CMSC330: Advanced Language Processing

Chris Kauffman

Last Updated:
Wed Oct 18 10:50:56 PM EDT 2023

Logistics

Exercise: Subtraction Trees

Consider these two parse trees for the given expression

```
let parsetree = parse_tokens (lex_string "10-2-3") in ...;
```

```
(* TREE A *)                  (* TREE B *)
Sub( Const 10,                Sub( Sub( Const 10,
     Sub( Const 2,                      Const 2),
          Const 3));;              Const 3);;
```

1. What are the arithmetic results of evaluating each of them?
2. Which do you expect to result from our previous parsers?
3. Which gives the "correct" result according to standard rules of arithmetic?

Answers: Subtraction Trees

Consider these two parse trees for the given expression

```
let parsetree = parse_tokens (lex_string "10-2-3") in ...;
```

```
(* TREE A *)                    (* TREE B *)
Sub( Const 10,                  Sub( Sub( Const 10,
     Sub( Const 2,                        Const 2),
          Const 3));;                Const 3);;
```

1. What are the arithmetic results of evaluating each of them?
   `A = 11, B = 5`
2. Which do you expect to result from our previous parsers?
   A has been the standard behavior of parsers from lecture lab
3. Which gives the "correct" result according to standard rules of arithmetic?
   B is the standard interpretation for arithmetic with left-to-right evaluation

Right Associativity vs Left Associativity in Parsing

- Chained operators like `1+2+3+4` have so far yielded "right-heavy" trees: the right branch grows deeply
- This is appropriate for **right associative** operators (like raising to a power) and commutative operators (order independent) operators like addition and multiplication BUT…
- Familiar operators like Subtraction and Division are **left associative**:

$$10 - 3 - 2 - 1 \equiv (((10 - 3) - 2) - 1) = 4$$

- Leads to several irritations:
  - Can address this in a CFG but…
  - Recursive descent parsers require special care reflect left-associativity in the trees they generate

Exercise: Compare Right/Left Associative Parsers

```
(* left_right_assoc.ml: *)

(* right associativity via standard recursion *)
and parsesub_right toks =
  let (lexpr, rest) = parse_num toks in         (* try higher prec first      *
  match rest with
  | Minus :: tail ->                            (* found -                    *
     let (rexpr,rest) = parsesub_right tail in  (* recursively gen right side *
     (Sub(lexpr,rexpr), rest)                   (* subtract left / right      *
  | _ -> (lexpr, rest)                          (* not a sub                  *

(* left associativity via iteration *)
and parsesub_left toks =
  let (lexpr, rest) = parse_num toks in         (* try higher prec first      *
  let rec iter lexpr toks =                     (* loop over adjacent exprs   *
    match toks with
    | Minus :: rest ->                          (* found -                    *
       let (rexpr,rest) = parse_num rest in     (* try higher prec            *
       iter (Sub(lexpr,rexpr)) rest             (* left Sub, iterate again    *
    | _ -> (lexpr, toks)
  in
  iter lexpr rest                               (* start iterating            *
```

Answers: Compare Right/Left Associative Parsers

- Right-associative recurses deeply to the right to generate right hand expression
- Left-associative iterates consuming subtraction expressions in a (tail recursive) loop
- Left-associative creates a left-heavy tree by combining right and left expressions in a `Sub` then passing it forward in the iteration to become the left branch

Token Streams and Buffering

- So far have assumed that Lexer tokenizes the entire input string prior to starting the parser
- This works for small inputs, but for large files may be inefficient
  - May need to store entire input (file) in memory during lexing
  - Must store entire token list in memory during parsing
- Real world lexer/parsers make this more efficient via a **lexing buffer**

- Lexing buffer stores only part of file and lexing **stream**
- API to see `next()` token and `consume()` it
- Frequently seen in interpreter and compiler tutorials

```
// imperative pseudocode for
// parsing add/sub expressions
// uses a lexing buffer
global lexbuf;
function parse_addsub(){
   var lexpr := parse_muldiv()
   while lexbuf.next() = "+" or "-"
      var op := lexbuf.next()
      lexbuf.consume()
      var rexpr := parse_muldiv()
      lexpr := make_tree(op,lexpr,rexpr)
   return lexpr
}
```

Lexing and Parsing Tools

- Generally do NOT want to write large-scale programs in assembly language: too many things can go wrong
- Generally do NOT want to write lexers/parsers by hand for large-scale languages: too many things can go wrong
- High-level programming languages improve over assembly through a **compiler** or interpreter: translate high-level code to low
- **Lexer/Parser Generators** improve over hand-written parser generators: translate high-level grammars to low-level code
- **Lex and Yacc**[1] are the classic tools to generate lexer/parsers
- Usually involves two input files
  1. Parser input to Yacc describes token kinds, grammar, actions
  2. Lexer input to Lex describes how characters translate to tokens
- Result in compilable code with built-in lexing buffer and efficient grammar recognition through finite automata

[1] Yacc is short for *Yet Another Compiler Compiler* as it is often used to generate the front-end of a compiler

OCaml Lex and Yacc

- OCaml comes with standard tools for language processing
  - `ocamllex`: lexer generator
  - `ocamlcyacc`: parser generator
- Input has special syntax, **not all normal OCaml**
- Will briefly survey these to get a flavor for them but Lex/Yacc are worth further study if you are interested in constructing programming languages
- Couch this in discussion a calculator language `arith` which is part of the code pack

```
> cd arith/
> make
...
> ./arithmain
arithmain> 1+1
2
arithmain> 5*9-2
43
arithmain> 10-3-2
5
```

OCaml Lex Input

- Simple structure mainly used to set up a rule for token kinds
- Has dependency on `arithparse.ml` for token kinds

```
(* arithlex.mll : OCaml lex source file *)

(* First section is raw ocaml between curlies *)
{
  open Arithparse;;           (* bring in token types from arithparse.mli *)
  exception Eof;;             (* declare exception type for end of file *)
}

(* second section defines how the lexer works *)
rule token = parse
 | [' ' '\t']         { token lexbuf }             (* skip recursing *)
 | ['\n' ]            { EOL }
 | ['0'-'9']+ as lxm { INT(int_of_string lxm) }   (* regex for numbers *)
 | '+'                { PLUS }
 | '-'                { MINUS }
 | '*'                { TIMES }
 | '/'                { SLASH }
 | '('                { OPAREN }
 | ')'                { CPAREN }
 | eof                { raise Eof }                (* end of file *)
```

OCaml Yacc Input 1

- Two main sections, first is shown
- Declares token types and main entry into parser

```
/* arithparse.mly: ocaml yacc sourc file defining a parser. Note the
   C-style comments rather than OCaml style */

/* first section defines token types used by parser using % directives */
%token <int> INT
%token PLUS MINUS TIMES SLASH
%token OPAREN CPAREN EOL

%type <int> main       /* type returned by production main */
%start main            /* entry production for parser */

/* end first section */
%%
```

OCaml Yacc Input 2

- Second section shows grammar **productions**
- Curlies to the right have **actions** associated with productions
- **Dollar variables** correspond to results of recursive grammar elements

```
...
%%
/* second section which shows expressions */
main:                                      /* initial production          */
  | plusminus EOL          { $1 }          /* $1 is result of plusminus   */
;
plusminus:                                 /* addition and subtraction    */
  | muldiv                 { $1 }          /* could be just mul/div       */
  | plusminus PLUS  muldiv { $1 + $3 }     /* or an addition              */
  | plusminus MINUS muldiv { $1 - $3 }     /* or a subtraction            */
;
muldiv:                                    /* multiplication and division */
  | ident                  { $1 }          /* could be just an ident      */
  | muldiv TIMES ident     { $1 * $3 }     /* or a multiplication         */
  | muldiv SLASH ident     { $1 / $3 }     /* or a division               */
;
ident:                                     /* identifier                  */
  | INT                     { $1 }         /* integer constant            */
  | OPAREN plusminus CPAREN { $2 }         /* opening parenthesis         */
;
```

A Main Function

```ocaml
(* arithmain.ml: main routine for lexing/parsing and interpreting an
   arithmetic language. This version directly interprets the language
   rather than building an expression tree.  *)
open Printf;;

let _ =
  try
    (* Lexing is an OCaml standard module for lexer support. Next line
       creates a lexing buffer. *)
    let lexbuf = Lexing.from_channel stdin in

    while true do                 (* loop over input until end of file *)
      printf "arithmain> %!";     (* print prompt *)

      (* Arithlex.token is a function that produces a token.
         Arithparse.main  function takes a token producer and a lexbuf.
         The next line lexes and parses an expression. *)
      let result = Arithparse.main Arithlex.token lexbuf in

      printf "%d\n%!" result;     (* print integer result *)

    done;                         (* end of input loop *)

  with Arithlex.Eof ->            (* eof exception pops out of loop *)
    printf "That's all folks!\n";
;;
```

Compiling Gets Complicated

- Compiling with lex/yacc is tricky as several functions like `Arithparse.main` defined based on grammar production rules
- Also compile order is tricky, best to put build sequence into a `Makefile` or other build system

```
 1 > make
 2 ocamllex arithlex.mll        # creates arithlex.ml
 3 11 states, 267 transitions, table size 1134 bytes
 4 ocamlyacc arithparse.mly     # creates arithparse.ml / arithparse.mli
 5 ocamlc -g -c arithparse.mli  # required by arithlex.ml
 6 ocamlc -g -c arithlex.ml     # required by arithparse.ml
 7 ocamlc -g -c arithparse.ml
 8 ocamlc -g -c arithmain.ml    # requires arithlex.cmo and arithparse.cmo
 9 ocamlc -g -o arithmain arithlex.cmo arithparse.cmo arithmain.cmo
10
11 > ./arithmain
12 arithmain> 1+3*2-4
13 3
```

- Note report on line 3: lexing statistics for finite automata generated which will recognize tokens
- `arthlex.ml` and `arithparse.ml`: valid OCaml but **machine generated code**, not meant for human eyes

Generating Parse Trees in Lex/Yacc

- `arith/` system directly interprets input during parsing through grammar actions as in

```
plusminus:                                /* addition and subtraction
  | muldiv                    { $1 }      /* could be just mul/div
  | plusminus PLUS  plusminus { $1 + $3 } /* or an addition
  | plusminus MINUS plusminus { $1 - $3 } /* or a subtraction
;
```

- This is typical of **interpreters** perform no further transformations or optimizations on the code
- Code pack include `arith-tree/` which changes this to create a data structure instead via code like

```
plusminus:                                     /* addition and subtraction
  | muldiv                    { $1 }           /* could be just mul/div */
  | plusminus PLUS  plusminus { Add($1,$3) } /* or an addition */
  | plusminus MINUS plusminus { Sub($1,$3) } /* or a subtraction */
;
```

- Resulting parse tree is captured in a main routine for printing, transformation, and evaluation
- Typical of a **compiler** or at least more sophisticated interpreter

How do other languages do it?

- OCaml and Lisp excel at **symbolic computation**: manipulating data like expression trees and token sequences
- OCaml makes it easy to declare new types of data that are algebraic with variants: very well suited for symbolic processing
- Lisp has untyped symbols built in, as easy as quoting as in the code `'add` is a symbol with name “add”
- Languages like C, Java, Python are a bit clunkier for symbolic processing
  - Symbols aren't innate in any of them: with string constants, **enumerations**, classes can emulate them
  - Takes more work and more lines of code than OCaml/Lisp mechanisms
- Also, none of these have standard lexer/parser generators (though many libraries exist for them)

Contrast: Symbolic Data in Java vs OCaml

- As a sample, today's code pack contains equivalent versions of the arithmetic langauge in OCaml and Java
- Both of these
  - Accept the same language like `1+2*3-12/4`
  - Use lexer/parser generators to specify high-level language
  - Accept user input on command line or interactively
  - Create an expression tree data structure
  - Print the data structure to the screen
- OCaml version uses `ocamllex / ocamlparse`,
  - Build 6 files $\rightarrow$ 17 files
- Java version uses ANTLR4 parser generator library
  - Build 5 files $\rightarrow$ 35 files

Contrast Stats: Symbolic Data in Java vs OCaml

OCaml `arith-tree/`

File	LOC	Purpose
`arithlex.mll`	15	Lexer definition
`arithparse.mly`	26	Grammar definition
	41	Subtotal
`arithexpr.ml`	33	Tree data type and printing
`arithmain.ml`	13	Main function for interactive input loop, printing
	88	Total Lines of Code

Java `arith-java/`

File	LOC	Purpose
`TokenType.java`	15	Declare token types with string names
`Arith.g`	34	Grammar file for ANTLR4
	49	Subtotal
`ArithMain.java`	123	Main routine, tree data, interface code, printing
	172	Total Lines of Code

- Not interactive: just parses command line arg and prints tree as
- Most of the code is interface glue matching classes to parse tree types via Visitor Pattern implementations
- Mostly due to Java classes not fitting expression trees as well as algebraic variants: classes are the **only way** to represent data in Java

Summary

- Writing lexers/parsers is hard, riddled with issues like left/right associativity
- Make life easier by employing a lexer/parser generator
- OCaml is well-suited for symbolic data processing via data type mechanisms and built-in data structures

  Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.
  – *Greenspun's Tenth Rule*