Class Announcements

- Sample solution for first homework is available on canvas
- Second homework deadline extension: Thursday, February 16
- Third homework has been posted. No deadline extension possible. Due Tuesday, 21.
- Midterm 1: Friday, February 24, in class, closed book and notes
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>
Table-driven predictive parsing algorithm

**Input:** a string \( w \) and a parsing table \( M \) for \( G \)

```
push eof /* symbol used as bottom of stack marker */
push Start Symbol
token ← next_token() /* get first token */
X ← top-of-stack
repeat
    if X is a terminal then
        if X = token then
            pop X
            token ← next_token()
        else error()
    else /* X is a non-terminal */
        if \( M[X, token] = X \rightarrow Y_1 Y_2 \cdots Y_k \) then
            pop X
            push \( Y_k, Y_{k-1}, \cdots, Y_1 \)
        else error()

    X ← top-of-stack
until X = eof

if token ≠ eof then error()
```

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

Now, a predictive parser looks like:

![Diagram of a predictive parser]

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

\textit{LL(1)}: left to right, leftmost derivation, lookahead(1)

\textit{LR(1)}: left to right, reverse rightmost derivation, lookahead(1)
Recursive Descent Parsing

Now, we can produce a simple recursive descent parser for an \texttt{LL(1)} grammar.

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

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

- There is a \texttt{main} routine to initialize all \texttt{globals} (e.g.: \texttt{token}) and call the start symbol. On return, check whether \texttt{token} == \texttt{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 \texttt{A} — call parsing procedure for \texttt{A}
  - token \texttt{t} — compare \texttt{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;
    case b,
        case eof: return true;
    break;
    default: return false;
}
```

How to parse input a a a b b b?
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

<expr> ::= + <expr> <expr> |
<digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

... expr:

switch token {
  case +: token := next_token( );
      /*1*/ expr( ); /*2*/ expr( ); /*3*/
        return;
  case 0..9: /*4*/ return digit( );
}

... digit:

switch token {
  case 1: token := next_token( );
        /*5*/
        return ;
  case 2: token := next_token( ); /*6*/ return;
  ...
}

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

\[
\langle \text{expr} \rangle ::= + \langle \text{expr} \rangle \langle \text{expr} \rangle | \langle \text{digit} \rangle
\]

\[
\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

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
Example: Simple Code Generation

\langle \text{expr} \rangle ::= + \langle \text{expr} \rangle \langle \text{expr} \rangle | \langle \text{digit} \rangle

\langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

int \text{expr}: // returns target register of operation
  int \text{target\_reg}; // "fresh" register
  int \text{reg1, reg2}; // other registers
  switch \text{token} {
    case +: \text{token} := \text{next\_token}();
      \text{target\_reg} = \text{next\_register}();
      \text{reg1} = \text{expr}(); \text{reg2} = \text{expr}();
      \text{CodeGen}(\text{ADD, reg1, reg2, target\_reg});
      \text{return target\_reg};
    case 0..9: \text{return digit}();
  }

int \text{digit}: // returns digit target register of operation
  int \text{target\_reg} = \text{next\_register}(); // "fresh" register
  switch \text{token} {
    case 1: \text{token} := \text{next\_token}();
      \text{CodeGen}(\text{LOADI, 1, target\_reg});
      \text{return target\_reg};
    case 2: \text{token} := \text{next\_token}();
      \text{CodeGen}(\text{LOADI, 2, target\_reg});
      \text{return target\_reg};
    ...
  }

Example: Simple Code Generation

What happens when you parse subprogram

“+ 2 + 1 2”?

Assumption:

first call to next_register() will return 1

The parsing produces:

loadI 2 => r2
loadI 1 => r4
loadI 2 => r5
add r4, r5 => r3
add r2, r3 => r1
Next Lecture

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

Next time:

- Recursive descent parsing examples (cont.)
- Project 1 overview
- Programming in C, pointers, explicit memory allocation
- Read ILOC description available on our class web page.