Class Information

ANNOUNCEMENTS

• Third homework has been posted: Due Friday, October 13, 11:59pm.

• Next class in AB2125 - moved only for this Friday.

• Sample solution for HW1 and grades have been posted.

• First project will be posted soon.

• Midterm exam: Friday, October 27, in class.

• Don’t forget to work on your C and Linux skills (ilab).
Review: LL(1) Parsing

Basic idea:

For any two productions $A ::= \alpha | \beta$ with $\alpha \in (T \cup N)^*$ and $\beta \in (T \cup N)^*$, we would like a distinct way of choosing the correct production to expand.

For $\alpha \in (T \cup N)^*$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first token in some string derived from $\alpha$.

That is

\[
\begin{align*}
\text{a} & \in \text{FIRST}(\alpha) \iff \text{a} \rightarrow^{*} \text{a}\gamma \text{ for some } \gamma \in (T \cup N)^* \\
\epsilon & \in \text{FIRST}(\alpha) \iff \text{a} \rightarrow^{*} \epsilon
\end{align*}
\]

For a non-terminal $A$, define $\text{FOLLOW}(A)$ as the set

\[
\begin{align*}
\text{a} & \in \text{FOLLOW}(A) \iff S \rightarrow^{*} \alpha A\text{a}\gamma \text{ for some } \\
\alpha, \gamma & \in (T \cup N)^*, \text{a} \in T, \text{and S the start symbol.}
\end{align*}
\]

Thus, a non-terminal’s FOLLOW set specifies the tokens that can legally appear after it.

FOLLOW sets are not defined for terminal symbols.

FIRST and FOLLOW sets can be constructed automatically
Review: LL(1) Parsing

Define \( FIRST^+(\delta) \) for rule \( A ::= \delta \)

- \( FIRST(\delta) - \{ \epsilon \} \cup \text{Follow}(A) \), if \( \epsilon \in FIRST(\delta) \)
- \( FIRST(\delta) \) otherwise

A grammar is LL(1) iff

\[
(A ::= \alpha \text{ and } A ::= \beta) \text{ implies } FIRST^+(\alpha) \cap FIRST^+(\beta) = \emptyset
\]
Back to Our Example

\[ S ::= a \, S \, b \mid \epsilon \]

\[ FIRST(aSb) = \{a\} \]
\[ FIRST(\epsilon) = \{\epsilon\} \]
\[ FOLLOW(S) = \{\text{eof, b}\} \]

\[ FIRST^+(aSb) = \{a\} \]
\[ FIRST^+(\epsilon) = (FIRST(\epsilon) - \{\epsilon\}) \cup FOLLOW(S) = \{\text{eof, b}\} \]

Is the grammar LL(1)?
Table-Driven LL(1) Parser

LL(1) parse table

Example:

S ::= a S b | ε

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>ε</td>
<td>ε</td>
<td>error</td>
</tr>
</tbody>
</table>

How to parse input a a a b b b?
Table-driven predictive parsing algorithm

Input: a string $w$ and a parsing table $M$ for $G$

push $\text{eof}$
push $\text{Start Symbol}$
token $\leftarrow$ next_token()

$X \leftarrow \text{top-of-stack}$
repeat
  if $X$ is a terminal then
    if $X = \text{token}$ then
      pop $X$
      token $\leftarrow$ next_token()
    else error()
  else /* $X$ is a non-terminal */
    if $M[X, \text{token}] = X \rightarrow Y_1 Y_2 \cdots Y_k$ then
      pop $X$
push $Y_k, Y_{k-1}, \cdots, Y_1$
    else error()

  $X \leftarrow \text{top-of-stack}$
until $X = \text{eof}$

if $\text{token} \neq \text{eof}$ then error()

See also Aho, Lam, Sethi, and Ullman, Figure 4.20, page 227
Predictive Parsing

Now, a predictive parser looks like:

Rather than writing code, we build tables.

Building tables can be automated!
Generating a Table-Driven Parser

A parser generator system often looks like:

This is true for both top down and bottom up parsers

$LL(1)$: left to right, leftmost derivation, lookahead(1)

$LR(1)$: left to right, reverse rightmost derivation, lookahead(1)
Project 1: tinyL Language

<program> ::= <stmtlist> .
<stmtlist> ::= <stmt> <morestmts>
<morestmts> ::= ; <stmtlist> | ϵ
<stmt> ::= <assign> | <print>
<assign> ::= <variable> = <expr>
<print> ::= ! <variable>
<expr> ::= + <expr> <expr> |
           - <expr> <expr> |
           * <expr> <expr> |
           / <expr> <expr> |
           <variable> |
<digit>

<variable> ::= a | b | c | d | e | f | g | h | i | j | k | x | y | z
<digit> ::= 0 | 1 | 2 | 3 | ... | 9
Recursive Descent Parsing

Now, we can produce a simple recursive descent parser from our favorite LL(1) expression grammar.

Recursive descent is one of the simplest parsing techniques used in practical compilers:

- Each non–terminal has an associated parsing procedure that can recognize any sequence of tokens generated by that non–terminal.

- There is a main routine to initialize all globals (e.g.: token) and call the start symbol. On return, check whether token == eof, and whether errors occurred. (Note: left-to-right evaluation of expressions).

- Within a parsing procedure, both non–terminals and terminals can be “matched”:
  - non–terminal A — call parsing procedure for A
  - token t — compare t with current input token; if match, consume input, otherwise ERROR

- Parsing procedures may contain code that performs some useful “computation” (syntax directed translation).
Recursive Descent Parsing (pseudo code)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>eof</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aSb</td>
<td>ε</td>
<td>ε</td>
<td>error</td>
</tr>
</tbody>
</table>

main: {
    token := next_token( );
    if (S( ) and token == eof) print ‘‘accept’’ else print ‘‘error’’;
}

bool S:
    switch token {
    case a:  
        token := next_token( );
        if (not S( )) return false; // recursive call to S;
        if token == b {
            token := next_token( )
            return true;
        }
        else
            return false;
        break;
    case b,
    case eof: return true;
        break;
    default: return false;
    }

How to parse input a a a b b b ?

CS 314 fall’17, lecture 9 page 11
Syntax Directed Translation

Examples:

1. Interpreter
2. Code generator
3. Type checker
4. Performance estimator

Use hand-written recursive descent LL(1) parser
Syntax-Directed Translation Skeleton

\[
\begin{align*}
<\text{expr}>& ::= + <\text{expr}><\text{expr}> | \\
<\text{digit}>& ::= 0|1|2|3|\ldots|9
\end{align*}
\]

... expr:

\[
\text{switch} \text{ token} \{ \\
\text{ case } +: \text{ token } := \text{ next\_token( )}; \\
\quad /*1*/ \text{ expr( )}; /*2*/ \text{ expr( )}; /*3*/ \text{ return}; \\
\text{ case } 0..9: /*4*/ \text{ return digit( )}; \\
\}
\]

... digit:

\[
\text{switch} \text{ token} \{ \\
\text{ case } 1: \text{ token } := \text{ next\_token( )}; \\
\quad /*5*/ \text{ return}; \\
\text{ case } 2: \text{ token } := \text{ next\_token( )}; /*6*/ \text{ return}; \\
\ldots
\}
\]

This skeleton code implements a tree walk over the parse tree. Define return values and put code where you need it.
Example: Interpreter

<expr> ::= + <expr> <expr> |

<digit>

<digit> ::= 0 | 1 | 2 | 3 | ... | 9

int expr: // returns value of expression
int val1, val2; // values
switch token {
    case +: token := next_token( );
    val1 = expr( ); val2 = expr( );
    return val1+val2;
    case 0..9: return digit( );
}

int digit: // returns value of constant
switch token {
    case 1: token := next_token( );
    return 1;
    case 2: token := next_token( );
    return 2;
    ...
}

Example: Interpreter

What happens when you parse subprogram

“+ 2 + 1 2” ?

The parsing produces:

5
Next Lecture

Things to do:
Start programming in C. Check out the web for tutorials.

Next time:

• More syntax-directed translation examples
• Project 1 overview
• Programming in C, pointers, explicit memory allocation
• Read Scott 5.1 - 5.3 (some background - chapter on CD)