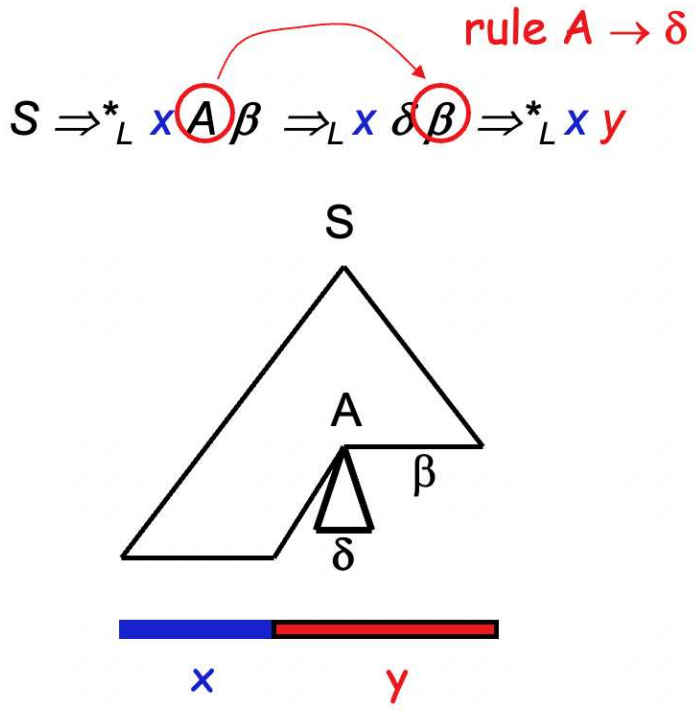
Class Announcements

- Second homework has been posted. Due next Tuesday.
- Third homework will be posted over the weekend.
- Midterm 1: Friday, February 24, in class, closed book and notes

Syntax Analysis (Scott 2.3 - 2.5 - skip 2.3.4)

Top-Down Parsing - LL(1)



Basic Idea:

- The parse tree is constructed from the root, expanding **non-terminal** nodes on the tree's frontier following a left-most derivation
- The input program is read from left to right, and input tokens are read (consumed) as the program is parsed
- The next **non-terminal** symbol (A in above figure) is replaced by one of its rules. The particular choice has to be unique, and uses parts of the input (partially parsed program), for instance the first **token** (here y[1]) of the remaining input

Predictive Parsing

Basic idea:

> For any two productions $A ::= \alpha \mid \beta$ 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 **FIRST**(α) as the set of tokens that appear as the first token in some string derived from α.

That is

$a \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* a\gamma$ for some $\gamma \in (T \cup N)^*$ and a is a token ($a \in T$), and

$\epsilon \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \epsilon$

For a non-terminal A, define **FOLLOW**(A) as the set

$a \in \text{FOLLOW}(A)$ *iff* $S \Rightarrow^* \alpha A \mathbf{a} \gamma$ for some $\alpha, \gamma \in (T \cup N)^*$, $a \in T$, and S the start symbol.

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

Predictive Parsing (cont.)

Key Property:

Whenever two productions $A ::= \alpha$ and $A ::= \beta$ both appear in the grammar, we would like

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$, and
- if $\alpha \Rightarrow^* \epsilon$ then $FIRST(\beta) \cap FOLLOW(A) = \emptyset$
- Analogue case for $\beta \Rightarrow^* \epsilon$. Note: due to first condition, at most one of α or β can derive ϵ.

This would allow the parser to make a correct choice with a lookahead of only one symbol!

LL(1) Grammar

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

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

A grammar is LL(1) iff

($A ::= \alpha$ and $A ::= \beta$) implies

$$FIRST^+(\alpha) \cap FIRST^+(\beta) = \emptyset$$

Our Example

S ::= a S b | ϵ

$FIRST$(aSb) = {a}

$FIRST(\epsilon) = \{\epsilon\}$

$FOLLOW$(S) = {eof, b}

$FIRST^+$(aSb) = {a}

$FIRST^+(\epsilon)$ = ($FIRST(\epsilon)$ - { ϵ}) $\cup$ FOLLOW(S) =
{eof, b}

Is the grammar LL(1)?

Table-Driven LL(1) Parser

LL(1) parse table

Example:

S ::= a S b | ϵ

	a	b	eof	other
S	aSb	ϵ	ϵ	error

How to parse input **a a a b b b** ?

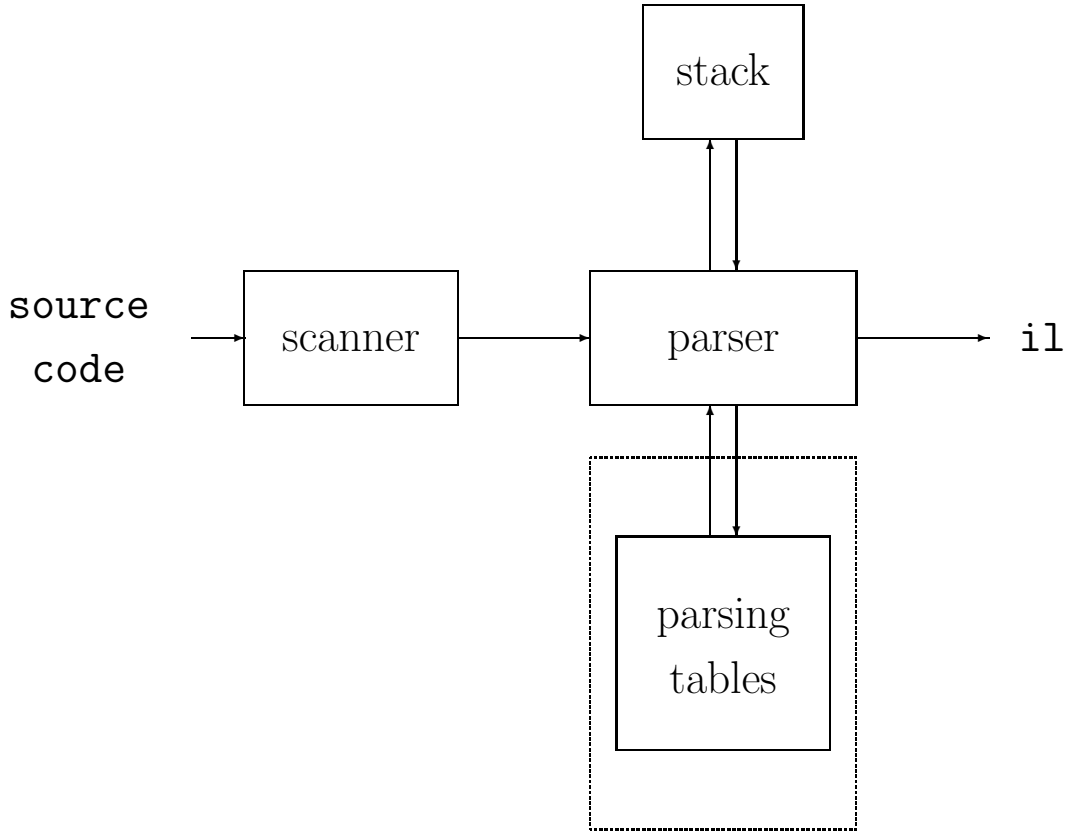
Table-driven predictive parsing algorithm

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

```
push 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

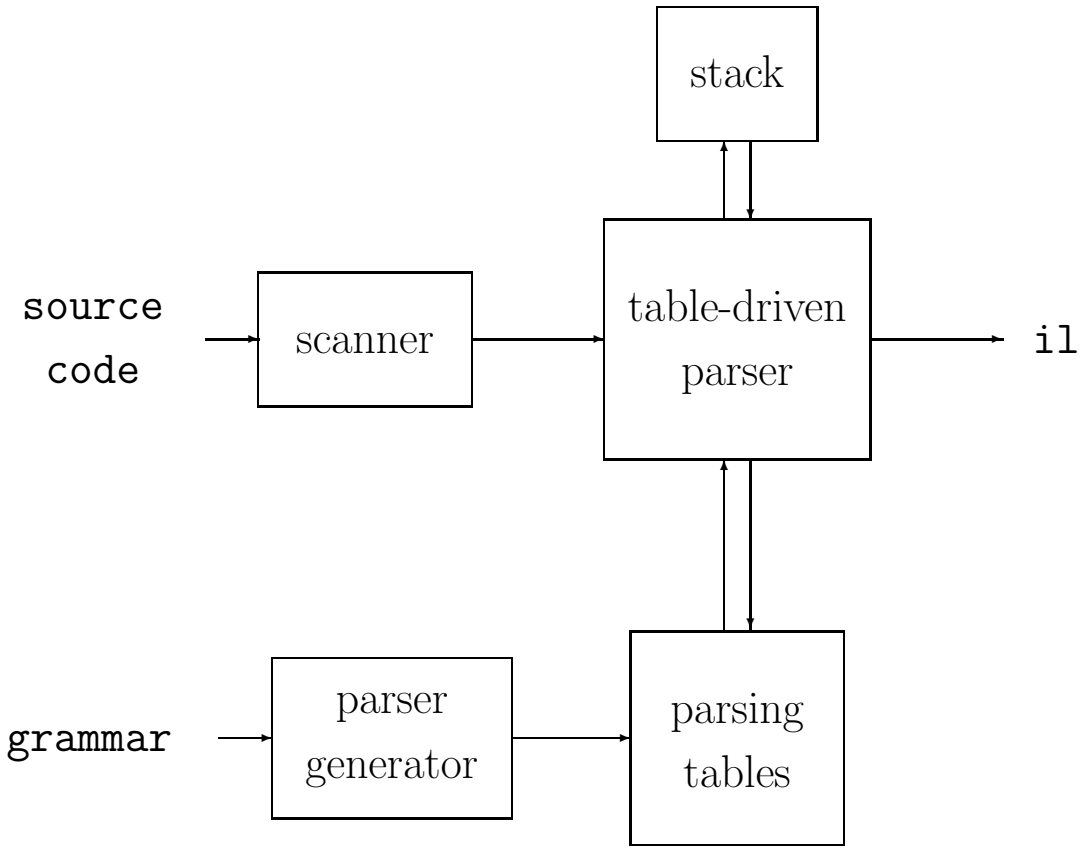
Now, a predictive parser looks like:



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** ?

Next Lecture

Things to do:

Start programming in C. Check out the web for tutorials.

Next time:

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