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

	a	b	eof	other
S	aSb	ϵ	ϵ	error

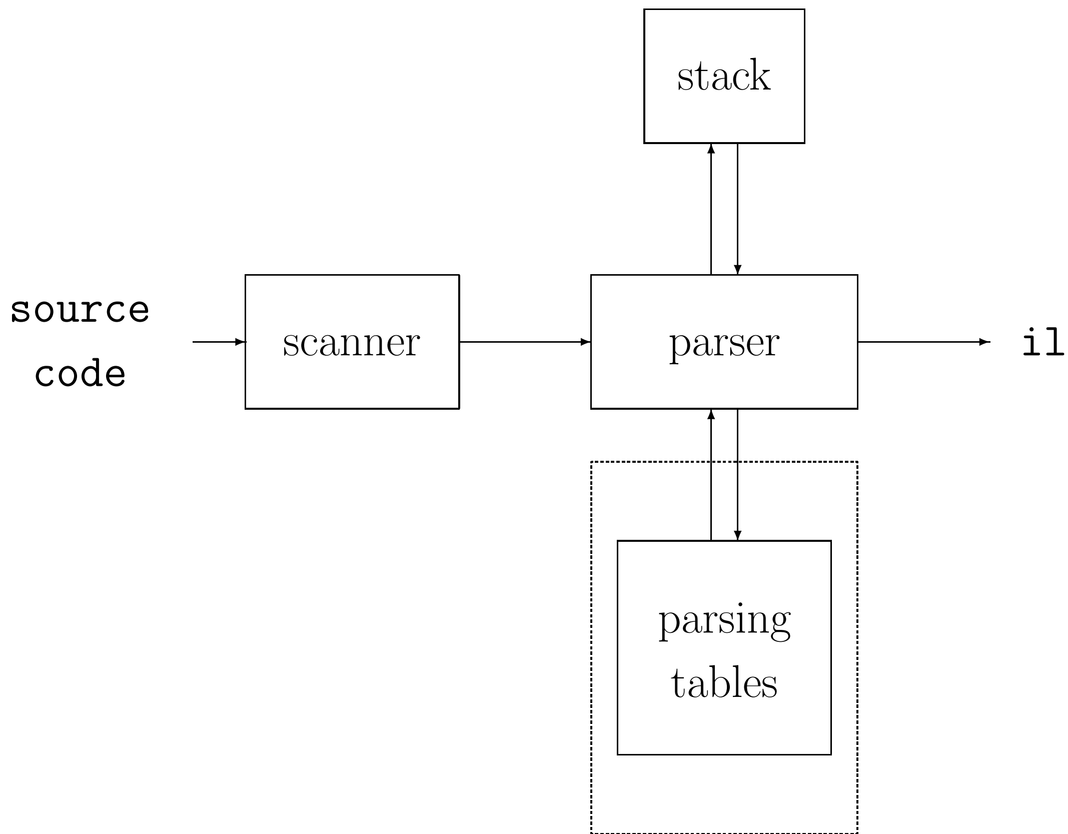
Table-driven predictive parsing algorithm

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 → Y1Y2 ··· Yk then
         pop X
         push Yk, Yk−1, ··· , Y1
      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

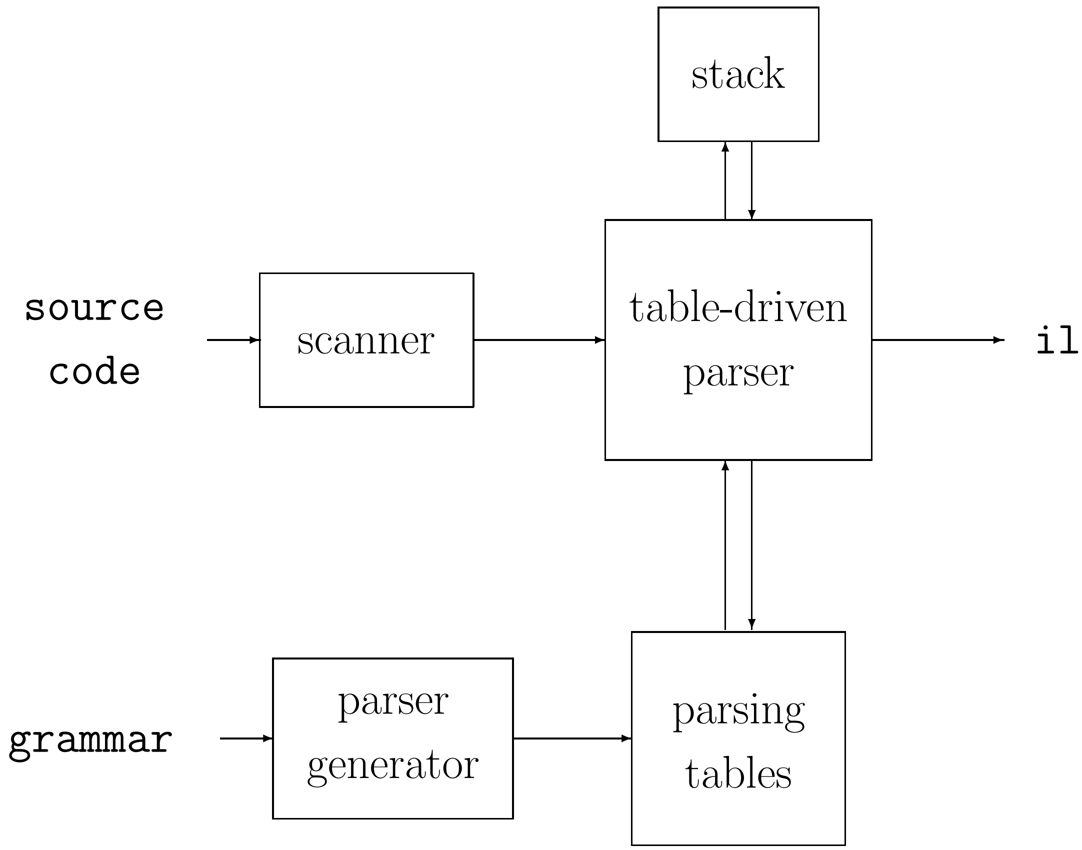
Now, a predictive parser looks like:



Rather than writing code, we build tables.

Building tables can be automated!

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)

Recursive Descent Parsing

Now, we can produce a simple recursive descent parser for an LL(1) 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)

	a	b	eof	other
S	aSb	ϵ	ϵ	error

```
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** ?

Syntax Directed Translation

Examples:

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

$\boxed{\text{Use hand-written recursive descent LL(1) parser}}$

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

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

<digit>

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

“$+\ \mathbf{2}\ +\ \mathbf{1}\ \mathbf{2}$” ?

The parsing produces:

5

Example: Simple Code Generation

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

<digit>

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

```
int expr:  // returns target register of operation
  int target_reg; // ``fresh'' register
  int reg1, reg2; // other registers
  switch token {
    case +:    token := next_token( );
               target_reg = next_register( );
               reg1 = expr( ); reg2 = expr( );
               CodeGen(ADD, reg1, reg2, target_reg);
               return target_reg;
    case 0..9: return digit( );
  }

int digit:  // returns target register of operation
  int target_reg = next_register( ); // ``fresh'' register
  switch token {
    case 1:    token := next_token( );
               CodeGen(LOADI, 1, target_reg);
               return target_reg;
    case 2:    token := next_token( );
               CodeGen(LOADI, 2, target_reg);
               return target_reg;
    ...
  }
```

Example: Simple Code Generation

What happens when you parse subprogram

“+ **2** + **1** **2**” ?

Assumption:

$\boxed{\text{first call to } \texttt{next\_register( )} \text{ will return 1}}$

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.