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, and a set of terminals or tokens T, a cfg G is a set of productions of the form**

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

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

- **Notation:**
  - **Elements of V: A, B, C**
  - **Elements of T: x, y, z**
  - **Elements of $V \cup T$ ("grammar symbols") : 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

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

Example

MethodDef → Type ident '(' Args ')' '{' Stmtlist '}'

Args → ϵ | NonEmptyArgs

NonEmptyArgs → Type ident |
Type ident ',' NonEmptyArgs

StmtlistArgs → ϵ | Stmt Stmtlist

Type → 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) — right-hand sides can contain regular expressions. (α, β, γ in $(V \cup T)*$.)**
  - $A \rightarrow \alpha(\beta)^*\gamma$

    $\Rightarrow A \rightarrow \alpha\ B\ \beta$

    $B \rightarrow \epsilon \mid \gamma B$
  - $A \rightarrow \alpha(\beta)^+\gamma$

    $\Rightarrow A \rightarrow \alpha\ B\ \gamma$

    $B \rightarrow \beta \mid \beta B$
  - $A \rightarrow \alpha(\beta)?\ \gamma$

    $\Rightarrow A \rightarrow \alpha\ B\ \gamma$

    $B \rightarrow \epsilon \mid \beta$
  - **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 either method. (Methods applicable to arbitrary grammars are not efficient.)
  - 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 next token, `parseA` decides which production from A to apply, say $A \rightarrow X_1 \ldots X_n$. Goes 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.**
  - **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.**
- **Here, we allow only one "lookahead symbol;" could allow more, but it gets more complicated to explain.**

Example 1: A → id | '(' A ')'

- **Define** `parseA: token list -> token list`
- `parseA toklis` **matches first part of** `toklis` **and returns remainder of** `toklis`**, or throws exception if syntax error.**

```
exception SyntaxError

type token = IDENT of string | LPAREN | RPAREN

let rec parseA toklis = match toklis with
    IDENT x :: tls -> tls
  | LPAREN :: tls ->
      (match (parseA tls) with
          (h::tls') -> if h = RPAREN
                       then tls'
                       else raise SyntaxError
        | _ -> raise SyntaxError)
   | _ -> raise SyntaxError;;
```

Example 2

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

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

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

let rec parseA toklis = match toklis with
    IDENT x :: tls -> tls
  | LPAREN :: tls ->
      (match (parseB tls) with
          (h::tls') -> if h = RPAREN
                       then tls'
                       else raise SyntaxError
        | _ -> raise SyntaxError)
   | _ -> raise SyntaxError

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

Example 3

- **Consider this grammar:**

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

- **Cannot parse that grammar using recursive descent (even with more than one lookahead symbol). 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 -> tls
  | LPAREN :: tls ->
      (match (parseB tls) with
          (h::tls') -> if h = RPAREN
                       then tls'
                       else raise SyntaxError
        | _ -> raise SyntaxError)
   | _ -> raise SyntaxError

and parseB toklis = match parseA toklis with
     tls' -> parseC tls'
   | _ -> raise SyntaxError

and parseC toklis = match toklis with
     PLUS :: tls' -> (match parseA tls' with
                        tls'' -> parseC tls''
                      | _ -> raise SyntaxError)
   | _ -> toklis;;
```

Generating syntax trees — ex. 1b

- **For simple grammar — A → id | '(' A ')' — define type for concrete 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 -> (tls, A2 y)
  | LPAREN :: tls ->
      (match (parseA tls) with
          (h::tls', t) -> if h = RPAREN
                          then (tls', A1 (LPAREN, t, RPAREN))
                          else raise SyntaxError
        | _ -> raise SyntaxError)
   | _ -> raise SyntaxError;;
```

Generating syntax trees — ex. 2b

- **Instead of specialized cst type, can use general tree structure.**

```
let rec parseA toklis = match toklis with
    IDENT x :: tls -> (tls, Node("A1", [Leaf (IDENT x)]))
  | LPAREN :: tls ->
      (match (parseB tls) with
          (h::tls', t)
             -> if h = RPAREN
                then (tls', Node("A2", [Leaf LPAREN;
                                        t; Leaf RPAREN]))
                else raise SyntaxError
        | _ -> raise SyntaxError)
   | _ -> raise SyntaxError

and parseB toklis = match toklis with
     INT i :: tls -> (tls, Node("B1", [Leaf (INT i)]))
   | _ -> (match parseA toklis with
            (tls, t) -> (tls, Node("B2", [t]))
          | _ -> raise SyntaxError);;
```

Generating syntax trees — ex. 3b

```
let rec parseA toklis = match toklis with
    IDENT x :: tls -> (tls, Leaf (IDENT x))
  | LPAREN :: tls ->
      (match (parseB tls) with
          (h::tls', t) -> if h = RPAREN
                          then (tls', Node("A1", [Leaf LPAREN;
                                                  t; Leaf RPAREN]))
                          else raise SyntaxError
        | _ -> raise SyntaxError)
   | _ -> raise SyntaxError

and parseB toklis = match parseA toklis with
     (tls', t) -> (match parseC tls' with
                     (tls'', t') -> (tls'', Node("B", [t; t']))
                   | _ -> raise SyntaxError)
   | _ -> raise SyntaxError

and parseC toklis = match toklis with
```

```
PLUS :: tls' -> (match parseA tls' with
                   (tls'', t) -> (match parseC tls'' with
                                    (tls''', t') -> (tls''',
                                                     Node("C1", [Leaf PLUS; t; t']))
                                  | _ -> raise SyntaxError)
                 | _ -> raise SyntaxError)
| _ -> (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 ASTs — ex. 3c

```
let rec parseA toklis = match toklis with
    IDENT x :: tls -> (tls, Leaf (IDENT x))
  | LPAREN :: tls ->
      (match (parseB tls) with
          (h::tls', t) -> if h = RPAREN
                          then (tls', t)
                          else raise SyntaxError
        | _ -> raise SyntaxError)
   | _ -> raise SyntaxError

and parseB toklis = match parseA toklis with
        (tls', t) -> (match parseC tls' with
                        (tls'', []) -> (tls'', t)
                      | (tls'', tlis) -> (tls'', Node("+", t :: tlis))
                      | _ -> raise SyntaxError)
   | _ -> raise SyntaxError

and parseC toklis = match toklis with
```

```
  PLUS :: tls' ->  (match parseA tls' with
                      (tls'', t) -> (match parseC tls'' with
                                       (tls''', t') -> (tls''', t :: t')
                                     | _ -> raise SyntaxError)
                    | _ -> raise SyntaxError)
| _ -> (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