Top-Down Parsing
Announcements...

• Programming Assignment 2 due Thurs Sept 22.
• Midterm Exam #1 on Thursday Sept 29
  - In Class
  - ONE handwritten page (2 sides).
  - Your handwriting
  - No computer printouts, no calculators or cellphones
  - Bring a pencil
Review

• We can specify language syntax using CFG
• A parser will answer whether $\sigma \in L(G)$
• ... and will build a parse tree
• ... which is essentially an AST
• ... and pass on to the rest of the compiler

• Next few lectures:
  - How do we answer $\sigma \in L(G)$ and build a parse tree?
• After that: from AST to ... assembly language
Lecture Outline

- Implementation of parsers
- Two approaches
  - Top-down
  - Bottom-up
- Today: Top-Down
  - Easier to understand and program manually
- Next: Bottom-Up
  - More powerful and used by most parser generators
Intro to Top-Down Parsing

• Terminals are seen in order of appearance in the token stream:
  \[ t_2 \ t_5 \ t_6 \ t_8 \ t_9 \]

• The parse tree is constructed
  - From the top
  - From left to right
Recursive Descent Parsing

• Consider the grammar 3.10 in text.

\[
\begin{align*}
S & \rightarrow \text{if } E \text{ then } S \text{ else } S \\
S & \rightarrow \text{begin } S \ L \\
S & \rightarrow \text{print } E \\
L & \rightarrow \text{end} \\
L & \rightarrow ; \ S \ L \\
E & \rightarrow \text{num} = \text{num}
\end{align*}
\]
**Recursive Descent Parsing: Parsing S**

S \rightarrow \text{if } E \text{ then } S \text{ else } S
S \rightarrow \text{begin } S \text{ L}
S \rightarrow \text{print } E
L \rightarrow \text{end}
L \rightarrow ; S \text{ L}
E \rightarrow \text{num = num}

(defun s() (case (car tokens)
  (if (eat 'if)
    (e)
    (eat 'then)
    (s)
    (eat 'else)
    (s))
  (begin (eat 'begin) (s) (l))
  (print (eat 'print) (e))
  (otherwise (eat 'if ))))

; cheap error. can't match if!
Recursive Descent Parsing: Parsing L

S -> if E then S else S
S -> begin S L
S -> print E
L -> end
L -> ; S L
E -> num = num

(defun l() (case (car tokens)
  (end (eat 'end))
  (|;| (eat '|;|) (s)(l))
  (otherwise (eat 'end)))))
Recursive Descent Parsing: parsing E

S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num = num

(defun e() (eat 'num)
  (eat '=)
  (eat 'num))
Recursive Descent Parsing: utilities

Get-token = pop
Parse checks for empty token list.

(defun eat (h)
  (cond ((equal h (car tokens))
    (pop tokens)) ;; (pop x) means (setf x (cdr x))
    (t (error "stuck at ~s" tokens)))))

(defun parse (tokens)(s)
  (if (null tokens) "It is a sentence"))
Recursive Descent Parsing : tests

(defparameter test '(begin print num = num ;; if num = num
then print num = num else print num = num end))

(parse test) ⇒ “It is a sentence”
(parse '(if num then num)) ⇒ Error: stuck at
(then num)
This grammar is very easy. Why?

\[
\begin{align*}
S &\rightarrow \text{if } E \text{ then } S \text{ else } S \\
S &\rightarrow \text{begin } S \text{ L} \\
S &\rightarrow \text{print } E \\
L &\rightarrow \text{end} \\
L &\rightarrow ; \ S \ L \\
E &\rightarrow \text{num = num}
\end{align*}
\]

We can always tell from the first symbol which rule to use. if, begin, print, end, ;, num.
Recursive Descent Parsing, “backtracking”
Example 2

- Consider another grammar...
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]
- Token stream is: \( \text{int}_5 \ast \text{int}_2 \)
- Start with top-level non-terminal \( E \)
- Try the rules for \( E \) in order
Recursive Descent Parsing. Backtracking

- Try $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
  - But $(\text{does not match input token } \text{int}_5)$
- Try $T_1 \rightarrow \text{int}$. Token matches.
  - But $+\text{ after } T_1 \text{ does not match input token } *$
- Try $T_1 \rightarrow \text{int} * T_2$
  - This will match \text{int} but $+\text{ after } T_1 \text{ will be unmatched}$
- Parser has exhausted the choices for $T_1$
  - Backtrack to choice for $E_0$

$E \rightarrow T + E | T$
$T \rightarrow \text{int} | \text{int} * T | (E)$
\text{int}_5 * \text{int}_2
Recursive Descent Parsing. Backtracking

• Try $E_0 \rightarrow T_1$
• Follow same steps as before for $T_1$
  - And succeed with $T_1 \rightarrow \text{int} \ast T_2$ and $T_2 \rightarrow \text{int}$
  - With the following parse tree

```
  E_0
    |
  T_1
    |
   |
int_5  *  T_2
    |
  |
int_2
```
Recursive Descent Parsing (Backtracking)

- Do we have to backtrack?? Trick is to look ahead to find the first terminal symbol to figure out for sure which rule to try.
- Indeed backtracking is not needed, if the grammar is suitable. This grammar is suitable for prediction.
- Sometimes you can come up with a “better” grammar for the same exact language.
Lookahead makes backtracking unnecessary

(defun E()
  (T)
  (case (car tokens)
    (+ (eat '+) (E)) ; E -> T+E
    (otherwise nil))
)

(defun T() ;; Lookahead resolves rule choice
  (case (car tokens)
    (\( (eat \() (E) (eat \))\) ) ; T->(E)
    (int (eat 'int)
      (case (car tokens)
        (* (eat '*)(T)) ; T -> int * T
        (otherwise nil))
      (otherwise int))
    (otherwise (eat 'end)))

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S \alpha \mid \ldots$
  - suggests a program something like...
  - `(defun S() (S) (eat 'a))`
- $S()$ will get into an infinite loop

- A left-recursive grammar has a non-terminal $S$
  $$S \Rightarrow ^+ S\alpha$$ for some $\alpha$
- Recursive descent does not work in such cases
Elimination of Left Recursion

• Consider the left-recursive grammar
  \[ S \rightarrow S \alpha | \beta \]

• \( S \) generates all strings starting with a \( \beta \) and followed by a number of \( \alpha \) \([\alpha, \beta \text{ are strings of terminals, in these examples.}]

• Can rewrite using right-recursion
  \[ S \rightarrow \beta S' \]
  \[ S' \rightarrow \alpha S' | \varepsilon \]
More Elimination of Left-Recursion

• In general

\[ S \rightarrow S \alpha_1 | ... | S \alpha_n | \beta_1 | ... | \beta_m \]

• All strings derived from \( S \) start with one of \( \beta_1, ..., \beta_m \) and continue with several instances of \( \alpha_1, ..., \alpha_n \)

• Rewrite as

\[
S \rightarrow \beta_1 S' | ... | \beta_m S' \\
S' \rightarrow \alpha_1 S' | ... | \alpha_n S' | \varepsilon
\]
General Left Recursion

- The grammar
  \[ S \rightarrow A \alpha | \delta \]
  \[ A \rightarrow S \beta \]
  is also left-recursive (even without a left-recursive RULE) because
  \[ S \Rightarrow^+ S \beta \alpha \]

- This left-recursion can also be eliminated
Summary of Recursive Descent

• Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - ... but that can be done automatically
• Not so popular because common parser-generator tools allow more freedom in making up grammars.
• (False) reputation of inefficiency
• If hand-written, powerful error correction and considerable flexibility.
• Sometimes Rec Des is used for lexical analysis. Balanced comment delimiters /*/..*/../..*/, e.g.
• In practice, backtracking does not happen ever.
Predictive Parsers: generalizing lookahead

- Like recursive-descent but parser can “predict” which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means “left-to-right” scan of input
  - L means “leftmost derivation”
  - k means “predict based on k tokens of lookahead”
- In practice, LL(1) is used
LL(1) Languages

• In recursive-descent, for each non-terminal and input token there may be a choice of production
• LL(1) means that for each non-terminal and token there is only one production
• Can be specified via 2D tables
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production
Predictive Parsing and Left Factoring

• Recall the grammar
  
  \[ E \rightarrow T + E \mid T \]
  
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

• Hard to predict because
  
  – For \( T \) two productions start with \( \text{int} \)
  
  – For \( E \) it is not clear how to predict

• A grammar must be left-factored before use for predictive parsing
Left-Factoring Example

• Recall the grammar
  \[
  E \rightarrow T + E \mid T \\
  T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)
  \]

• Factor out common prefixes of productions
  \[
  E \rightarrow T X \\
  X \rightarrow + E \mid \varepsilon \\
  T \rightarrow (E) \mid \text{int} Y \\
  Y \rightarrow \ast T \mid \varepsilon
  \]
LL(1) Parsing Table Example

• Left-factored grammar
  
  \[
  E \rightarrow T X \\
  T \rightarrow ( E ) | \text{int } Y \\
  X \rightarrow + E | \epsilon \\
  Y \rightarrow * T | \epsilon 
  \]

• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>T</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>+ E</td>
<td></td>
<td></td>
<td>\epsilon</td>
<td>\epsilon</td>
</tr>
<tr>
<td>T</td>
<td>int</td>
<td>Y</td>
<td></td>
<td></td>
<td>( E )</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>* T</td>
<td></td>
<td>\epsilon</td>
<td></td>
<td>\epsilon</td>
<td>\epsilon</td>
</tr>
</tbody>
</table>
LL(1) Parsing Table Example (Cont.)

- **Consider the [E, int] entry**
  - “When current non-terminal is E and next input is int, use production $E \rightarrow T X$
  - This production can generate an int in the first place

- **Consider the [Y, +] entry**
  - “When current non-terminal is Y and current token is +, get rid of Y”
  - Y can be followed by + only in a derivation in which $Y \not{\epsilon} \epsilon$
LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
  - Consider the $[E,\ast]$ entry
  - “There is no way to derive a string starting with $\ast$ from non-terminal $E$”
Using Parsing Tables

• Method similar to recursive descent, except
  - For each non-terminal X
  - We look at the next token a
  - And chose the production shown at \([X, a]\)
• We use a stack to keep track of pending non-terminals
• We reject when we encounter an error state
• We accept when we encounter end-of-input
LL(1) Parsing Algorithm

initialize stack = <S $> and next
repeat
  case stack of
    <X, rest> : if T[X,nextinput] = Y_1...Y_n
                 then stack ← <Y_1... Y_n ,rest>;
                 else error ();
    <t, rest>  : if t = nextinput
                 then stack ← <rest>;
                 else error ();

  until stack is empty
## LL(1) Parsing Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>E $</td>
<td>int * int $</td>
<td>T X</td>
</tr>
<tr>
<td>T X $</td>
<td>int * int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int * int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>* T</td>
</tr>
<tr>
<td>* T X $</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>$</td>
<td>ε</td>
</tr>
<tr>
<td>X $</td>
<td>$</td>
<td>ε</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>

Prof. Fateman  CS 164  Lecture 8
Constructing Parsing Tables

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined
- We want to generate parsing tables from CFG
Constructing Parsing Tables (Cont.)

• If $A \rightarrow \alpha$, where in the line $A$ do we place $\alpha$?
• In the column of $\dagger$ where $\dagger$ can start a string derived from $\alpha$
  - $\alpha \emptyset^* \dagger \beta$
  - We say that $\dagger \in \text{First}(\alpha)$
• In the column of $\dagger$ if $\alpha$ is $\varepsilon$ and $\dagger$ can follow an $A$
  - $S \emptyset^* \beta A \dagger \delta$
  - We say $\dagger \in \text{Follow}(A)$
Computing First Sets

Definition
\[
\text{First}(X) = \{ t \mid X \to^* t\alpha \} \cup \{ \varepsilon \mid X \to^* \varepsilon \}
\]

Algorithm sketch:
1. First(t) = \{ t \}
2. \( \varepsilon \in \text{First}(X) \) if X \( \to \varepsilon \) is a production
3. \( \varepsilon \in \text{First}(X) \) if X \( \to A_1 \ldots A_n \)
   - and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
4. First(\( \alpha \)) \( \subseteq \) First(X) if X \( \to A_1 \ldots A_n \alpha \)
   - and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
First Sets. Example

• Recall the grammar

\[ E \rightarrow T \ X \]
\[ T \rightarrow ( \ E ) \mid \text{int} \ Y \]
\[ X \rightarrow + \ E \mid \epsilon \]
\[ Y \rightarrow * \ T \mid \epsilon \]

• First sets

\[ \text{First}( \ () = \{ ( ) \} \]
\[ \text{First}( \ ) = \{ ) \} \]
\[ \text{First}( \text{int}) = \{ \text{int} \} \]
\[ \text{First}( + ) = \{ + \} \]
\[ \text{First}( * ) = \{ * \} \]
\[ \text{First}( \ T ) = \{ \text{int}, ( ) \} \]
\[ \text{First}( \ E ) = \{ \text{int}, ( ) \} \]
\[ \text{First}( \ X ) = \{ +, \epsilon \} \]
\[ \text{First}( \ Y ) = \{ *, \epsilon \} \]
Computing First Sets by Computer

- Recall the grammar
  \[ E \rightarrow T \times \]
  \[ T \rightarrow (E) \mid \text{int} \ Y \]
  \[ X \rightarrow + E \mid \varepsilon \]
  \[ Y \rightarrow * T \mid \varepsilon \]

- First sets
  \[ \text{First}(\ )) = \{ ( ) \} \]
  \[ \text{First}(\ ) ) = \{ ( ) \} \]
  \[ \text{First}(\text{int}) = \{ \text{int} \} \]
  \[ \text{First}(\ + ) = \{ + \} \]
  \[ \text{First}(\ * ) = \{ * \} \]
  \[ \text{First}(\ T ) = \{ \text{int}, ( ) \} \]
  \[ \text{First}(\ E ) = \{ \text{int}, ( ) \} \]
  \[ \text{First}(\ X ) = \{ +, \varepsilon \} \]
  \[ \text{First}(\ Y ) = \{ *, \varepsilon \} \]
Computing Follow Sets

• Definition:
  \[
  \text{Follow}(X) = \{ \, \triangledown \mid S \rightarrow^* \beta \ X \, \triangledown \delta \, \}
  \]

• Intuition
  - If \( X \triangledown A \ B \) then \( \text{First}(B) \subseteq \text{Follow}(A) \) and \( \text{Follow}(X) \subseteq \text{Follow}(B) \)
  - Also if \( B \triangledown^* \varepsilon \) then \( \text{Follow}(X) \subseteq \text{Follow}(A) \)
  - If \( S \) is the start symbol then \( \$ \in \text{Follow}(S) \)
Computing Follow Sets (Cont.)

Algorithm sketch:
1. \( \$ \in \text{Follow}(S) \)
2. \( \text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X) \)
   - For each production \( A \rightarrow \alpha X \beta \)
3. \( \text{Follow}(A) \subseteq \text{Follow}(X) \)
   - For each production \( A \rightarrow \alpha X \beta \) where \( \varepsilon \in \text{First}(\beta) \)
Follow Sets. Example

• Recall the grammar
  \[ E \rightarrow T X \quad X \rightarrow + E | \epsilon \]
  \[ T \rightarrow ( E ) | \text{int} Y \quad Y \rightarrow * T | \epsilon \]

• Follow sets
  \[
  \text{Follow}(+ ) = \{ \text{int}, ( ) \} \quad \text{Follow}(\ast ) = \{ \text{int}, ( ) \}
  \text{Follow}( ( ) = \{ \text{int}, ( ) \} \quad \text{Follow}( E ) = \{ ), \$ \}
  \text{Follow}( X ) = \{ $, , \} \quad \text{Follow}( T ) = \{ +, ), \$ \}
  \text{Follow}( ) = \{ +, ), \$ \} \quad \text{Follow}( Y ) = \{ +, ), \$ \}
  \text{Follow}( \text{int}) = \{ *, +, ), \$ \} \]
Constructing LL(1) Parsing Tables

• Construct a parsing table $T$ for CFG $G$

• For each production $A \rightarrow \alpha$ in $G$ do:
  - For each terminal $t \in \text{First}(\alpha)$ do
    • $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    • $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    • $T[A, \$] = \alpha$
Computing with Grammars. Step One: Representing a grammar in Lisp.

(defparameter lect8 ;; here’s one way
  '((E -> T X)
    (T -> \( E \))
    (T -> int Y)
    (X -> + T)
    (X -> )
    (Y -> * T)
    (Y -> )))
Computing some useful information

(defun rhs(r) (cddr r)) ;; e.g. r is (E -> T + E)
(defun lhs(r) (first r))

(defun non-terminals(g) (remove-duplicates (mapcar #'lhs g)))

(defun terminals(g)
    (set-difference (reduce #'union (mapcar #'rhs g))
        (non-terminals g) ))

Prof. Fateman  CS 164  Lecture 8
Representing sets

(defmacro First (x) ;x is a symbol
  `(gethash ,x First))

(defmacro Follow(x) ;x is a symbol
  `(gethash ,x Follow))

(defmacro addinto(place stuff)
  `(setf ,place (union ,place ,stuff)))

;; alternatively, if we have just one set, like
;; which symbols are nullable, we might just
;; assign (setf nullable ‘())
;; and (push ‘x nullable) ;; to insert x into that set...
;; same as (setf nullable (cons ‘x nullable))
;; you know this from your lexical analysis program, though..
Compute Nullable set

;;;; Compute nullable set of a grammar. The non-terminal symbol X is
;;;; nullable if X can derive an empty string, X => .. => .. => empty.
;;;; Given
;;;; grammar g, return a lisp list of symbols that are nullable.
(defun nullables (g)
  (let ((nullable nil)
        (changed? t))
    (while changed?
      (setf changed? nil)
      (dolist (r g) ; for each rule
        (cond
         ;; if X is already nullable, do nothing.
         ((member (lhs r) nullable) nil)
         ;; for each rule (X -> A B C ),
         ;; X is nullable if every one of A, B, C is nullable
         ((every #'(lambda (z) (member z nullable))(rhs r))
          (push (lhs r) nullable)
          (setf changed? t)))))
    (sort nullable #'string<)))) ; sort to make it look nice

See firstfoll.cl for details
```lisp
(defun firstset(g) ;; g is a list of grammar rules
  (let ((First (make-hash-table)) ;; First is a hashtable, in addition
to a relation First[x]
    (nullable (nullablesset g))
    (changed? t))
  ;; for each terminal symbol j, First[j] = {j}
  (dolist (j (terminals g))
    (setf (First j) (list j)))
  (while changed?
    (setf changed? nil)
    (dolist (r g)
      ;; for each rule in the grammar  X -> A B C
      ;; see next slide...
      ;; did this First set or any other First set
      ;; change in this run?
      (setf changed? (or changed? (< setsize (length (First X)))))))
  ; exit from loop
  First ))
```

See firstfoll.cl for details
(defun firstset(g) ;; g is a list of grammar rules
  (let ((First (make-hash-table)) ;; First is a hashtable, in addition to a
    (nullable (nullablesset g))
    (changed? t))
    ;; for each terminal symbol j, First[j] = {j}
    (dolist (j (terminals g))
      (setf (First j) (list j)))
    (while changed?
      (setf changed? nil)
      (dolist (r g)
        ;; for each rule in the grammar  X -> A B C
        (let* ((X (lhs r))
          (RHS (rhs r))
          (setsize (length (First X))))
          ;; First[X]= First[X] U First[A]
          (cond ((null RHS) nil)
            (t (addinto (First X) (First (car RHS))))
            (while (member (car RHS) nullable)
              (pop RHS)
              (addinto (First X) (First (car RHS)))
            )))
        ;; did this First set or any other First set
        ;; change in this run?
        (setf changed? (or changed? (< setsize (length (First X)))))
      )); exit from loop
    First))
((defun followset(g);; g is a list of grammar rules
 (let ((First (firstset g))
 (Follow (make-hash-table))
 (nullable (nullables set g))
 (changed? t))
 (while
 changed?
 (setf changed? nil)
 (dolist (r g)
 ;; for each rule in the grammar  X -> A B C D
 ;; (format t "-%s rule is ~s" r)
 (do ((RHS (rhs r)(cdr RHS)))
 ;; test to end the do loop
 ((null RHS) 'done )
 ;; let RHS be, in succession,
 ;; (A B C D)
 ;; (B C D)
 ;; (C D)
 ;; (D)
 (if (null RHS) nil ;; no change in follow set for erasing rule
 (let* ((A (car RHS))
 (Blist (cdr RHS)) ; e.g. (B C D)
 (Asize (length (Follow A))))
 (if(every #'(lambda(z)(member z nullable)) Blist)
 ;; X -> A <nullable> ... then anything
 ....more
 (See firstfoll.cl for details)
((defun followset(g);; g is a list of grammar rules
    ;; ; ; ; . . .
    (if(every #'(lambda(z)(member z nullable)) Blist)
        ;; X -> A <nullable> ... then anything
        ;; following X can follow A:
        ;; Follow[A] = Follow[A] U Follow[X]
        (addinto (Follow A)(Follow (lhs r)))))
    (if Blist
        ;; not empty
        ;; Follow[A]= Follow[A] U First[B]
        (addinto (Follow A)(First (car Blist))))
    (while (and Blist (member (car Blist) nullable))
        ;; false when Blist =()
        ;; if X -> A B C and B is nullable, then
        ;; Follow[A]=Follow[A] U First(C)
        (pop Blist)
        (addinto (Follow A)(First (car Blist)))
    (setf changed? (or changed? (< Asize (length (Follow A)))))))

    ;; Remove the terminal symbols in Follow table
    ;; are uninteresting
    ;; Return the hashtable "Follow" which has pairs like <X (a b)>.
    (mapc #'(lambda(v)(remhash v Follow)) (terminals g))
    ;; (printfols Follow) ; print the table for human consumption
    Follow ; for further processing
))
# Predictive parsing table

(pptab lect8)

<table>
<thead>
<tr>
<th>First Sets symbol</th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>E</td>
<td>( int</td>
</tr>
<tr>
<td>T</td>
<td>( int</td>
</tr>
<tr>
<td>X</td>
<td>+</td>
</tr>
<tr>
<td>Y</td>
<td>*</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Follow Sets symbol</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td>T</td>
<td>) +</td>
</tr>
<tr>
<td>X</td>
<td>)</td>
</tr>
<tr>
<td>Y</td>
<td>) +</td>
</tr>
</tbody>
</table>
Predictive parsing table

(\texttt{ht2grid(pptab lect8)})
rows = (E T X Y), cols= (|(| |)| * + int)

\begin{tabular}{|c|c|c|c|c|}
\hline
 & (    ) & * & + & \text{int} \\
\hline
E & \text{E -> T X} & | & | & | \text{E -> T X} \\
T & \text{T -> ( E )} & | & | & | \text{T -> int Y} \\
X & | & \text{X ->} & | & \text{X -> + T} \ |
Y & | & \text{Y ->} & \text{Y -> * T} & \text{Y} \\
\hline
\end{tabular}
Notes on LL(1) Parsing Tables

• If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - And in other cases as well

• Most programming language grammars are not LL(1), but could be made so with a little effort.

• Firstfoll.cl builds an LL(1) parser. About 140 lines of Lisp code. (With comments, debugging code, test data, the file is about 550 lines)
Review

• For some grammars / languages there is a simple parsing strategy based on recursive descent. It even can be automated: Predictive parsing

• Next: a more powerful parsing strategy