarrow \text{ab} | \text{a} \)
- Consider string \( \text{c a d} \)

Aside: Top-down Depth-first Parsing

- Input string \( \text{cad} \)
- Try \( S \rightarrow \text{cAd} \)
  - Leaf c matches input c
- Try a rule for \( A: A \rightarrow \text{ab} \)
  - Leaf a matches input a
  - But b \( \neq \) d. Backtrack to A
- Try second rule for \( A: A \rightarrow \text{a} \)
  - Leaf a matches a. d matches d
  - Done: \( S \Rightarrow \text{cAd} \Rightarrow \text{cad} \)

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

- 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, $A \to t_1 t_2 \ldots t_k t_{k+1} B \ldots$
  - 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

Exercise

- Draw the parse tree for expression
  \[ id + id \ast id + id \]

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()
  - if lookahead() == t then consume current token, else PARSE_ERROR
Recursive Descent

start → expr $$
expr → term term_tail 
term_tail → + term term_tail | ε
term → id factor_tail 
factor_tail → * id factor_tail | ε

start() 
case lookahead() of
  id: match($$) ($$ - end-of-input marker)
otherwise PARSE_ERROR

expr() 
case lookahead() of
  id: match(id); expr(); match($$)
otherwise PARSE_ERROR

term_tail() 
case lookahead() of
+: match(‘+’); term(); term_tail()
*: match(‘*’); term(); term_tail()
$: skip 
otherwise PARSE_ERROR

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 lookahead token
    - We are interested in one token of lookahead
  - Entry “nonterminal on token” contains the production to apply or contains nothing

Question

- Fill in the LL(1) parsing table for the comma-separated list grammar

Table-driven Top-down Parsing

Uses parse_stack, parse_table

expected_sym : symbol := parse_stack.pop
if expected_sym is a terminal or $$ then
  match(expected_sym)
  if expected_sym = $$ then return // SUCCESS!
else
  if parse_table[expected_sym, lookahead()] = ERROR then
    return PARSE_ERROR
  else
    production : production := parse_table[expected_sym, lookahead()]
    foreach sym : symbol in reverse from production
      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

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

Intuition

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

FIRST and FOLLOW sets

- Let α be any sequence of nonterminals and terminals
  - FIRST(α) contains the set of terminals a that begin the strings derived from α
  - If there is a derivation α ⇒* ε, then ε is in FIRST(α)
- Let A be a nonterminal
  - 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 ⇒* A...B... ⇒*... Computing FIRST

- Given a grammar, apply these rules until no more terminals or ε can be added to any FIRST(α) set
  1. If α starts with a terminal a, then FIRST(α) = {a}
  2. If α is a nonterminal X and X → ε, then add ε to FIRST(X)
  3. If α is nonterminal X → Y1... Yk, then place a in FIRST(X) if for some i, α is in FIRST(Yi) and ε is in all of FIRST(Y1),... FIRST(Yi-1), FIRST(Yi+1),... FIRST(Yk). If ε is in all of FIRST(Yi), add ε to FIRST(X).
  - Everything in FIRST(Yi) is surely in FIRST(X)
  - If Yi does not derive ε, then we add nothing more; Otherwise, we add FIRST(Yi), and so on
Example

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

FIRST(\text{start}) = \{ \text{id} \}
FIRST(\text{expr}) = \{ \text{id} \}
FIRST(\text{term}) = \{ \text{id} \}
FIRST(\text{term}\_\text{tail}) = \{ +, \varepsilon \}
FIRST(\text{term} + \text{term}\_\text{tail}) = \{ + \}
FIRST(\text{factor}\_\text{tail}) =

Question

\[
\begin{align*}
\text{start} & \rightarrow \text{list} \quad \text{list} \rightarrow \text{id} \text{list}\_\text{tail} \\
\text{list}\_\text{tail} & \rightarrow , \text{id} \text{list}\_\text{tail} | ;
\end{align*}
\]

- Compute FIRST sets:
  \[
  \begin{align*}
  \text{FIRST(\text{start})} &= \quad \text{FIRST(\text{list})} = \\
  \text{FIRST(\text{list})} &= \quad \text{FIRST(\text{list}\_\text{tail})} = \\
  \text{FIRST(\text{list} $$)} &= \quad \text{FIRST(, \text{id} \text{list}\_\text{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 \( A \rightarrow \alpha \beta \), then everything in \( \text{FIRST}(\beta) \) except for \( \varepsilon \) is in \( \text{FOLLOW}(B) \)
  (2) If there is a production \( A \rightarrow \alpha B \), or a production \( A \rightarrow \alpha \beta \) where \( \text{FIRST}(\beta) \) contains \( \varepsilon \), then everything in \( \text{FOLLOW}(A) \) is in \( \text{FOLLOW}(B) \)

Example

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

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

\[
\begin{align*}
\text{start} \Rightarrow \text{expr} \Rightarrow \text{term} \text{term}\_\text{tail} \Rightarrow \text{term} + \text{term} \text{term}\_\text{tail} \Rightarrow \text{term} + \text{term} $$
\end{align*}
\]

\[
\begin{align*}
+ \text{follows} \text{term} & \quad $$ \text{follows} \text{term}
\end{align*}
\]