CS 421 Lecture 9: Bottom-up Parsing

- Announcements
- OCaml self-help hints
- Lecture outline
  - Bottom-up parsing
  - ocamlyacc

Announcements

- MP4 has been posted
  - MiniJava lexer

- Reminder: midterm exam date – Thursday, July 2

OCaml self-help hints

- Consult the CS 421 resource guide:
  - http://www.cs.uiuc.edu/class/su09/cs421/
  - Use "Tips for using OCaml top level" to speed up working with the interactive environment
  - Consult the OCaml manual when you want a definitive answer about something
    - May be technical, not "user-friendly"
- Ask on the newsgroup
  - If you are having a problem, it's likely somebody has run into it already, or they will in the future.
- Ask Google
  - It probably knows…

OCaml self-help hints

- Be careful about
  - Data types, and type inference
  - Operator precedence
- Common OCaml error messages:
  - syntax error (underlined)
  - unbound value use (underlined)
  - Pattern matching is not exhaustive. Here is a counterexample:
    …
  - This expression has type <*type1*> but is here used with <*type2*>
    - Watch out especially for "unit"
  - <whatever error> in <*file*>.ml at line <*line*> characters <*chars*>

Top-down vs. bottom-up parsing

- Why is top-down called “top-down?”
  - As we consume tokens, we build a parse tree.
  - At any one time, we are filling in the children of a particular non-terminal.
  - *As soon as we decide which production to use*, we can fill in the tree.
  - In this sense, we are building the tree from the top (root) down (to the leaves).
    - Nature and Computer Science disagree on this point

Top-down parsing

- Example:
  - $E \rightarrow id \ \ T$
  - $T \rightarrow \varepsilon \mid + E \mid * E$

Input: x + y * z

Bottom-up parsing

- Works by creating small parse trees and joining them together into larger ones.
- Example: Input: x + y * z
  - $E \rightarrow id\ \ T$
  - $T \rightarrow \varepsilon \mid + E \mid * E$
- Start constructing trees, put them on stack:
  - Construct tree *x*: {x}
  - Add tree +: {*x*, +}
  - Add tree *y*: {*x*, +, *y*}
  - Add tree *: {*x*, +, y, *}
  - Add tree *z*: {*x*, +, *y*, *, *z*}

Bottom-up parsing (cont)

- Construct parse tree by merging:
  - $\{x, +, y, *, z\}$
  - Apply $T \rightarrow \varepsilon$
    - $\{x, +, y, *, z, T \rightarrow \varepsilon\}$
  - …

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 $A \to \alpha$: 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 the start symbol, look-ahead is EOF
  - Reject
- Bottom-up parsing is also called *shift-reduce parsing*

Shift-reduce example 1

- Example: Input: x ; y ; z
  - $L \rightarrow L \,;\, E \mid E$
  - $E \rightarrow id$

Action	Stack	Input
S		x ; y ; z
R $E \rightarrow id$	x	; y ; z
…		

Shift-reduce example 1

- Example:
  - $L \rightarrow L \,;\, E \mid E$
  - $E \rightarrow id$

<u>Action</u>	<u>Stack</u>	<u>Input</u>

Shift-reduce example 2

- Example: Input: x + 10 * y
  - $E \rightarrow E + T \mid T$
  - $T \rightarrow T * P \mid P$
  - $P \rightarrow id \mid int$

Action	Stack	Input
S		x + 10 * y
R $P \rightarrow id$	x	+ 10 * y
…		

Shift-reduce example 2

- Example:
  - $E \rightarrow E + T \mid T$
  - $T \rightarrow T * P \mid P$
  - $P \rightarrow id \mid int$

Action	Stack	Input

Bottom-up parsing

- This is hard!
  - How can we build a parser that works like this?
- Shift-reduce parsing is not usually done "by hand"
  - Automated parser generator tools
  - Generate parser code based on grammar specification
    - Similar to ocamllex and regular expressions for lexing
- Ocaml's parser generator is called ocamlyacc
  - "yet another compiler-compiler"

Using ocamlyacc

- Create grammar specification in a text file
  - <*grammar*>.mly
- Execute
  - ocamlyacc <*grammar*>.mly
- Produces
  - code for parser in <*grammar*>.ml
  - 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 (lexbuf -> token) and a lexbuf as arguments
  - Aside: we'll see more functions being passed around as arguments soon…
- Returns *semantic attribute* of corresponding entry point

Example – expression grammar

- We will take a simple expression grammar and create a parser to parse inputs and produce abstract syntax
- Grammar:
  - $M \to Exp\ eof$
  - $Exp \to Term \mid Term + Exp \mid Term - Exp$
  - $Term \to Factor \mid Factor * Term \mid Factor / Term$
  - $Factor \to id \mid (Exp)$
- 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'] {tokenize 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 "expparse.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 actions of rules and to give specialized error recovery
- May be omitted
- <*footer*> is 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 precedende; 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 the same type as declared (or inferred) for *nonterminal*
  - Access values semantic attributes of symbols by position: $1 for first symbol, $2 for second, *etc.*

Next class

- Finish up parsing (yay!)
- Big question: how to choose whether to shift or reduce?
  - ocamlyacc uses a method – called *LALR(1)* – to construct tables that say which 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.
    - We will discuss these conflicts and give some advice on how to resolve them.