Programming Language Syntax: Scanning and Parsing

Read: Scott, Chapter 2.2 and 2.3.1
Lecture Outline

- Overview of scanning
- Overview of top-down and bottom-up parsing

- Top-down parsing
  - Recursive descent
  - LL(1) parsing tables
Scanning

- Scanner groups characters into tokens
- Scanner simplifies the job of the parser

- Scanner is essentially a Finite Automaton
  - Regular expressions specify the syntax of tokens
  - Scanner recognizes the tokens in the program

```
position = initial + rate * 60;

id = id + id * num;
```

Programming Languages CSCI 4430, A. Milanova
Why most programming languages disallow nested multi-line comments?

- Comments are usually handled by the scanner, which essentially is a DFA. Handling multiline comments would require recognizing \((/\*)^n(\*/)^n\) which is beyond the power of a DFA.
Calculator Language

 Tokens

\[
\begin{align*}
times & \rightarrow * \\
plus & \rightarrow + \\
id & \rightarrow \text{letter} \ ( \text{letter} \ | \ \text{digit} \ ) ^* \\
\end{align*}
\]

except for \textit{read} and \textit{write} which are keywords (keywords are tokens as well)
Ad-hoc (By hand) Scanner

skip any initial white space (space, tab, newline)
if current_char in \{ +, * \}
    return corresponding single-character token (\textit{plus} or \textit{times})
if current_char is a letter
    read any additional letters and digits
    check to see if the resulting string is \textit{read} or \textit{write}
    if so, then return the corresponding token
else return \textit{id}
else announce an \texttt{ERROR}
The Scanner as a DFA

- Start: space, tab, newline
- letter: +, *
- letter, digit: +, *
Building a Scanner

- Scanners are (usually) **automatically generated** from regular expressions:
  - Step 1: From a Regular Expression to an NFA
  - Step 2: From an NFA to a DFA
  - Step 3: Minimizing the DFA
- **lex/flex** utilities generate scanner code
- Scanner code explicitly captures the states and transitions of the DFA
Table-Driven Scanning

... 
cur_state := 1
loop
   read cur_char
   case scan_tab[cur_char, cur_state].action of
      move:
         ...
      cur_state = scan_tab[cur_char, cur_state].new_state
   recognize: // emits the token
tok = token_tab[current_state]
   unread cur_char --- push back char
   exit loop
error:

# Table-Driven Scanning

<table>
<thead>
<tr>
<th>space, tab, newline</th>
<th>*</th>
<th>+</th>
<th>digit</th>
<th>letter</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sketch of table: scan_tab and token_tab. See Scott for details.
Today’s Lecture Outline

- Overview of scanning
- Overview of top-down and bottom-up parsing

Top-down parsing
  - Recursive descent
  - LL(1) parsing tables
A Simple Calculator Language

\[\text{asst\_stmt} \rightarrow \text{id} = \text{expr} \quad \text{// asst\_stmt is the start symbol}\]
\[\text{expr} \rightarrow \text{expr} + \text{expr} \mid \text{expr} \ast \text{expr} \mid \text{id}\]

Character stream: \(\text{position} = \text{initial} + \text{rate} \ast \text{time}\)

Token stream: \(\text{id} = \text{id} + \text{id} \ast \text{id}\)

Parse tree:

(Parse tree simplified to fit on slide.)
A Simple Calculator Language

\[
\text{asst\_stmt} \rightarrow \text{id} = \text{expr} \quad // \text{asst\_stmt} \text{ is the start symbol} \\
\text{expr} \rightarrow \text{expr} + \text{expr} \mid \text{expr} * \text{expr} \mid \text{id}
\]

**Character stream:** \( \text{position} + \text{initial} = \text{rate} \times \text{time} \)

**Token stream:** \( \text{id} + \ldots \)

**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.
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!
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 (LL and LR) 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

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

Generates comma-separated lists of \text{id}'s.
E.g., \text{id ; id, id, id, id ;}

Example derivation:
\[ \text{list} \Rightarrow \text{id list\_tail} \]
\[ \Rightarrow \text{id , id list\_tail} \]
\[ \Rightarrow \text{id , id ;} \]
Top-down Parsing

- Terminals are seen in the order of appearance in the token stream
  
  \[ \text{id}, \text{id}, \text{id}; \]

- 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

\[
\begin{align*}
\text{list} & \rightarrow \text{id} \text{ list}_tail \\
\text{list}_tail & \rightarrow , \text{id} \text{ list}_tail \mid ;
\end{align*}
\]
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

list → id list_tail
list_tail → , id list_tail | ;
Today’s Lecture Outline

- Overview of scanning
- Overview of top-down and bottom-up parsing

- Top-down parsing
  - Recursive descent
  - LL(1) parsing tables
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 (i.e., for any input) what production to applies, based on one token of lookahead?

  \[
  \text{id , id , id ;}
  \]

  \[
  \uparrow \quad \uparrow \quad \uparrow \quad \uparrow
  \]

- Yes, there is at most one choice (i.e., at most one production applies)

- This grammar is an LL(1) grammar

\[
\text{list} \rightarrow \text{id list_tail}
\]
\[
\text{list_tail} \rightarrow \text{, id list_tail} \mid ;
\]
A new grammar

What language does it generate?
  - Same, comma-separated lists

Can we predict based on one token of lookahead?

```latex
list \rightarrow list\_prefix ;
list\_prefix \rightarrow list\_prefix , \text{id} | \text{id}
```

```latex
\text{id} , \text{id} , \text{id} ;
```
Top-down Predictive Parsing

- Back to predictive parsing

- “Predicts” production to apply based on one or more lookahead token(s)
  - Parser always gets it right!
  - There is no need to backtrack, undo expansion, then try a different production

- 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:

$$\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{expr} \\
& \quad \quad | \text{expr} \times \text{expr} \\
& \quad \quad | \text{id} \\
\text{expr} & \rightarrow \text{expr} + \text{term} \mid \text{term} \\
\text{term} & \rightarrow \text{term} \times \text{id} \mid \text{id} \\
\text{expr} & \rightarrow \text{term} \text{term}_\text{tail} \\
\text{term}_\text{tail} & \rightarrow + \text{term} \text{term}_\text{tail} \mid \varepsilon \\
\text{term} & \rightarrow \text{id} \text{factor}_\text{tail} \\
\text{factor}_\text{tail} & \rightarrow \times \text{id} \text{factor}_\text{tail} \mid \varepsilon
\end{align*}$$
Exercise

- Draw parse tree for expression

\[ \text{id} + \text{id} \ast \text{id} + \text{id} \]
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

\[
\text{start} \rightarrow \text{expr } $$
\]
\[
\text{expr} \rightarrow \text{term } \text{term}_\text{tail}
\]
\[
\text{term} \rightarrow \text{id } \text{factor}_\text{tail}
\]
\[
\text{term}_\text{tail} \rightarrow + \text{ term } \text{term}_\text{tail} | \varepsilon
\]
\[
\text{factor}_\text{tail} \rightarrow * \text{id } \text{factor}_\text{tail} | \varepsilon
\]

\begin{align*}
\text{start()} & \quad \text{case lookahead() of} \\
& \quad \text{id: expr(); match($$) ($$ - end-of-input marker)} \\
& \quad \text{otherwise PARSE\_ERROR}
\end{align*}

\begin{align*}
\text{expr()} & \quad \text{case lookahead() of} \\
& \quad \text{id: term(); term}_\text{tail()} \\
& \quad \text{otherwise PARSE\_ERROR}
\end{align*}

\begin{align*}
\text{term}_\text{tail()} & \quad \text{case lookahead() of} \\
& \quad \text{+: match('+'); term(); term}_\text{tail()} \\
& \quad $$: \text{skip} \\
& \quad \text{otherwise: PARSE\_ERROR}
\end{align*}

Predicting production \( \text{term}_\text{tail} \rightarrow + \text{ term } \text{term}_\text{tail} \)

Predicting epsilon production \( \text{term}_\text{tail} \rightarrow \varepsilon \)
Recursive Descent

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

\[
\text{term()}
\]

\[
\begin{align*}
\text{case lookahead() of} \\
\text{id: match(‘id’); factor\_tail()} \\
\text{otherwise: PARSE\_ERROR}
\end{align*}
\]

\[
\text{factor\_tail()}
\]

\[
\begin{align*}
\text{case lookahead() of} \\
\text{*: match(‘*’); match(‘id’); factor\_tail();} \\
+,$$\$: \text{skip} \\
\text{otherwise PARSE\_ERROR}
\end{align*}
\]

Predicting production \( \text{factor\_tail} \rightarrow *\text{id} \text{ factor\_tail} \)

Predicting production \( \text{factor\_tail} \rightarrow \varepsilon \)
LL(1) Parsing Table

- But how does the parser “predict”?  
  - E.g., how does the parser know to expand a `factor_tail` by `factor_tail → ε` on `+` and `$$`?
- 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
LL(1) Parsing Table

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>α</td>
</tr>
</tbody>
</table>

- E.g., entry “nonterminal A on terminal a” contains production $A \rightarrow \alpha$

Meaning: when parser is at nonterminal $A$ and lookahead token is $a$, then parser expands $A$ by production $A \rightarrow \alpha$
## LL(1) Parsing Table

<table>
<thead>
<tr>
<th>Production</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start $\rightarrow$ expr $$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr $\rightarrow$ term term_tail</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>term $\rightarrow$ id factor_tail</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>term_tail $\rightarrow$ + term term_tail</td>
<td></td>
<td></td>
<td></td>
<td>ε</td>
</tr>
<tr>
<td>factor_tail $\rightarrow$ * id factor_tail</td>
<td></td>
<td></td>
<td></td>
<td>ε</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Production</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>start</td>
<td>expr $$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>expr</td>
<td>expr</td>
<td>term term_tail</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>term</td>
<td>term</td>
<td>id factor_tail</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>term_tail</td>
<td>term_tail</td>
<td>-</td>
<td>+</td>
<td>term term_tail</td>
</tr>
<tr>
<td>factor_tail</td>
<td>factor_tail</td>
<td>-</td>
<td>ε</td>
<td>* id factor_tail</td>
</tr>
</tbody>
</table>
Question

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

\[
\begin{align*}
\text{start} & \rightarrow \text{list } \$$ \\
\text{list} & \rightarrow \text{id list_tail} \\
\text{list_tail} & \rightarrow , \text{id list_tail} \mid ;
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th></th>
<th>;</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>list $$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>list</td>
<td>id list_tail</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>list_tail</td>
<td>-</td>
<td>, id list_tail</td>
<td>;</td>
<td>-</td>
</tr>
</tbody>
</table>
The End