CS 421 Lecture 7: Grammars and parsing

- Announcements
- MP2 review
- Lecture outline
  - Context-free grammars
  - Top-down, a.k.a. recursive descent, parsing

Announcements

- TA office hours
  - I2CS: Tue, Thu 4-5pm CDT
  - On-campus: Wed 4-5pm CDT
- MP2 solutions posted

MP2 review

- Problem 7

```
flatten : ‘a list list -> ‘a list

flatten [[1;2;3]; [4;5]; [8;2;3;4]];;

let rec flatten lst = match lst with …
```

MP2 review

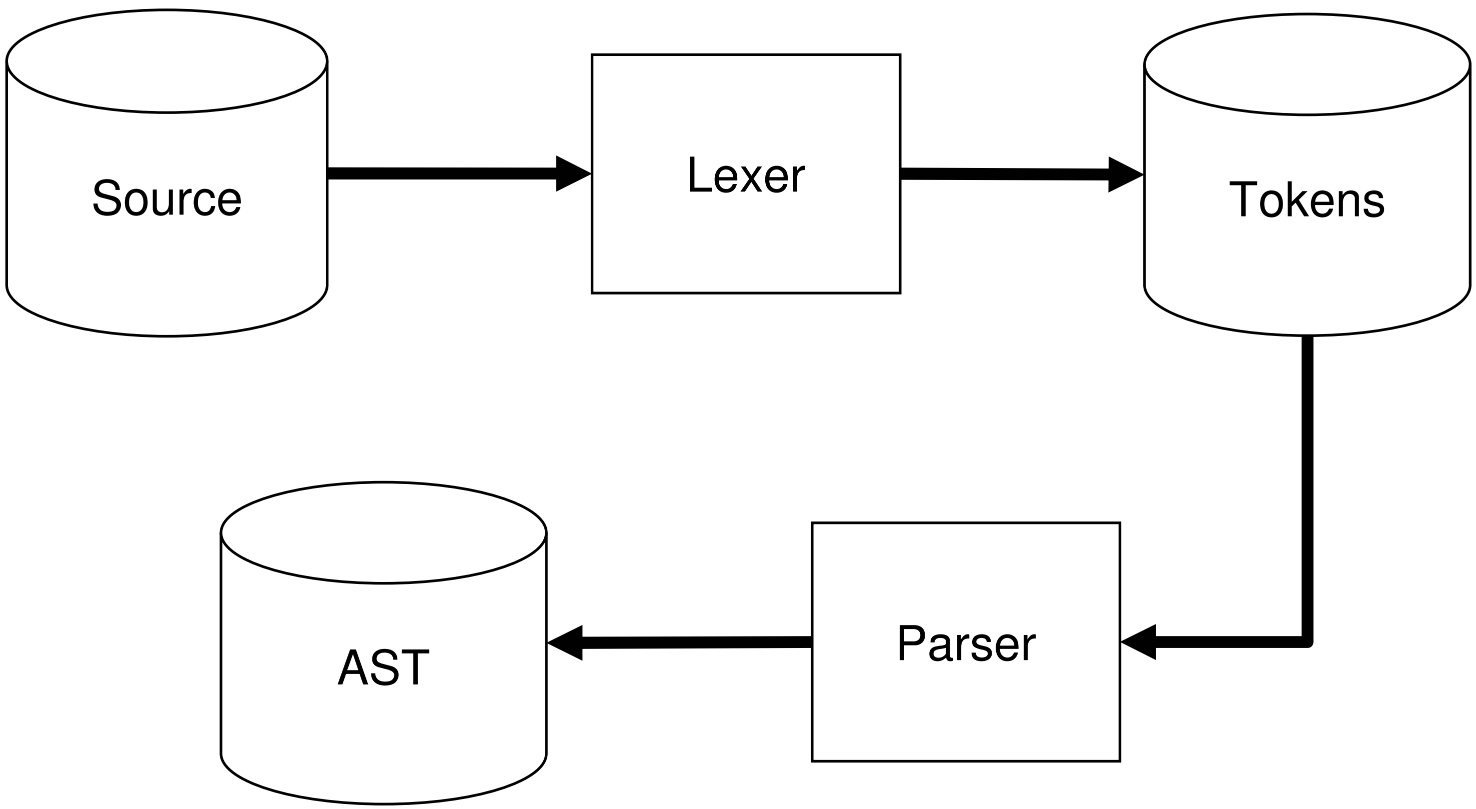
- Problem 7

```
flatten : 'a list list -> 'a list

flatten [[1;2;3]; [4;5]; [8;2;3;4]];;

let rec flatten lst = match lst with
  [] -> []
  | []::xs -> flatten xs
  | (x::xs)::ys -> x::(flatten (xs::ys));;
```

Review: compiler front-end



Intro to grammars and languages

- Grammar
  - Finite set of *terminals*
  - Finite set of *non-terminals*
  - Finite set of *production rules*
  - Start symbol
- Language
  - Set of strings recognized by a grammar

Grammars: Chomsky hierarchy

- Unrestricted
  - Recursively-enumerable languages
  - Recognized by a Turing machine
- Context-sensitive
  - Context-sensitive languages
  - Recognized by a linear bounded automaton (LBA)
- Context-free
  - Context-free languages
  - Recognized by a push-down automaton (PDA)
- Regular
  - Regular languages
  - Recognized by a finite state automaton (FSA)

Context-free grammar

- Given:
  - Set of terminals (tokens) T
  - Set of non-terminals (variables) V
- A cfg G is a set of *productions* of the form
  - $A \rightarrow X_1 \ldots X_n \quad (n \geq 0)$

  where
  - $A \in V, X_1 \ldots X_n \in G = V \cup T$
  - One symbol designated as "start symbol"

Notation

- $A \rightarrow X_1 \ldots X_n$
  - Also written $A ::= X_1 \ldots X_n$
- When $n = 0$, write $A \rightarrow \varepsilon$
  - Instead of $A \rightarrow$
- When there is more than one production from A, say
  - $A \rightarrow X_1 \ldots X_n$ and $A \rightarrow Y_1 \ldots Y_n$
  - Instead write: $A \rightarrow X_1 \ldots X_n \mid Y_1 \ldots Y_n$

Example

- Expressions
  - Exp → intlit | variable | Exp + Exp | Exp * Exp
- Sentences include
  - 3
  - x
  - 3+x
  - 3+x*y
- Tree representation

Example

- Method definition:

```
MethodDef     → Type ident '(' Args ')' '{' Stmtlist '}'
Args          → ε | NonEmptyArgs
NonEmptyArgs  → Type ident | Type ident ',' NonEmptyArgs
Stmtlist      → ε | Stmt Stmtlist
Type          → ident | int | boolean
```

- Sentence:

```
int fun(boolean b) { }
```

- Tree representation
  - ??

Syntax trees

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

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

More notation

- Backus-Naur Form (BNF)
  - Symbol → expression
  - Expression: terminals, symbols, |
- Extended BNF (EBNF)
  - Symbol → "terminal" | 'terminal' | <symbol> | … ;
  - RegExp-like extensions: exp*, exp+, exp?, *etc.*

Example

- EBNF:
  - $A \rightarrow X_1 \ldots X_i \ (Y_1 \ldots Y_k)^* \ X_{i+1} \ldots X_n$
    - $A \rightarrow X_1 \ldots X_i \ \mathrm{B} \ X_{i+1} \ldots X_n$
    - $B \rightarrow \varepsilon \mid Y_1 \ldots Y_k B$
  - $A \rightarrow X_1 \ldots X_i \ (Y_1 \ldots Y_k)+ \ X_{i+1} \ldots X_n$
    - $A \rightarrow X_1 \ldots X_i \ \mathrm{B} \ X_{i+1} \ldots X_n$
    - $B \rightarrow Y_1 \ldots Y_k \mid Y_1 \ldots Y_k B$
  - $A \rightarrow X_1 \ldots X_i \ (Y_1 \ldots Y_k)? \ X_{i+1} \ldots X_n$
    - $A \rightarrow X_1 \ldots X_i \ \mathrm{B} \ X_{i+1} \ldots X_n$
    - $B \rightarrow \varepsilon \mid Y_1 \ldots Y_k$
- Args rule from previous example:

```
Args → (Type ident (‘,’ Type ident)*)?
```

Parsing

- From list of tokens, construct a 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) parsers
  - 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 $\texttt{parseX}_{\texttt{i}}$ for non-terminals.

- Details: `parseA : token list -> token list` (almost)
  - Each function will return a list of *remaining* tokens
  - Error is reported if any of the X_i is a token that does not match the input token.
  - Input is accepted if parse function returns empty list.

parseA: actual type

```
parseA : token list -> (token list) option

type 'a option  = None | Some 'a
```

Example 1

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

Example 1 (cont.)

- A → id | '(' A ')'

```
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;;
```

Example 2

- A → id | '(' B ')'
- B → int | 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;;
```

Example 3

- Consider this grammar:
  - $A \rightarrow$ id | '(' B ')'
  - $B \rightarrow$ A | A '+' B
  - Unfortunately, cannot parse using recursive descent

- This grammar, which has the same sentences:
  - $A \rightarrow$ id | '(' B ')'
  - $B \rightarrow$ A C
  - $C \rightarrow$ '+' A C | ε
  - *Is* parsable by recursive descent

Example 3 (cont.)

- Tree representation: ((x + y) + z)

Example 3 (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 – ex. 1b

- 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 :: tls -> Some (tls, A2 (IDENT x))
  | 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 – ex. 2b

- 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; RPAREN]))
                else None
        | _ -> None)
  | _ -> None
...
```

Generating syntax trees – ex. 2b

```
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 – ex. 3b

```
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; 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
...
```

Generating syntax trees – ex. 3b

```
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

AST for example 3

- Convert CST to AST

- … or generate AST during parsing

Generating ASTs – ex. 3c

```
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
...
```

Generating ASTs – ex. 3c

```
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?
    - "LL1()" condition
    - Ambiguity
  - Grammar transformations