

Lecture 9-10 — Top-down parsing

- Top-down (recursive descent) parsing
 - Simple examples
 - FIRST sets
 - Handling ϵ -productions; FOLLOW sets
 - Producing ASTs
 - Expression grammars

Top-down recognition

Top-down, or *recursive descent*, parsing is a parsing method where the structure of the parser directly follows the structure of the cfg.

A $\rightarrow$ **id** | **'(' B ')'**
B $\rightarrow$ **int** | **A**

```
exception SyntaxError
type token = IDENT of string | LPAREN | RPAREN | INT of int | EOF

let rec parseA toklis = match (hd toklis) with
  IDENT x -> tl toklist
  | LPAREN -> let r = parseB (tl toklis)
               in (if hd r = RPAREN then tl r
                   else raise SyntaxError)
  | _ -> raise SyntaxError

and parseB toklis = match (hd toklis) with
  INT i -> tl toklis | _ -> parseA toklis

let parse toklis = (parseA toklis = [EOF])
```

Ex: Top-down parsing

- Write a recursive descent recognizer for this grammar:

$A \rightarrow \text{int} \mid '(' B ')'$
 $B \rightarrow '+' A B \mid \text{int}$

```
type token = INT of int | LPAREN | RPAREN | PLUS | EOF
```

```
let rec parseA toklis = match (hd toklis) with
```

```
and parseB toklis =
```

```
and parse toklis = (parseA toklis = [EOF])
```

Top-down parsing, formally

- **Basic idea:** Define a parsing function parse_A for each $A \in N$. In parse_A , choose a production, then go through it one symbol at a time, consuming tokens and recursively calling parse functions for non-terminals.

- **Formally:** For each non-terminal A , define function:

$\text{parse}_A \text{ toklis} =$ based on hd toklis , choose production:

if $A \rightarrow X_1 X_2 \dots X_n$ chosen: $\text{parse}_{X_n}(\dots (\text{parse}_{X_2} (\text{parse}_{X_1} \text{ toklis})) \dots)$

else if $A \rightarrow Y_1 Y_2 \dots$ chosen: *etc.*

where parse_X is the parsing function for X if $X \in N$. (Note that if $n = 0$, this just returns toklis .) For $X = t \in T$:

$\text{parset toklis} =$ if $\text{hd toklis} = t$ then tl toklis else error

Top-down recognizer exercise

$L \rightarrow E R$

$E \rightarrow \text{id}$

$R \rightarrow ; L \mid \epsilon$

```
type token = IDENT of string | SEMIC | EOF  
let rec parseL toklis =
```

```
and parseE toklis =
```

```
and parseR toklis =
```

```
and parse toklis = (parseL toklis = [EOF])
```

A couple of new definitions...

- If $t \in T$, we will define the set of t -forms, or t -sentences, to be just $\{t\}$.
- If $\alpha = X_1X_2 \dots X_n \in S^*$, an α -form is any string $\beta_1\beta_2 \dots \beta_n \in S^*$ such that, for each i , β_i is an X_i -form. An α -sentence is an α -form in T^* .

“choose production based on hd toklis”

- Choosing the right production is the heart of the parsing problem when using this method: Based on just the next input token, there must be a *unique* production that could work. Use the idea of “FIRST sets”:
 - For $\alpha \in S^*$, $\text{FIRST}(\alpha)$ is the set of all terminal symbols that can be the first symbol in an α -sentence.
 - To choose the production for `parseA`, find the *unique* production $A \rightarrow \alpha$ such that `hd toklis` $\in \text{FIRST}(\alpha)$.
 - If the production is not unique — you can’t parse this grammar by recursive descent!

FIRST set examples

$A \rightarrow \text{id} \mid ' (' B ')'$

$B \rightarrow \text{int} \mid A$

$\text{FIRST}(A) = \{ \quad \}$

$\text{FIRST}(B) = \{ \quad \}$

$A \rightarrow \text{int} \mid ' (' B ')'$

$B \rightarrow A '+' B$

$\text{FIRST}(A) = \{ \quad \}$

$\text{FIRST}(B) = \{ \quad \}$

Calculating FIRST sets

- Assume, for now, no ϵ -productions.
- $FIRST(G)$ returns a table mapping N to $\mathcal{P}(T)$. The entry for A is written as $FIRST(G)[A]$.

```
 $FIRST(G) =$   
   $fsts_0 =$  empty table (i.e. maps every  $A \in N$  to  $\{\}$ )  
   $i = 0$   
  repeat {  $i = i+1$ ;  $fsts_i =$  empty table  
    for every production  $A \rightarrow \alpha$  in  $G$ :  
       $fsts_i(A) = fsts_i(A) \cup RHSFirst(\alpha, fsts_{i-1})$   
    } until  $fsts_i = fsts_{i-1}$   
  return  $fsts_i$ 
```

```
 $RHSFirst(X\alpha, fsts) =$  if  $X \in T$  return {  $X$  }  
                        else return  $fsts[X]$ 
```

- Define $FIRST(\alpha) = RHSFirst(\alpha, FIRST(G))$.

FIRST sets calculation

$$\begin{aligned} E &\rightarrow T \mid T + E \\ T &\rightarrow P \mid P * T \\ P &\rightarrow \mathbf{id} \mid (E) \end{aligned}$$

fst_0 :

E	
T	
P	

fst_1 :

E	
T	
P	

fst_2 :

E	
T	
P	

fst_3 :

E	
T	
P	

fst_4 :

E	
T	
P	

Ex: FIRST sets calculation

$$\begin{aligned} E &\rightarrow T \mid E + T \\ T &\rightarrow P \mid T * P \\ P &\rightarrow \mathbf{id} \mid (E) \end{aligned}$$

fst_0 :

E	
T	
P	

fst_1 :

E	
T	
P	

fst_2 :

E	
T	
P	

fst_3 :

E	
T	
P	

fst_4 :

E	
T	
P	

“choose production based on hd toklis” (cont.)

- We can now be more precise about the definition of `parseA`:

`parseA toklis = choose the unique  $A \rightarrow \alpha$  s.t.  $\text{hd toklis} \in FIRST(\alpha)$ :`

`if  $A \rightarrow X_1 X_2 \dots X_n$  chosen: parse $X_n$ (... (parse $X_2$  (parse $X_1$  toklis))...)`

`else if  $A \rightarrow Y_1 Y_2 \dots$  chosen: etc.`

- If there is no production for which $\text{hd toklis} \in FIRST(\alpha)$, then it is a syntax error.
- If the production cannot be chosen uniquely, the grammar cannot be parsed by this method. (What about the grammar on the last slide?)

Dealing with ϵ -productions

- We add the symbol “•” to *FIRST* sets to indicate that the set contains the empty string (ϵ). That is, $\bullet \in FIRST[A]$ indicates that A is nullable.
- The definition of *FIRST* above still works, but we need to change *RHSFirst*:

$$\begin{aligned} RHSFirst(X_1 X_2 \dots X_n, fsts) = \\ \text{if } n=0 \text{ return } \{\bullet\} \\ \text{else if } X_1 \in T \text{ return } \{ X_1 \} \\ \text{else if } \bullet \notin fsts[X_1] \text{ return } fsts[X_1] \\ \text{else return } (fsts[X_1] - \{\bullet\}) \cup RHSFirst(X_2 \dots X_n, fsts) \end{aligned}$$

- Again, $FIRST(\alpha) = RHSFirst(\alpha, FIRST(G))$.

Ex: FIRST sets calculation

$E \rightarrow T E'$
 $E' \rightarrow \epsilon \mid + E$
 $T \rightarrow P T'$
 $T' \rightarrow \epsilon \mid * T$
 $P \rightarrow \mathbf{id} \mid (E)$

$firsts_0:$

E	
E'	
T	
T'	
P	

$firsts_1:$

E	
E'	
T	
T'	
P	

$firsts_2:$

E	
E'	
T	
T'	
P	

$firsts_3:$

E	
E'	
T	
T'	
P	

$firsts_4:$

E	
E'	
T	
T'	
P	

Ex: FIRST sets calculation

$S \rightarrow A \mathbf{b}$

$A \rightarrow \mathbf{a} \mid B \mid \epsilon$

$B \rightarrow \mathbf{b} \mid \epsilon$

S	
A	
B	

$fst s_1:$

s	
A	
B	

$fst s_2:$

S	
A	
B	

$fst s_3:$

S	
A	
B	

FOLLOW sets

- When ϵ -productions are present, there is another problem:
 - Consider the grammar:
$$A \rightarrow B \text{ int}$$
$$B \rightarrow B \text{ id} \mid \epsilon$$
and suppose the input is 5.
 - $5 \in FIRST(B \text{ int})$, so the first (and only) production from A is chosen, and we call `parseB`. However, $5 \notin FIRST(B \text{ id})$ or $FIRST(\epsilon)$, so no production for B can be chosen, and the input is rejected!
 - Should instead realize that 5 can *follow* an occurrence of B , and since B is nullable, that is good enough.

FOLLOW sets (cont.)

- **$FOLLOW(G)$ returns a table that gives, for ever $A \in N$, the set of tokens that can immediately follow A in any sentential form. ($FOLLOW(G)[A]$ can contain *eof*, but not $\bullet$.)**

$FOLLOW(G) =$
 $flws_0 =$ table mapping S to $\{ \text{eof} \}$, and every other $A \in N$ to $\{\}$
 $i = 0$
 repeat {
 $i = i+1$; $flws_i = flws_0$
 for every $B \in N$:
 for every occurrence of B in a production $A \rightarrow \alpha B \beta$:
 $flws_i[B] = flws_i[B] \cup (FIRST(\beta) - \{\bullet\})$
 if $\bullet \in FIRST(\beta)$ then $flws_i[B] = flws_i[B] \cup flws_{i-1}[A]$
 } until $flws_i = flws_{i-1}$
 return $flws_i$

Ex: FOLLOW sets calculation

$E \rightarrow T E'$
 $E' \rightarrow \epsilon \mid + E$
 $T \rightarrow P T'$
 $T' \rightarrow \epsilon \mid * T$
 $P \rightarrow \text{id} \mid (E)$

FIRST:

E	id, (
E'	•, +
T	id, (
T'	•, +
P	id, (

$flws_0$:

E	
E'	
T	
T'	
P	

$flws_1$:

E	
E'	
T	
T'	
P	

$flws_2$:

E	
E'	
T	
T'	
P	

$flws_3$:

E	
E'	
T	
T'	
P	

Ex: FOLLOW sets calculation

$S \rightarrow A \mathbf{b}$

$A \rightarrow \mathbf{a} \mid B \mid \epsilon$ $flws_0:$

$B \rightarrow \mathbf{b} \mid \epsilon$

S	
A	
B	

$flws_1:$

S	
A	
B	

$flws_2:$

S	
A	
B	

$flws_3:$

S	
A	
B	

Putting it all together

- Given grammar G , calculate **FIRST** and **FOLLOW** sets.
- For each $A \in N$:
 - Suppose all productions for A are $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$.
 - Define, for $1 \leq i \leq k$):
$$F_{A,i} = \begin{cases} \text{FIRST}(\alpha_i) \cup \text{FOLLOW}(A), & \text{if } \bullet \in \text{FIRST}(\alpha_i) \\ \text{FIRST}(\alpha_i), & \text{otherwise} \end{cases}$$
- For each A , all $F_{A,i}$ must be pairwise disjoint. Otherwise, the grammar cannot be parsed by recursive descent! (The grammar is said to be “non-LL(1).”)
- Then define `parseA` as shown earlier: given input token t , choose the unique production $A \rightarrow \alpha_i$ such that $t \in F_{A,i}$, or raise an error.

Putting it all together (cont.)

- Consider:

$$\begin{array}{lcl}
 E & \rightarrow & T E' \\
 E' & \rightarrow & \epsilon \mid + E \\
 T & \rightarrow & P T' \\
 T' & \rightarrow & \epsilon \mid * T \\
 P & \rightarrow & \text{id} \mid (E)
 \end{array}$$

- We have previously calculated:

FIRST:	E		FOLLOW:	E	
	E'			E'	
	T			T	
	T'			T'	
	P			P	

Putting it all together (cont.)

- Calculate set F for each right-hand side:

$$F_{E,1} = \{ \quad \}$$

$$F_{E',1} = \{ \quad \} \quad F_{E',2} = \{ \quad \}$$

$$F_{T,1} = \{ \quad \}$$

$$F_{T',1} = \{ \quad \} \quad F_{T',2} = \{ \quad \}$$

$$F_{P,1} = \{ \quad \} \quad F_{P,2} = \{ \quad \}$$

- Is this grammar LL(1)?

Another example: FIRST sets

$S \rightarrow A B A \mid \mathbf{c} C$

$A \rightarrow \mathbf{a} \mid \epsilon$

$B \rightarrow \mathbf{b} D \mid \epsilon$

$C \rightarrow A D \mid \mathbf{b}$

$D \rightarrow \mathbf{a} A \mid \mathbf{c}$

$fst_0:$

S	
A	
B	
C	
D	

$fst_1:$

S	
A	
B	
C	
D	

$fst_2:$

S	
A	
B	
C	
D	

$fst_3:$

S	
A	
B	
C	
D	

Another example (cont.): FOLLOW sets

$S \rightarrow A B A \mid \mathbf{c} C$

$A \rightarrow \mathbf{a} \mid \epsilon$

$B \rightarrow \mathbf{b} D \mid \epsilon$

$C \rightarrow A D \mid \mathbf{b}$

$D \rightarrow \mathbf{a} A \mid \mathbf{c}$

FIRST:

S	
A	
B	
C	
D	

$flws_0:$

S	
A	
B	
C	
D	

$flws_1:$

S	
A	
B	
C	
D	

$flws_2:$

S	
A	
B	
C	
D	

$flws_3:$

S	
A	
B	
C	
D	

Non-LL(1) grammars

- Many grammars are not LL(1). There are two particularly common problems, one of which is easy to fix.
- If two productions from A start the same way, you can use *left refactoring*:

$$A \rightarrow \alpha \beta \mid \alpha \gamma \Rightarrow \begin{array}{l} A \rightarrow \alpha B \\ B \rightarrow \beta \mid \gamma \end{array}$$

Non-LL(1) grammars (cont.)

- A grammar is *left-recursive* if there is a non-terminal A and $\alpha \in S^*$ such that $A\alpha$ is an A -form.
- Left-recursion can be direct (i.e. there is a rule $E \rightarrow E + T$), or indirect (there are rules $E \rightarrow E'$, and $E' \rightarrow E + T$).
- Left-recursive grammars can never be LL(1), because the left-recursion will lead to an infinite loop.
- Sometimes, left-recursion can be removed:

$$\begin{aligned} A \rightarrow A \alpha \mid \beta &\Rightarrow A \rightarrow \beta B \\ B &\rightarrow \epsilon \mid \alpha B \end{aligned}$$

Producing ASTs

- AST's can be produced during a top-down parse, just as during a bottom-up parse.

- Change the type of a parsing function from

`parseA: token list  $\rightarrow$  token list`

to

`parseA: token list  $\rightarrow$  token list * ast`

- When `parseA` make a recursive call to, say, `parseB`, it can take the **AST** returned from `parseB` and then put it into the **AST** returned from `parseA`.

Handling expressions

- Only the *left*-recursive, stratified expression grammar correctly enforces left-associativity of addition and multiplication and precedence of multiplication.
- However, that grammar cannot be parsed with recursive descent because it is left-recursive.
- The grammar from our last two exercises:

$$E \rightarrow T E'$$

$$E' \rightarrow \epsilon \mid + E$$

$$T \rightarrow P T'$$

$$T' \rightarrow \epsilon \mid * T$$

$$P \rightarrow \text{id} \mid (E)$$

is LL(1), but it enforces *right*-associativity.

Handling expressions (cont.)

- So do we have to give up on top-down parsing for parsing programming languages? Yes and no:
 - There is no way around the problem: no right-recursive grammar can enforce left-associativity, and no left-recursive grammar can be parsed using recursive descent.
 - However, we can parse expressions and handle associativity manually, when converting to ASTs:

$$E \rightarrow T E' \{ \text{if } \$2 = \text{Empty then } \$1 \text{ else addplus } \$1 \ \$2 \}$$

where addplus re-jiggers the additions:

```
addplus t1 t2 = match t2 with
  Plus(Plus(t21, t22) as a, t2') -> Plus(addplus t1 a, t2')
| Plus(t21, t22) -> Plus(Plus(t1, t21), t22)
| _ -> Plus(t1, t2)
```