Lecture 9 — Top-down parsing

- Top-down parsing is a method simple enough to be used for writing parsers by hand. Compiler writers sometimes prefer to write parsers by hand because parser generators can be difficult to use and because some kinds of processing can be hard to fit into the generator.
- Top-down (recursive descent) parsing
  - Simple examples of recognizers and parsers
  - FIRST and FOLLOW sets
  - Handling non-LL(1) grammars
  - Expression grammars

Top-down parsing

- *Top-down*, or *recursive descent*, parsing is a parsing method where the structure of the parser directly follows the structure of the cfg.
- Each non-terminal A becomes a function `parse`A that reads the input and *consumes the part of it corresponding to an A-sentence.*
- Since each production for a non-terminal A represents a different way to form an A-sentence, the first job of `parse`A is to determine which production to use.
- We will consider only the case where that determination is made by looking at the next input token (the "lookahead symbol") and nothing else.

"Consuming" inputs

- The central idea of a recursive descent parser is that the function `parse`A matches an A-sentence as a prefix of the input, and consumes (or "discards") that prefix before returning.
- What does "consume" mean?
  - Can use global variable `input` that points to the next character, and then increment it *as a side effect*.
  - Can use a `token list` argument and *return a suffix of the argument*. Because we don't know how to write functions with side effects in OCaml, we will use this method.

Parsing and recognizing

- A *recognizer* simply reads the input and returns the remainder of the input, and no other value. For us, a recognizer has type

  `token list` $\longrightarrow$ `token list`

  and can also raise a syntax error. The input is considered *syntactically incorrect* if either (1) a syntax error is raised, or (2) there are still tokens (other than EOF) remaining in the input after calling the parser.

- A *parser* reads the input and returns the remainder of the input *and* a tree (the parse tree or AST). It has type

  `token list` $\longrightarrow$ `token list * ast`

A recursive-descent recognizer

- We will write a recognizer for this grammar:

  A → id | '(' B ')'
  B → int | A

- Define the token type as

```
type token = IDENT of string | LPAREN | RPAREN | INT of int | EO
```

- What will `parseA` return for these inputs?
  - `[LPAREN; INT 3; RPAREN; EOF]` [EOF]
  - `[IDENT "x"; INT 3; EOF]` [INT 3; EOF]
  - `[INT 3; IDENT "x"; EOF]` Raise Error
  - `[LPAREN; IDENT "x"; RPAREN; IDENT "y"; EOF]` [IDENT "y", EOF]
- What will `parseB` return for these inputs?
  - `[INT 3; RPAREN; EOF]` [RPAREN; EOF]
  - `[LPAREN; IDENT "x"; RPAREN; RPAREN; EOF]` [RPAREN, EOF]

A recursive-descent recognizer (cont.)

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
```

A recursive-descent parser

- **Adding code to construct a parse tree is straightforward.**

```
type token = IDENT of string | LPAREN | RPAREN | INT of int | EOF
type parsetree = A1 of token | A2 of token * parsetree * token
               | B1 of token | B2 of parsetree

(* parseA, parseB: token list -> token list * parsetree *)
let rec parseA toklis = match (hd toklis) with
    IDENT x -> (tl toklis, A1 (IDENT x))
  | LPAREN -> let (r,t) = parseB (tl toklis)
              in (if hd r = RPAREN then (tl r, A2(LPAREN, t, RPAREN))
                                   else raise SyntaxError)
   | _ -> raise SyntaxError

and parseB toklis = match (hd toklis) with
     INT i -> (tl toklis, B1 (INT i))
   | _ -> parseA toklis ;;
```

A recursive-descent parser (cont.)

- **What will `parseA` return for these inputs?**
  - `[IDENT "x"; INT 3; EOF]`

    [INT 3; EOF]
  - `[LPAREN; INT 3; RPAREN; EOF]`

    [EOF]
- **What will `parseB` return for these inputs?**
  - `[INT 3; RPAREN; EOF]` [RPAREN; EOF]
  - `[LPAREN; IDENT "x"; RPAREN; RPAREN; EOF]` [RPAREN; EOF]

Top-down parsing, formally

- **Given grammar G, for each $A \in N$, define:**

  `parseA toklis` = **based on** `hd toklis`, **choose production;**
  if $A \rightarrow X_1 X_2 \ldots X_n$ **chosen:** parseX_n(... (parseX_2 (parseX_1 toklis))..
  else if $A \rightarrow Y_1 Y_2 \ldots Y_p$ **chosen:** parseY_p(... (parseY_2 (parseY_1 toklis)).
  else if ***etc.***

  and for each $t \in T$, define

```
parset toklis = if hd toklis = t then tl toklis
                                 else raise SyntaxError
```

- **Note that if $n = 0$,** `parseA` **returns** `toklis`.

Top-down recognizer exercise

L → **E R**
E → **id**
R → **;** **L** | ϵ

```
type token = IDENT of string | SEMIC | EOF
let rec parseL toklis =
and parseE toklis =
and parseR toklis =
and parseIDENT toklis =
and parseSEMIC toklis =
```

parseR (parseE toklis)

parseIDENT toklis

if hd toklis = SEMIC
then parseL (parseSEMIC toklis) else toklis

if hd toklis = IDENT then tl toklis else error

if hd toklis = SEMIC then tl toklis else error

Top-down recognizer exercise redone

- **In practice, we never define separate functions for tokens, but instead inline them. Rewrite your parser with `parseIDENT` and `parseSEMIC` inlined:**

L → **E R**
E → **id**
R → **;** **L** | ϵ

```
type token = IDENT of string | SEMIC | EOF
let rec parseL toklis = parseR (parseE toklis)

and parseE toklis = if hd toklis = IDENT then tl toklis else error

and parseR toklis = match hd toklis with
                      SEMIC -> parseL (tl toklis)
                    | _ -> toklis
```

What now?

- We are "done"! Given a grammar, we can simply transcribe it directly into a recursive descent recognizer. (Then, if you like, inline the boring parts, and add tree-building code.)
- What's left to talk about? Couple of things...
  - The crucial point is where we said, "based on `hd toklis`, choose production." That is not always easy — in fact, it is not always possible. *It all depends on the grammar.*
  - We need to discuss *how* to make that decision, and *when* it is actually impossible.
  - First, there is one situation where our parsers will definitely not work...

Left recursion

- **The grammar**

  L $\rightarrow$ **L E ;** $\mid \epsilon$
  E $\rightarrow$ **id**

  produces this parser:

```
let rec parseL toklis = match hd toklis with
                IDENT _ -> parseSEMIC (parseE (parseL toklis))
              | _ -> toklis

and parseE toklis = parseIDENT toklis

and parseIDENT toklis = ... and parseSEMIC toklis = ...
```

- **Do you see the problem?**

A couple of new definitions...

- Left-recursive grammars definitely won't work. (We'll say a little more about this later.) We now consider the other problem: deciding which production to use.
- We already have the notion of an A-sentence — a string of terminals that can be derived from A (in a given grammar). Now extend that to any sequence α of grammar symbols:

  If $\alpha = X_1 X_2 \ldots X_n \in S^*$, an α-*sentence* is any string of terminals consisting of an X_1-sentence followed by an X_2-sentence, etc. (If X_i is a terminal symbol, then an "X_i-sentence" consists of just X_i itself.)

- One more definition: $\alpha \in S^*$ is *nullable* if ϵ is an α-*sentence*. That is: either $n = 0$ or α consists of nullable non-terminals.

α-sentences exercise

- **Given this grammar:**

  A $\rightarrow$ **int** | **(B)**
  B $\rightarrow$ **+ A B** | **int** | ϵ

  give examples of α-sentences for each α:

  - 3 : 3
  - A : (5)
  - + A B : +(5)6
  - (+ A B) : (+(5)6)

"choose production based on hd toklis"

- We are in `parse`A trying to decide which production to use. Use the idea of "FIRST sets" (we ignore ϵ-productions for the moment):
  - For $\alpha \in S^*$, FIRST(α) 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` $\in$ FIRST(α).
  - If there is no such production, reject the input.
  - If the production is not unique — that is, if the FIRST sets for all productions from A are not mutually disjoint — you can't parse this grammar by recursive descent!

FIRST set examples

A → **id** | **(B)**
B → **int** | **A**

FIRST(id) = { id }

FIRST(int) = { int }

FIRST((B)) = { (}

FIRST(A) = { id, (}

A → **int** | **(B)**
B → **A+B**

FIRST(int) = { int }

FIRST(A+B) = { int, (}

FIRST((B)) = { (}

Dealing with ϵ-productions

- If the grammar contains an ϵ-production, we can use it if no other production works.
- Determining which production to use changes to this:
  - For $\alpha \in S^*$, FIRST(α) is as defined above, except: if α is nullable, add the special symbol $\bullet$ to FIRST(α).
  - For any non-terminal, if the FIRST sets of its right-hand sides are not mutually disjoint, this grammar cannot be parsed top-down.
  - In `parseA`, choose the production $A \rightarrow \alpha$ such that `hd toklis` $\in$ FIRST(α); if there is none, choose the production for which $\bullet \in \alpha$; if there is none, reject the input.

Another FIRST sets example

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

FIRST(TE' **)** = { id, (}

FIRST(ϵ **)** = { • }

FIRST($+E$ **)** = { + }

FIRST(PT' **)** = { id, (}

FIRST(ϵ **)** = { • }

FIRST($*T$ **)** = { * }

FIRST(id **)** = { id }

FIRST((E) **)** = { (}

Another FIRST sets example (cont.)

- **Write the first few parsing functions for the above grammar:**

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

```
let rec parseE toklis = parseE' (parseT toklis)

and parseE' toklis =  match hd toklis with
                        + -> parseE (tl toklis)
                      | _ -> toklist

and parseT toklis =  parseT' (parseP toklis)
```

FOLLOW sets

- There remains a subtle problem with our parser construction process. Consider this grammar:

$$\begin{array}{lcl} A & \rightarrow & BC \\ B & \rightarrow & \mathbf{b} \mid \epsilon \\ C & \rightarrow & \mathbf{b}\ \mathbf{c} \end{array}$$

- This *seems to be* parsable top-down — i.e. FIRST sets of right-hand sides are non-overlapping. However, input `bc` is not handled correctly.

- Problem is, if b is the first input, we cannot tell which of these will be the correct parse tree:

LL(1) grammars

- For non-terminal A, FOLLOW(A) is the set of terminal symbols that can *immediately* follow A in a sentential form.
- *Def.* A grammar is $LL(1)$ iff it can be parsed correctly using recursive descent, with only one lookahead symbol.
- *Thm* A grammar is LL(1) under the following conditions:
  - For any $A \in S$, FIRST sets of all right-hand sides of A are mutually disjoint.
  - If there are productions $A \rightarrow \alpha$ and $A \rightarrow \beta$, and $\bullet \in$ FIRST(α), then there cannot be a terminal symbol t such that $t \in$ FOLLOW(A) and $t \in$ FIRST(β).
  - It has no left-recursive rules.
- Note that no ambiguous grammar is LL(1).

Non-LL(1) grammars

- Many grammars are not LL(1). There are two common problems that can sometimes be solved by fairly simple changes in the grammar.
  - If two productions from A start the same way, you can use *left factoring*:

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

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:

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

Handling expressions

- Recall from last week: The *left*-recursive, stratified expression grammar correctly enforces left-associativity and precedence of multiplication:

  $$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{T} \mid \mathbf{T}$$
  $$\mathbf{T} \rightarrow \mathbf{id} \mid \mathbf{T} * \mathbf{id}$$

- However, that grammar cannot be parsed with recursive descent because it is left-recursive.

- The *right*-recursive grammar

  $$\mathbf{E} \rightarrow \mathbf{T} + \mathbf{E} \mid \mathbf{T}$$
  $$\mathbf{T} \rightarrow \mathbf{id} \mid \mathbf{id} * \mathbf{T}$$

  is not LL(1), but ...

Handling expressions (cont.)

- that can be fixed by left-factoring:

$$\begin{array}{l} \mathbf{E} \rightarrow \mathbf{T} + \mathbf{E} \mid \mathbf{T} \\ \mathbf{T} \rightarrow \mathbf{id} \mid \mathbf{id} * \mathbf{T} \end{array} \quad \Rightarrow \quad \begin{array}{lcl} \mathbf{E} & \rightarrow & \mathbf{T}\ \mathbf{E'} \\ \mathbf{E'} & \rightarrow & \epsilon \mid + \mathbf{E} \\ \mathbf{T} & \rightarrow & \mathbf{id}\ \mathbf{T'} \\ \mathbf{T'} & \rightarrow & \epsilon \mid * \mathbf{T} \end{array}$$

- This is LL(1). Unfortunately, it has really ugly parse trees. But the real problem with it is that it enforces *right*-associativity.

- Can we fix the right-associativity problem? Yes and no: We can't fix the grammar — no right-recursive grammar can enforce left-associativity, and no left-recursive grammar is LL(1). But, ...

Handling expressions (cont.)

- **... we can fix the AST. Start with a straightforward parser. The AST reflects the shape of the concrete syntax tree, so enforces right-associativity.**

```
type ast = ID of string | ADD of ast * ast | MULT of ast * ast | NONE

let rec parseE toklis = let (r,t) = parseT toklis
                        in let (r',t') = parseE' r
                           in (r', if t'=NONE then t else ADD(t,t'))

and parseE' toklis = if hd toklis = PLUS
                     then parseE (tl toklis)
                     else (toklis, NONE)

...

# parseE [IDENT "x"; PLUS; IDENT "y"; PLUS; IDENT "z"; EOF];;
- : token list * ast = ([EOF], ADD (ID "x", ADD (ID "y", ID "z")))
```

Handling expressions (cont.)

- **But we can just re-jigger the AST as we parse. Define:**

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

and change `parseE` in one place:

```
let rec parseE toklis = let (r,t) = parseT toklis
                        in let (r',t') = parseE' r
                           in (r', if t'=NONE then t else addplus t t')
```

- **Now we get left-recursion:**

```
# parseE [IDENT "x"; PLUS; IDENT "y"; PLUS; IDENT "z"; EOF];;
- : token list * ast = ([EOF], ADD (ADD (ID "x", ID "y"), ID "z"))
```

Wrap-up

- Today we discussed:
  - Top-down (recursive descent) parsing
  - FIRST and FOLLOW sets, used in constructing top-down parsers (and in deciding whether a grammar can be parsed top-down).
  - Handling expressions top-down. (*See supplementary notes on web for fuller version of this discussion.*)
- We discussed it because:
  - It is a method that can be used for hand-writing parsers, which is sometimes easier than using a generator.
- In the next class, we will:
  - Begin to talk about the "back end" of the compiler
- What to do now:
  - *HW5*