Lecture 9: Bottom-up parsing; ocamlyacc

- **Bottom-up parsing**
- **ocamlyacc**

Top-down vs. bottom-up parsing

- **Why is top-down called "top-down"?**

  As we consume tokens, we build a parse tree. At any time, we are filling in the children of a particular non-terminal. *As soon as we decide what production to use*, **we can fill in the tree. In this sense, we are building the tree from the top down.**

- **Example:**

$$E \rightarrow id\ T$$
$$T \rightarrow \epsilon \mid +\ E \mid *\ E$$

  Input: x + y * z

Bottom-up parsing

- **Bottom-up parsing works by creating small parse trees and joining them together into larger ones.**
- **Example:**
$$\begin{aligned} E &\rightarrow id\ T \\ T &\rightarrow \epsilon \mid + E \mid * E \end{aligned}$$

Input: x + y * z

How bottom-up parsing works

- **Keep a stack of small parse trees. Based on what's in this stack, and the next input token, take one of these actions:**
  - **Shift: Move lookahead token to stack**
  - **Reduce $A \rightarrow \alpha$: If roots of trees on stack match α, replace those trees on stack by single tree with root A.**
  - **Accept: Reduce when non-terminal is goal, look-ahead is eof**
  - **Reject**
- **Bottom-up parsing is also called *shift-reduce parsing*.**

Shift-reduce example 1

- **Example:** $$\begin{aligned} L &\rightarrow L\ ;\ E \mid E \\ E &\rightarrow id \end{aligned}$$

  Input: x; y; z

Shift-reduce example 2

- **Example:**

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

Input: x + 10 * y

Using ocamlyacc

- **Input attribute grammar is put in file $<grammar>$.mly**
- **Execute ocamlyacc $<grammar>$.mly**
- **Produces code for parser in $<grammar>$.ml and interface (including type declaration for tokens) in $<grammar>$.mli**

Parser code

- $<grammar>$.ml defines one parsing function per entry point
- Parsing function takes a lexing function (lexer buffer to token) and a lexer buffer as arguments
- Returns semantic attribute of corresponding entry point

Example - expression grammar

In this example, we will take a simple expression grammar and create a parser to parse inputs and produce abstract syntax.

Grammar:

$$
\begin{array}{lcl}
M & \rightarrow & Exp\ eof \\
Exp & \rightarrow & Term \mid Term\ +\ Exp \mid Term\ -\ Exp \\
Term & \rightarrow & Factor \mid Factor\ *\ Term \mid Factor\ /\ Term \\
Factor & \rightarrow & id \mid (\ Exp\ )
\end{array}
$$

Abstract syntax:

```
(* File: expr.ml *)
type expr =
    Plus of expr * expr
  | Minus of expr * expr
  | Mult of expr * expr
  | Div of expr * expr
  | Id of string
```

Example - lexer

```
(* File: exprlex.mll *)
let numeric = ['0' - '9']
let letter = ['a' - 'z' 'A' - 'Z']
rule tokenize = parse
  | "+" {Plus_token}
  | "-" {Minus_token}
  | "*" {Times_token}
  | "/" {Divide_token}
  | "(" {Left_parenthesis}
  | ")" {Right_parenthesis}
  | letter (letter | numeric | "_")* as id {Id_token id}
  | [' ' '\t' '\n'] {token lexbuf}
  | eof {EOL}
```

Example - parser

```
(* File: exprparse.mly *)
%{ open Expr
%}
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
```

Example - parser (exprparse.mly)

```
expr:
    term                          {$1}
  | term Plus_token expr          {Plus($1,$3)}
  | term Minus_token expr         {Minus($1,$3)}

term:
    factor                        {$1}
  | factor Times_token term       {Mult($1,$3)}
  | factor Divide_token term      {Div($1,$3)}

factor:
    Id_token {Id $1}
  | Left_parenthesis expr Right_parenthesis {$2}

main:
  | expr EOL {$1}
```

Example - using parser

```
# #use "expr.ml";;
...
# #use "exprparse.ml";;
...
# #use "exprlex.ml";;
...
# let test s =
let lexbuf = Lexing.from_string(s^"\n") in
   main tokenize lexbuf;;
# test "a + b";;
- :  expr = Plus(Id "a",Id "b")
```

ocamlyacc Input

- **File format:**

```
%{
  <header>
%}
  <declarations>
%%
  <rules>
%%
  <trailer>
```

ocamlyacc $<header>$

- Contains arbitrary Ocaml code
- Typcially used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted
- $<footer>$ similar. Possibly used to call parser

ocamlyacc *<declarations>*

- `%token` *symbol* ... *symbol*
  Declare given symbols as tokens

- `%token` *<type>* *symbol* ... *symbol*
  Declare given symbols as token constructors, taking an argument of type *type*

- `%start` *symbol* ... *symbol*
  Declare given symbols as entry points; functions of same names in *<grammar>***.ml**

ocamlyacc <*declarations*>

- `%type` <*type*> *symbol* ... *symbol*
  Specify type of attributes for given symbols. Mandatory for start symbol

- `%left` *symbol* ... *symbol*

- `%right` *symbol* ... *symbol*

- `%nonassoc` *symbol* ... *symbol*
  Associate precedences and associativities to given symbols. Same line, same precedence; earlier line, lower precedence (broadest scope)

ocamlyacc <*rules*>

- *nonterminal* :
  symbol ... *symbol* { *semantic_action* }
  | ...
  | *symbol* ... *symbol* { *semantic_action* }
  ;
- Semantic actions are arbitrary Ocaml expressions
- Must be of same type as declared (or inferred) for *nonterminal*
- Access values semantic attributes of symbols by position: $1 for first symbol, $2 for second, etc.

Friday's class

- **Big question: how to choose whether to shift or reduce.**
- **ocamlyacc uses a method — called $LALR(1)$ — to construct tables which say what action to take.**
- **There are times when there is no good way to make this decision. (ocamlyacc will reject grammar and give an error message.) In bottom-up parsing, these are called *conflicts*. There are two types: shift/reduce and reduce/reduce.**
- **As with top-down parsing, these problems can sometimes be resolved by modifying the grammar.**
- **On Friday, will discuss these conflicts and give some advice on how to resolve them.**

MP 6

- **MP 6 starts with a grammar embedded in an incomplete ocamlyacc specification. You will need to finish the spec:**
  - **Remove "extended BNF" productions - ocamlyacc cannot handle them**
  - **Resolve grammar conflicts**
  - **Fill in actions so as to produce ASTs.**