Programming Language Syntax: Scanning and Parsing

Read: Scott, Chapter 2.2 and 2.3.1
Lecture Outline

- Quiz 1
- 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

\[
\text{position} = \text{initial} + \text{rate} \times 60;
\]

\[
id = \text{id} + \text{id} \times \text{num}
\]
Preprocessor

---

Preprocessor

Scanner
Parser (ad-hoc)
Semantic Analyzer

Compiler

Preprocessor + Lexer
Parser
Semantic Analyzer

...
Calculator Language

- Tokens

  \[ \text{times} \rightarrow * \quad \text{// } \ast \text{ is a character in calculator language} \]
  \[ \text{plus} \rightarrow + \]
  \[ \text{id} \rightarrow \text{letter} \ (\ \text{letter} \ | \ \text{digit} \ )^* \quad \text{// } \ast \text{ is the Kleene star} \]

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

Start

1

2

3

4

letter

space, tab, newline

letter, digit

letter

+
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></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-</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></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

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

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

![Diagram showing the flow from character stream to parser to parse tree]

**Token stream:** id = id + id * id

**Parse tree:**

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

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

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

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

\begin{center}
\begin{tikzpicture}
  \node {Scanner};
  \node [below] {Parser};
  \node [above] (expr) \text{Token stream: \text{id} + ...};
  \node [above] (stmt) \text{Parse tree:};
  \node [above] (rate) \text{Token stream is ill-formed according to our grammar, parse tree construction fails, therefore Syntax error!};
  \node [above] (time) \text{Most compiler errors occur in the parser.};
  \path (stmt) -- (expr);
  \path (expr) -- (stmt);
  \path (rate) -- (stmt);
  \path (time) -- (stmt);
\end{tikzpicture}
\end{center}
Given an arbitrary CFG, one can build a parser that parses a string of length $n$ in (essentially) $O(n^3)$
- 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

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

\[
\text{list\_tail} \rightarrow , \ \text{id} \ \text{list\_tail} \mid ;
\]

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

Example derivation:
\[
\text{list} \Rightarrow \text{id} \ \text{list\_tail} \\
\Rightarrow \text{id} , \ \text{id} \ \text{list\_tail} \\
\Rightarrow \text{id} , \ \text{id} ;
\]

Parse tree:
Top-down Parsing

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

- The parse tree is constructed
  - From the top to the leaves
  - Corresponds to a leftmost derivation

- Look at leftmost nonterminal in current sentential form, and lookahead terminal and “predict” which production to apply

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

E.g. on \text{list} and \text{id} expand by

\[
\text{list} \rightarrow \text{id} \; \text{list}_\text{tail}.
\]

\[
\begin{array}{c}
\text{list} \\
\downarrow \\
\text{id} \\
\downarrow \\
, \; \text{id} \; \text{list}_\text{tail} \\
\downarrow \\
, \; \text{id} \; \text{list}_\text{tail} \\
\downarrow \\
; \\
\end{array}
\]
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 rightmost derivation in reverse

Two main parser actions:
1. Shift token on parse tree and advance input pointer
2. Reduce nodes into intermediate node. E.g. \[ , id \, list\_tail \] reduce into \[ list\_tail \].

\[
\begin{align*}
  &list \rightarrow id \, list\_tail \\
  &list\_tail \rightarrow , \, id \, list\_tail \mid ;
\end{align*}
\]
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 leftmost derivation
    - Parse corresponds to leftmost derivation
  - k stands for “need k tokens of lookahead to predict”
- We are interested in LL(1)
Can we always predict (i.e., for any input) what production to applies, based on 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
A new grammar

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

Can we predict based on one token of lookahead?

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

No. Seeing \text{id}, parser has no way of knowing whether it is a list of \text{id}, \text{id}, \text{id},... or just \text{id}.

Grammar is not LL(1).
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)
    - No ambiguous grammar is LL(1).
- Unambiguous version:
  - Still not LL(1). Why?
- LL(1) version:
  - Eliminates left recursion.
Exercise

- Draw parse tree for expression

\[
\text{id} + \text{id} * \text{id} + \text{id}
\]
Recursive Descent

- Each **nonterminal** has a procedure
- The right-hand-sides (rhs) of productions for that 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} \; $$
\][
\[
\text{expr} \rightarrow \text{term} \; \text{term\_tail}
\]

\[
\text{term\_tail} \rightarrow + \; \text{term} \; \text{term\_tail} \mid \varepsilon
\]

\[
\text{term} \rightarrow \text{id} \; \text{factor\_tail}
\]

\[
\text{factor\_tail} \rightarrow * \; \text{id} \; \text{factor\_tail} \mid \varepsilon
\]

start()

    case lookahead() of
        \text{id}: expr(); match($$) ($$ - end-of-input marker)
    otherwise PARSE\_ERROR

expr()

    case lookahead() of
        \text{id}: term(); term\_tail()
    otherwise PARSE\_ERROR

term\_tail()

    case lookahead() of
        +: match('+'); term(); term\_tail()
        $$: skip
    otherwise: PARSE\_ERROR
Recursive Descent

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

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

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

Predicting production \text{factor\_tail} \rightarrow *\text{id \ 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 $A \rightarrow α$

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

\[
\begin{align*}
\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} \mid \varepsilon \\
\text{factor}_\text{tail} & \rightarrow * \ \text{id } \ \text{factor}_\text{tail} \mid \varepsilon
\end{align*}
\]

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

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

```
start → list $$
list → id list_tail
list_tail → , id list_tail | ;
```

<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