Top-Down Parsing

CS164

Lecture 8

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

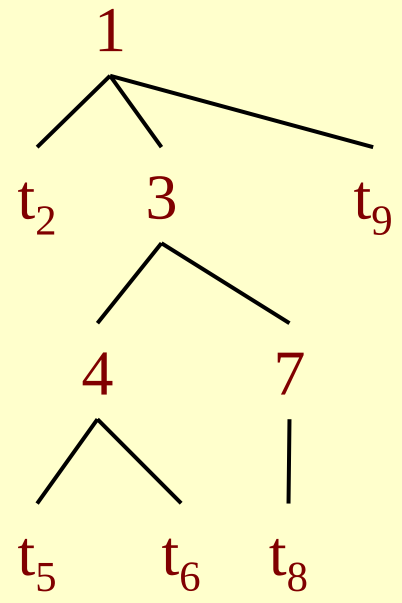
- Implementation of parsers
- Two approaches
  - Top-down
  - Bottom-up
- Today: Top-Down
  - Easier to understand and program manually
- Next: Bottom-Up
  - More powerful and used by most parser generators

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..

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

Recursive Descent Parsing: Parsing S

S-> if E then S else S

S -> begin S L

S -> print E

L -> end

L -> ; S L

E -> 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?

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

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…

  $E \to T + E \mid T$

  $T \to \text{int} \mid \text{int} * T \mid (E)$

- Token stream is: $\text{int}_5 * \text{int}_2$
- Start with top-level non-terminal E

- Try the rules for E in order

Recursive Descent Parsing. Backtracking

$E \to T + E \mid T$

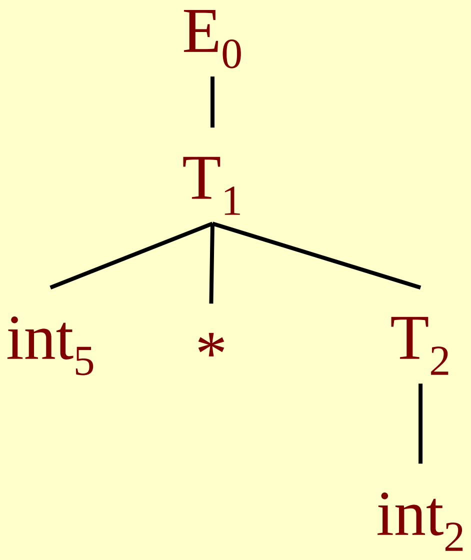
$T \to \text{int} \mid \text{int} * T \mid (E)$

$\text{int}_5 * \text{int}_2$

- Try $E_0 \to T_1 + E_2$
- Then try a rule for $T_1 \to (E_3)$
  - But (does not match input token int_5
- Try $T_1 \to \text{int}$. Token matches.
  - But + after T_1 does not match input token *
- Try $T_1 \to \text{int} * T_2$
  - This will match int but + after T_1 will be unmatched
- Parser has exhausted the choices for T_1
  - Backtrack to choice for E_0

Recursive Descent Parsing. Backtracking

- Try $E_0 \to T_1$
- Follow same steps as before for T_1
  - And succeed with $T_1 \to$ int * T_2 and $T_2 \to$ int
  - With the following parse tree



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, <u>if the grammar is suitable.</u> 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)                     ; T -> int | int*T
       (case (car tokens)               ; look beyond int
          (* (eat '*)(T))               ; T -> int * T
          (otherwise nil)))             ; T -> int

    (otherwise (eat 'end))))
```

When Recursive Descent Does Not Work

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

- A <u>left-recursive grammar</u> has a non-terminal S

  $S \Rightarrow^{+} S\alpha$ for some α
- Recursive descent does not work in such cases

Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \to S\ \alpha \mid \beta$$

- S generates all strings starting with a β and followed by a number of α **[α, β are strings of terminals, in these examples.]**

- Can rewrite using <u>right</u>-recursion

$$S \to \beta\ S'$$

$$S' \to \alpha\ S' \mid \varepsilon$$

More Elimination of Left-Recursion

- In general

$$S \to S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1, \ldots, \beta_m$ and continue with several instances of $\alpha_1, \ldots, \alpha_n$
- Rewrite as

$$S \to \beta_1\ S' \mid \ldots \mid \beta_m\ S'$$
$$S' \to \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$$

General Left Recursion

- The grammar

  $S \to A\ \alpha \mid \delta$

  $A \to S\ \beta$

  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

  $E \rightarrow T + E \mid T$

  $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

- Hard to predict because
  - For T two productions start with 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

$E \to T + E \mid T$

$T \to \text{int} \mid \text{int} * T \mid (E)$

- Factor out common prefixes of productions

$E \to T\, X$

$X \to + E \mid \varepsilon$

$T \to (E) \mid \text{int}\ Y$

$Y \to * T \mid \varepsilon$

LL(1) Parsing Table Example

- Left-factored grammar

  $E \rightarrow T\ X$ $\qquad\qquad$ $X \rightarrow +\ E \mid \varepsilon$

  $T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- The LL(1) parsing table:

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

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 ❾ ε

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
  - Consider the [E,*] entry
  - "There is no way to derive a string starting with * 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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

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 α ?
- In the column of t where t can start a string derived from α
  - α❾* t β
  - We say that t $\in$ First(α)
- In the column of t if α is ε and t can follow an A
  - S ❾* β A t δ
  - We say t $\in$ Follow(A)

Computing First Sets

Definition

$$\text{First}(X) = \{\, t \mid X \to^* t\alpha\} \cup \{\varepsilon \mid X \to^* \varepsilon\}$$

Algorithm sketch:

1. $\text{First}(t) = \{\, t \,\}$
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 $1 \le i \le n$
4. $\text{First}(\alpha) \subseteq \text{First}(X)$ if $X \to A_1 \ldots A_n\, \alpha$
   - and $\varepsilon \in \text{First}(A_i)$ for $1 \le i \le n$

First Sets. Example

- Recall the grammar

$E \to T\ X$	$X \to +\ E \mid \varepsilon$
$T \to (\ E\ ) \mid \text{int}\ Y$	$Y \to *\ T \mid \varepsilon$

- First sets

First(( ) = { (}	First(T) = {int, (}
First( )) = {) }	First(E) = {int, (}
First(int) = { int }	First(X) = {+, ε }
First(+) = { + }	First(Y) = {*, ε }
First(*) = { * }	

Computing First Sets by Computer

- Recall the grammar

$E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- First sets

First(( ) = { (} $\qquad$ First(T) = {int, (}

First( )) = {) } $\qquad$ First(E) = {int, (}

First(int) = { int } $\qquad$ First(X) = {+, ε }

First(+) = { + } $\qquad$ First(Y) = {*, ε }

First(*) = { * }

Computing Follow Sets

- Definition:

  Follow(X) = { t | S $\to^*$ β X t δ }

- Intuition
  - If X ❾ A B then First(B) ⊆ Follow(A) and Follow(X) ⊆ Follow(B)
  - Also if B ❾* ε then Follow(X) ⊆ Follow(A)
  - If S is the start symbol then $ ∈ 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

$E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- Follow sets

Follow(+) = { int, (}	Follow(*) = { int, (}
Follow(( ) = { int, (}	Follow(E) = {), $}
Follow(X) = {$,) }	Follow(T) = {+,) , $}
Follow( )) = {+ ,), $}	Follow(Y) = {+,) , $}
Follow(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 $t \in$ First(α) do
    - T[A, t] = α
  - If $\varepsilon \in$ First(α), for each $t \in$ Follow(A) do
    - T[A, t] = α
  - If $\varepsilon \in$ First(α) and $\$ \in$ Follow(A) do
    - T[A, \$] = α

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) ))
```

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

See firstfoll.cl for details

```
 ;; 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 nullableset(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
```

Compute Firstset

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 relation First[x]
        (nullable (nullableset 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    ))
```

```
 (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 (nullableset 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))
                   ))
                                              ;end of inner while
           ;; 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    ))
```

Followset in Lisp

See firstfoll.cl for details

```
((defun followset(g);; g is a list of grammar rules
 (let ((First (firstset g))
         (Follow (make-hash-table))
         (nullable (nullableset g))
         (changed? t))
   (while
    changed?
    (setf changed? nil)
     (dolist (r g)
         ;; for each rule in the grammar  X -> A B C D
         ;;(format t "~%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
```

Followset in Lisp, continued

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)

First Sets
symbol     First
------------------
(          (
)          )
*          *
+          +
E          ( int
T          ( int
X          +
Y          *
int        int
```

```
Follow Sets
symbol     Follow
------------------
E          )
T          ) +
X          )
Y          ) +
```

Predictive parsing table

```
(ht2grid(pptab lect8))
rows = (E T X Y), cols= (|(| |)| * + int)

                  (             )             *             +             int
------------------------------
E              |E -> T X     |             |             |             |E -> T X
T              |T -> ( E )   |             |             |             |T -> int Y
X              |             |X ->         |             |X -> + T     |
Y              |             |Y ->         |Y -> * T     |Y
```

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