Lecture 7: Top-down parsing

- **Context-free grammars**
- **Top-down parsing, a.k.a. recursive-descent parsing**

Context-free Grammar

- **Def: Given set of non-terminals V, set of terminals or tokens T, a cfg G is a set of productions of the form**

$$A \rightarrow X_1 \ldots X_n \quad (n \geq 0)$$

where $A \in V$, $X_1, \ldots, X_n \in G = V \cup T$

- **Notation:**
  - **Elements of V: A, B, C**
  - **Elements of T: x, y, z**
  - **Elements of G: X, Y**

More Notation

- $A \to X_1 \ldots X_n$ **also written**

$$A ::= X_1 \ldots X_n$$

- **When** $n = 0$ **write**

$$A \to \epsilon$$

instead of

$$A \to$$

- **When there is more than one production from** A**, say**

$$A \to X_1 \ldots X_n \textbf{ and } A \to Y_1 \ldots Y_n$$

write

$$A \to X_1 \ldots X_n \mid Y_1 \ldots Y_n$$

Example 1

- **Expressions**
  - **Exp $\longrightarrow$ intlit | variable | Exp + Exp | Exp * Exp**
- **Sentences include**
  - **3**
  - **x**
  - **3+x**
  - **3+x*y**

Example 2

MethodDef $\rightarrow$ Type ident (Args) { Stmtlist }

Args $\rightarrow \epsilon$ | NonEmptyArgs

NonEmptyArgs $\rightarrow$ Type ident |
Type ident , NonEmptyArgs

StmtlistArgs $\rightarrow \epsilon$ | Stmt Stmtlist

Type $\rightarrow$ ident | int | boolean

- **Sentence: int fun(boolean b) { }**

Syntax Tree

- **A syntax tree is a tree whose internal nodes are labelled with non-terminals such that if a node is labelled A, its children are labelled $X_1, \ldots, X_n$ for some production $A \rightarrow X_1, \ldots, X_n$.**

Sentences of a grammar are frontiers of syntax tree whose root is the start symbol.

More notation

- **"Extended Backus-Naur Form" (EBNF)**
  - $A \rightarrow X_1 \ldots X_i (Y_1 \ldots Y_k)^* X_{i+1} \ldots X_n$

    $\Rightarrow A \rightarrow X_1 \ldots X_i \ B \ X_{i+1} \ldots X_n$

    $B \rightarrow \epsilon \mid Y_1 \ldots Y_k B$
  - $A \rightarrow X_1 \ldots X_i (Y_1 \ldots Y_k)^+ X_{i+1} \ldots X_n$

    $\Rightarrow A \rightarrow X_1 \ldots X_i \ B \ X_{i+1} \ldots X_n$

    $B \rightarrow Y_1 \ldots Y_k \mid Y_1 \ldots Y_k B$
  - $A \rightarrow X_1 \ldots X_i (Y_1 \ldots Y_k)? \ X_{i+1} \ldots X_n$

    $\Rightarrow A \rightarrow X_1 \ldots X_i \ B \ X_{i+1} \ldots X_n$

    $B \rightarrow \epsilon \mid Y_1 \ldots Y_k$
  - **Example: Args $\rightarrow$ (Type ident (, Type ident)*)?**

Parsing

- From list of tokens, construct syntax tree
- Simpler problem: determine whether list of tokens is a sentence ("recognition").
- Two types of parsers: Top-down and Bottom-up.
- We will discuss recursive descent (top-down) and LR(1) (bottom-up)
  - Not all grammars can be parsed by any particular method
  - Recursive descent is easier to use by hand.
    LR(1) requires a generator.
  - LR(1) more powerful: can be applied to more grammars.

Top-down parsing by recursive descent

- **Idea: Define a function `parseA` for each non-terminal A. Given token, decide which production from A to apply, say $A \rightarrow X_1 \ldots X_n$. Go through $X_1 \ldots X_n$ in sequence, consuming tokens in $X_1 \ldots X_n$, and recursively calling parsing function `parseX`$_i$ for non-terminals.**

- **Details:**

  - **Each function will return list of *remaining* tokens**
  - **Error is reported if a any of the X_i is a token that does not match the input token.**
  - **Input is accepted if parse function returns empty list.**

Example: A → id | '(' A ')'

- **Define** `parseA: token list -> (token list) option`
  - `'a option = None | Some 'a`
- `parseA toklis` **matches first part of** `toklis` **and returns remainder of** `toklist`**, or** `None` **if syntax error.**

```
type token = IDENT of string | LPAREN | RPAREN

let rec parseA toklis = match toklis with
    IDENT x :: tls -> Some tls
  | LPAREN :: tls ->
      (match (parseA tls) with
          Some (h::tls') -> if h = RPAREN
                            then Some tls'
                            else None
        | _ -> None)
   | _ -> None;;
```

Slightly more complicated example

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

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

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

let rec parseA toklis = match toklis with
    IDENT x :: tls -> Some tls
  | LPAREN :: tls ->
      (match (parseB tls) with
          Some (h::tls') -> if h = RPAREN
                            then Some tls'
                            else None
        | _ -> None)
   | _ -> None

and parseB toklis = match toklis with
     INT i :: tls -> Some tls
   | _ -> parseA toklis;;
```

Even more complicated example

- **Consider this grammar:**

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

- **Unfortunately, cannot parse that grammar using recursive descent. This grammar has the same sentences and is parsable by recursive descent:**

  A $\rightarrow$ **id** | **'(' B ')'**
  B $\rightarrow$ **A C**
  C $\rightarrow$ **'+' A C** | ϵ

```
type token = IDENT of string | LPAREN | RPAREN | PLUS
```

cont.

```
let rec parseA toklis = match toklis with
    IDENT x :: tls -> Some tls
  | LPAREN :: tls ->
      (match (parseB tls) with
          Some (h::tls') -> if h = RPAREN
                            then Some tls'
                            else None
        | _ -> None)
   | _ -> None

and parseB toklis = match parseA toklis with
     Some tls' -> parseC tls' | None -> None

and parseC toklis = match toklis with
     PLUS :: tls' -> (match parseA tls' with
                         Some tls'' -> parseC tls''
                       | None -> None)
   | _ -> Some toklis;;
```

Generating syntax trees

- **For simple grammar — A → id | '(' A ')' — define type for syntax trees:**

```
type cst = A1 of token * cst * token | A2 of token
```

- **Parse function returns pair of remaining tokens and syntax tree created by this non-terminal:**

```
let rec parseA toklis = match toklis with
    IDENT x as y :: tls -> Some (tls, A2 y)
  | LPAREN :: tls ->
      (match (parseA tls) with
          Some (h::tls', t) -> if h = RPAREN
                               then Some (tls', A1 (LPAREN, t, RPAREN))
                               else None
        | _ -> None)
   | _ -> None;;
```

Generating syntax trees — second grammar

- **Don't need to create specialized cst type — can use general tree structure.**

```
type tree = Node of string * tree list | Leaf of token;;

let rec parseA toklis = match toklis with
    IDENT x :: tls -> Some (tls, Node("A1", [Leaf (IDENT x)]))
  | LPAREN :: tls ->
      (match (parseB tls) with
          Some (h::tls', t)
             -> if h = RPAREN
                then Some (tls', Node("A2", [Leaf LPAREN;
                                             t; Leaf RPAREN]))
                else None
        | _ -> None)
   | _ -> None
```

```
and parseB toklis = match toklis with
      INT i :: tls -> Some (tls, Node("B1", [Leaf (INT i)]))
    | _ -> (match parseA toklis with
              Some (tls, t) -> Some(tls, Node("B2", [t]))
            | None -> None);;
```

Generating syntax trees — third grammar

```
let rec parseA toklis = match toklis with
    IDENT x :: tls -> Some (tls, Leaf (IDENT x))
  | LPAREN :: tls ->
      (match (parseB tls) with
          Some (h::tls', t) ->
                      if h = RPAREN
                      then Some (tls', Node("A1", [Leaf LPAREN;
                                 t; Leaf RPAREN]))
                      else None
        | _ -> None)
   | _ -> None

and parseB toklis = match parseA toklis with
     Some (tls', t) -> (match parseC tls' with
                   Some (tls'', t') -> Some(tls'', Node("B", [t; t']))
                 | None -> None)
   | None -> None
```

```
and parseC toklis = match toklis with
     PLUS :: tls' ->
         (match parseA tls' with
                Some (tls'', t) -> (match parseC tls'' with
                     Some(tls''', t') -> Some(tls''',
                          Node("C1", [Leaf PLUS; t; t']))
                   | None -> None)
              | None -> None)
    | _ -> Some (toklis, Node("C2", []));;
```

Generating abstract syntax trees

- **Concrete syntax tree shows every production, even though some are not *semantically* significant - e.g. no reason to keep tokens '(' and ')' in tree.**
- **AST should have simplest structure that retains all significant details.**
- **For this grammar, should retain effect of parenthesization (would be important if we used minus instead of plus).**
- **AST form: Interior nodes of arbitrary arity, labeled with "PLUS"; leaf nodes labeled with identifier**

Generating abstract syntax trees

```
let rec parseA toklis = match toklis with
    IDENT x :: tls -> Some (tls, Leaf (IDENT x))
  | LPAREN :: tls ->
      (match (parseB tls) with
          Some (h::tls', t) -> if h = RPAREN
                               then Some (tls', t)
                               else None
        | _ -> None)
   | _ -> None

and parseB toklis = match parseA toklis with
     Some (tls', t) -> (match parseC tls' with
                          Some (tls'', []) -> Some (tls'', t)
                        | Some (tls'', tlis) ->
                                Some (tls'', Node("+", t :: tlis))
                        | None -> None)
   | None -> None
```

```
and parseC toklis = match toklis with
     PLUS :: tls' ->
            (match parseA tls' with
                Some (tls'', t) -> (match parseC tls'' with
                     Some(tls''', t') -> Some(tls''', t :: t')
                   | None -> None)
              | None -> None)
   | _ -> Some (toklis, []) ;;
```

Next class

- More formal treatment of recursive-descent parsing
  - When can a grammar be parsed using recursive descent?
    - "LL(1)" condition
    - Ambiguity
  - Grammar transformations