Lexing and Recursive Descent Parsing

February 4, 2002
CS 131: Programming Languages

What is Lexing?
- Lexing: breaking program source text into "words" called tokens.
  - Other names for process: tokenizing, scanning
- Tokens include keywords, punctuation, identifiers, constants, etc.

Why Do (Separate) Lexing?
- Simplifies the work of the parser
  - Parsing is generally more complex and expensive
  - Hides uninteresting details
    - Strip out source comments
    - Strip out whitespace (sometimes)
    - Case of letters (sometimes)
- Benefits of modularity
  - Easier to understand
  - Easier to modify

Example

```c
float match0(char *s) /* find a zero */
{
    if (!strncmp(s, "0.0", 3))
        return 0.0;
}
```
Lexing Challenges

- Non-regular token specifications
  - Fortran Hollerith constants: 9H
  - Compilers
- Skipping nested comments

- Layout significance
  - Position of code determines meaning
  - E.g., offside rule
    ```
    if x = 0 then x = 1
    if y = 0 then y = x + z
    z := 0 where
    else x = 2
    z := 1 z = 4
    z := x + y
    ```

Lexer Implementations

- Handwritten
  - E.g., implementation of automaton
- Machine-Generated
  - Tools like Lex, Flex, ..., take descriptions of tokens
    (regular expressions and the code to run when
    these are seen) and spit out code for a lexer.
  - May be difficult to handle above "challenges"

Lexing Challenges

- Context-dependent tokenization
  - May have to use lookahead or other information
  - E.g., Fortran
    ```
    DO 10 I = 1, 15
    DO 10 I = 1.15
    REAL X
    REAL X = 3.5
    INTEGER FUNCTION A(I)
    ```

Parsing Context-Free Languages

- Algorithms exist for parsing arbitrary context-free languages.
  - But these take $O(n^2)$ time, where $n$ is the length
    of the input string.
  - Theoretically possible to do better, e.g., $O(n^{2.81})$
- Certain particular CF grammars allow
  unambiguous, linear-time parsing.
  - Programming languages designed to fall into this
class!
Predictive Parsing

• Corresponds to constructing a parse tree out of the input "top-down"

• At each step, choose a non-terminal and "predict" how it will be expanded
  - Try to use information about the input to guide prediction
    • Generally the first character or characters of the input.
  - In worst case, end up trying all possibilities (breadth-first search)
  - Fortunately, for some grammars we can always find the (unique) right prediction

Predictive Parsing Example

<table>
<thead>
<tr>
<th>Prediction Stack</th>
<th>Input Stream</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>a b b</td>
<td>Predict S→aB</td>
</tr>
<tr>
<td>a B</td>
<td>a b b</td>
<td>Match</td>
</tr>
<tr>
<td>B</td>
<td>a b b</td>
<td></td>
</tr>
</tbody>
</table>

When Does This Work?

• Predictive parsing looks at the first token(s) of the input and makes a prediction.

• A grammar which requires at most \( k \) tokens of the input to always choose the right production is called LL(\( k \)).
  - The grammar of the previous example is LL(1).

• A formal language is called LL(\( k \)) if there exists at least one LL(\( k \)) grammar
  - Along with potentially many non-LL(\( k \)) grammars.

First, Follow, and Nullable

• We use \( \alpha \) and \( \beta \) to denote “sentential forms”
• FIRST(\( \alpha \)) is the set of terminals \( t \) such that \( \alpha \rightarrow^* t\beta \)
  That is, FIRST(\( \alpha \)) is the set of terminals that can begin a string derived from \( \alpha \).
• FOLLOW(\( X \)) is the set of terminals \( t \) such that \( S \rightarrow^* \alpha X t \beta \)
  That is, the set of terminals that can immediately follow \( X \) in some derivation from the start symbol \( S \).
• Finally, we say that \( \alpha \) is nullable if \( \alpha \rightarrow^* \varepsilon \).
**LL(1) Prediction Algorithm**

- Assume the prediction stack has a nonterminal \( \mathcal{A} \) and the input starts with \( t \).
- We choose to predict a use of the rule
  \[ \mathcal{A} \rightarrow \alpha \]
  if either
  1. \( t \in \text{FIRST}(\alpha) \)
  2. or, \( \alpha \) is nullable and \( t \in \text{FOLLOW}(\mathcal{A}) \)
- In an LL(1) grammar, there is always at most one rule satisfying one or both of these.

**Recursive Descent**

- Implementation of predictive parsing where the prediction stack is implicit
- Defines a function for each nonterminal to find the prefix of the input stream matching that nonterminal
- Works by choosing a production for the nonterminal and recursively matching the right-hand-side against the input stream.

**Recursive Descent Example**

```plaintext
datatype token = IF | THEN | ELSE | BEGIN | END | PRINT | SEMI | EQ | NUM of int
exception ParseError
fun eat(tok,t::ts) =
  if (tok=t) then ts else raise ParseError
  | eat(_,[[]]) = raise ParseError

fun S(t::ts) =
  (case t of
   IF => let val ts2 = E(ts)
            val ts3 = eat(THEN,ts2)
            val ts4 = S(ts3)
            val ts5 = eat(ELSE,ts4)
            val ts6 = S(ts5)
   in ts6 end
   | BEGIN => let val ts2 = S(ts)
            val ts3 = L(ts2)
   in ts3 end
   | PRINT => let val (exp, ts2) = E(ts)
            in ts2 end
   | _ => raise ParseError
  )
```

**Yes/No**

\( S : \text{token list} \rightarrow \text{token list} \)

```plaintext
fun S(t::ts) =
  (case t of
   IF => let val ts2 = E(ts)
            val ts3 = eat(THEN,ts2)
            val ts4 = S(ts3)
            val ts5 = eat(ELSE,ts4)
            val ts6 = S(ts5)
   in ts6 end
   | BEGIN => let val ts2 = S(ts)
            val ts3 = L(ts2)
   in ts3 end
   | PRINT => let val (exp, ts2) = E(ts)
            in ts2 end
   | _ => raise ParseError
  )
```
Yes/No

(* L : token list -> token list *)
and L(t::ts) =
(case t of
  END => ([], ts)
| SEMI => let val ts2 = S(ts)
           val ts3 = L(ts2)
          in
           ts3
          end
| _ => raise ParseError)

(* E : token list -> token list *)
and E((Num n)::EQ::(Num m)::ts) = ts
| E _ = raise ParseError

Generating Abstract Syntax

datatype stmt = COND of exp * stmt * stmt
| BLOCK of stmt list
| OUTPUT of exp
and exp = CONST of int
| EQTEST of exp * exp

(* S : token list -> stmt * token list *)
(* L : token list -> stmt list * token list *)
(* E : token list -> exp * token list *)

Generating Abstract Syntax

fun S(t::ts) =
(case t of
  IF => let val (exp1,ts2) = E(ts)
          val ts3 = eat(THEN,ts2)
          val (stmt1,ts4) = S(ts3)
          val ts5 = eat(ELSE,ts4)
          val (stmt2,ts6) = S(ts5)
          in
          (COND(exp1,stmt1,stmt2), ts6) end
| BEGIN => let val (stmt, ts2) = S(ts)
          val (stmts, ts3) = L(ts2)
          in
          (BLOCK(stmt::stmts),ts3) end
| PRINT => let val (exp, ts2) = E(ts)
          in
          (OUTPUT(exp),ts2) end
| _ => raise ParseError)
Problems: Left Recursion

- A grammar is said to be left-recursive if there is a production sequence of the form
  \[ X \rightarrow^* X \ldots \]
- Left-recursive rules break recursive descent.
  - Obvious left recursion can sometimes be replaced by right recursion

\[
\begin{align*}
E & \rightarrow E - n \\
E & \rightarrow n
\end{align*}
\]

\[
\begin{align*}
E \rightarrow n E' \\
E' \rightarrow - n E' \\
E' & \mid \epsilon
\end{align*}
\]

(here \( n \) represents an arbitrary integer)

Problem

- Another problem arises if two productions for the same nonterminal begin with the same nonterminal(s):

\[
\begin{align*}
S & \rightarrow a X \\
S & \rightarrow a Y
\end{align*}
\]

- A fix?

LL(\( k \)) Summary

- There exist non-LL(\( k \)) formal languages
  \[ \{a^n b^n \mid n \geq 1\} \cup \{a^n c^n \mid n \geq 1\} \]
- Many grammars are not LL(\( k \))
  - Particularly ones found in language definitions.
- But, most programming languages are either LL(\( k \)) or "close enough"
  - i.e., can be parsed with recursive descent & a few hacks.

LR(\( k \))

- Main other class of grammars for PLs
  - Parsing is bottom-up construction of parse trees
  - Related variants include SLR(\( k \)), LALR(\( k \)), ...
  - Apply to "larger" class of useful grammars
    - E.g., no problem with left-recursion
    - Often used by parser generators (e.g., Yacc)
- Take the compilers course next spring for more info