Announcements

- HW1 due on Tuesday, September 11th
  - You can handwrite and scan, or type
    - File is limited to 5M in Submitt
  - Submit electronically in Submitt
    - You _can_ create a team of 1
    - Ideal size is 2
    - Maximal size is 3
  - Use forum for questions

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

- Intro to parsing: top-down vs. bottom-up
- Top-down parsing
  - Intro
  - A backtracking depth-first 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 * 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 + ...

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

- For any CFG, one can build a parser that runs in $O(n^3)$
  - Well-known algorithms

- But $O(n^3)$ time is unacceptable for a parser in a compiler!

Grammar for Comma-separated Lists

```plaintext
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 ;

Example derivation:

```
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
  1. From the top to the leaves
  2. Corresponds to a left-most derivation
  3. 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
  1. From the leaves to the top
  2. 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
  1. First $L$ stands for “left-to-right” scan of input
  2. Second $L$ stands for “left-most” derivation
  3. Parse corresponds to left-most derivation
  4. $k$ 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 using one token of lookahead?

- Yes, there is at most one choice (i.e., at most one production applies)
- This grammar is an LL(1) grammar

Fall 18 CSCI 4430, A Milanova

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
  $\text{S} \rightarrow \text{cAd}$
  $\text{A} \rightarrow \text{ab} | \text{a}$
- Consider string $\text{c ad}$

Fall 18 CSCI 4430, A Milanova

Aside: Top-down Depth-first Parsing

- Input string $\text{cad}$
- Start with start symbol
- Try $\text{S} \rightarrow \text{cAd}$
  - 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 $A$: $A \rightarrow a$
  - Leaf a matches a. d matches d
  - Done: $\text{S} \Rightarrow \text{cAd} \Rightarrow \text{cad}$

Fall 18 CSCI 4430, A Milanova

Aside: Backtracking, more generally

- A general search technique
  - Searches for a solution in (large) space
- 1. Having a 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
  \(S \Rightarrow c A d\) (try \(A \Rightarrow a b\) to get cabd, because no match, backtrack) \(c A d \Rightarrow S \rightarrow c A d\) (try \(A \Rightarrow a \) to get cad) cad works!

Sentential form is \(t_1 t_2 \ldots t_k A \ldots\) (partial solution)
- Initially \(k = 0\) and \(A\ldots\) is the start symbol
- Try a production for \(A\) (leftmost nonterminal)
  - Say, \(A \Rightarrow t_1 t_2 \ldots t_k A \ldots B\ldots\) to get \(t_1 t_2 \ldots t_k t_{k+1} t_{k+2} B\ldots\)
  - Backtrack if necessary
- Accept when there are no more nonterminals and all terminals match, or reject when there does not exist a point where there is an untried production
- Problems?

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

\[
\begin{align*}
expr &\rightarrow term \ term\_tail \\
term\_tail &\rightarrow + \ term \ term\_tail \mid \epsilon \\
term &\rightarrow id \ factor\_tail \\
factor\_tail &\rightarrow * \ id \ factor\_tail \mid \epsilon
\end{align*}
\]

- Draw the parse tree for expression \(id + id * id + id\)

Top-down, Predictive Parsing

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

\[
\begin{align*}
expr &\rightarrow expr + expr \\
expr &\rightarrow expr * term \\
expr &\rightarrow id \\
term &\rightarrow term + term\_tail \mid \epsilon \\
term &\rightarrow term * id \mid id \\
factor\_tail &\rightarrow term\_tail + term\_tail\_tail \mid \epsilon \\
factor\_tail &\rightarrow id \ factor\_tail \\
factor\_tail &\rightarrow id \ factor\_tail \mid \epsilon
\end{align*}
\]

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

LL(1) Parsing Table

- One dimension is nonterminal to expand
- Other dimension is lookahead token

<table>
<thead>
<tr>
<th>A</th>
<th>α</th>
</tr>
</thead>
</table>

E.g., entry “nonterminal A on terminal a” contains production $A \rightarrow α$
- This means, when the parser is at nonterminal A and the lookahead token in the stream is a, the parser must expand A by production $A \rightarrow α$
Question

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

\[
\begin{array}{c|c|c|c}
& \text{id} & \text{list} & \text{list_tail} \\
\hline
\text{start} & \text{list} & - & - \\
\text{list} & \text{id list_tail} & - & - \\
\text{list_tail} & - & \text{id list_tail} & ; \\
\end{array}
\]

Table-driven Top-down Parsing

Uses \text{parse_stack}, \text{parse_table}

\[
\begin{align*}
\text{parse_stack} &\leftarrow \text{start_symbol} \\
\text{loop} &
\begin{align*}
\text{expected_sym} &\leftarrow \text{parse_stack}.\text{pop} \\
\text{if} & \text{expected_sym} \text{is a terminal or} \$
\text{then} & \\
\text{match} & (\text{expected_sym}) \\
\text{else} & \\
\text{if} & \text{parse_table}[\text{expected_sym}, \text{lookahead()}] = \text{ERROR} \\
\text{then} & \text{PARSE_ERROR} \\
\text{else} & \text{production} \leftarrow \text{parse_table}[\text{expected_sym, lookahead()}] \\
\text{foreach} & \text{sym} \in \text{reverse} \text{from production} \\
\text{parse_stack} &\leftarrow \text{parse_stack}.\text{push}(\text{sym})
\end{align*}
\]

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

LL(1) Parsing Tables

- We can construct an LL(1) parsing table for any context-free grammar
  - In general, the table will contain 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 subclass of context-free grammars. Why?

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
  - $\text{FIRST}(\alpha)$ is the set of terminals $a$ that begin the strings derived from $\alpha$
  - If there is a derivation $\alpha \Rightarrow^* \varepsilon$, then $\varepsilon$ is in $\text{FIRST}(\alpha)$

- Let $A$ be a nonterminal
  - $\text{FOLLOW}(A)$ is the set of terminals $b$ (including special end-of-input marker $\$$) that can appear immediately to the right of $A$ in some sentential form:
    
    $\text{start} \Rightarrow^* \ldots Ab \ldots \Rightarrow^* \ldots$

---

**Next Class**

- Next class
  - Conclude top-down parsing
  - Start bottom-up parsing

- Read Chapter 2.3.3 in Scott's book