Lecture 9: Bottom-up parsing; ocamlyacc

- Using ocamlyacc
- Bottom-up parsing
  - Shift-reduce parsing
  - Bottom-up parsing as “handle-pruning”
- Midterm exam 1 next week!
  - Time & loc: Wednesday, Feb. 24, 7:30PM, 100 MSEB
  - Let us know about conflicts today!
  - Will post syllabus for exam, review questions, etc.
  - Class on Tuesday, Feb. 23, is optional review session.

Using ocamlyacc

- **Input 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 parsing function, with two arguments: a lexing function (lexer buffer to token) and a lexer buffer

- Each production has an associated semantic attribute — usually the CST or AST for that node. After the parser parses that part of the input using this production, it returns the semantic attribute.

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 - exprlex.mll

```
(* 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’] {tokenize lexbuf}
  | eof {EOL}
```

Example - exprparse.mly (part 1)

```
(* File: exprparse.mly *)
%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 - exprparse.mly (part 2)

```
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
- Typically used to give types and functions needed for the semantic values 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

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

ocamlyacc <*declarations*>

- `%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_value* }
  | ...
  | *symbol* ... *symbol* { *semantic_value* }
  ;
- Semantic values are arbitrary Ocaml expressions
- Must be of same type as declared (or inferred) for *nonterminal*
- In semantic values, access values of symbols by position: $1 for first symbol, $2 for second, etc.

Bottom-up parsing

- **Bottom-up parsing**
  - **Bottom-up vs. top-down**
  - **Shift-reduce parsing**
  - **Bottom-up parsing as “handle-pruning”**
  - **Parsing conflicts (will discuss in detail on Thursday)**

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:**

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

  Input: x + y * z

Bottom-up parsing

- **Bottom-up parsing works by creating small parse trees and joining them together into larger ones.**

- **Example:**
$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * id \mid id \end{array}$$

Input: x * y + z * w

Shift-reduce parsing

- Here's how bottom-up parsers work:
    - 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: When stack consists of just the start symbol, and 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:**
$$\begin{array}{lll} E & \rightarrow & E\ +\ T \mid T \\ T & \rightarrow & T\ *\ P \mid P \\ P & \rightarrow & id \mid int \end{array}$$

Input: x + 10 * y

(From now on, we will write down only the $roots$ **of the trees on the stack.)**

LR(1) parsing

- **Shift-reduce parsing is a general mechanism that can produce any parse tree, as long as the correct shift/reduce decision is made at every point.**

- **The big question is: based on the contents of the stack and the remaining input, what is the correct action?**

- **LR(1) parsing is a method of analyzing the grammar to determine how to make this decision. In LR(1) parsing, the s/r decision is based on the** *roots* **of the trees on the stack, and on** *one lookahead symbol***.**

- **LR(1) parsers are very efficient in practice.**

LR(1) parsing (cont.)

- Variants of LR(1) — such as "SLR(1)" and "LALR(1)" — are used because the required analysis can be done more efficiently.

- LR(1) is applicable only to grammars where this decision can be made unambiguously for any stack configuration and input symbol. This is not always possible.

- Note that in earlier shift/reduce examples, decision could be made mechanically by simple rules (see below).

- Our next problem is to understand when a grammar is *not* LR(1)...

Non-LR(1) example

Consider this grammar and two possible inputs:

- $$\begin{aligned} A &\rightarrow B \ , \ int \\ B &\rightarrow id \mid id \ , \ B \end{aligned}$$

- **Inputs: “x,10” and “x,y,10”**

Handle-pruning

- **Here is one way to think of bottom-up parsing:**
  - **Pretend you already know the parse tree. The shift-reduce process clips off, or "prunes," parts of the tree, until nothing is left. In particular, it prunes the children of one node.**
  - **A group of sibling nodes is eligible for pruning if they are all childless (either tokens or nonterminals all of whose children have already been pruned).**
  - **In LR parsing, pruning is always done at the *leftmost* eligible group of sibling nodes. This group is called the *handle* of the (partially pruned) parse tree.**

Example 1 of handle-pruning

Grammar: E $\rightarrow$ E,id | id

Input: x,y,z

Example 2 of handle-pruning

Consider this grammar and two possible inputs:

Grammar: $A \rightarrow B\ ,\ int$

$B \rightarrow id \mid id\ ,\ B$

Input: x,y,10

Example 3 of handle-pruning

Grammar: E $\rightarrow$ E+E | id

Input: x+y+z

Handle-pruning and s/r parsing

- **The connection between shift-reduce parsing and handle-pruning is this:** ***The stack contains the frontier of the pruned tree, up to but never exceeding the handle. The handle is exactly the symbols on the top of the stack that are reduced in a reduce action.***

- **In actual parsing, we don't have the parse tree, so we can only guess what it will look like in the end. The big question for bottom-up parsing is when to shift and when to reduce. Another way to say this is:** ***how do we know when we've seen the handle - i.e. that the handle is on the stack?***

- **In example 1 above, this was easy. In examples 2 and 3, it is not possible.**

LR(1) example

For this grammar, parsing decisions can always be made based on the stack and one lookahead symbol:

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

Thursday'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 Thursday, will discuss these conflicts and give some advice on how to resolve them.

MP 5

- MP 5 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.