ml-yacc

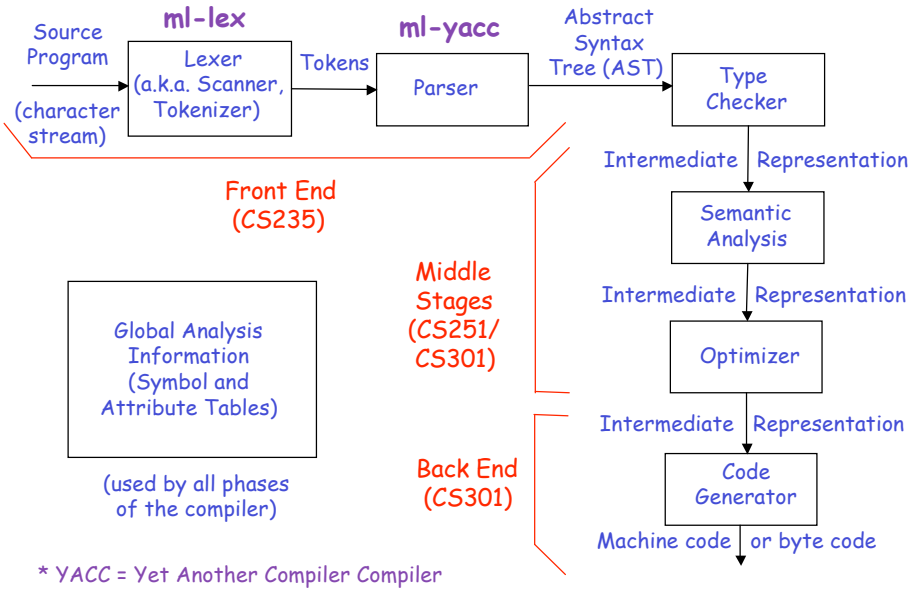
A Tool For Automatically Constructing LALR(1) Parsers

Monday, November 24, 2008
Reading: Appel 3.3

CS235 Languages and Automata

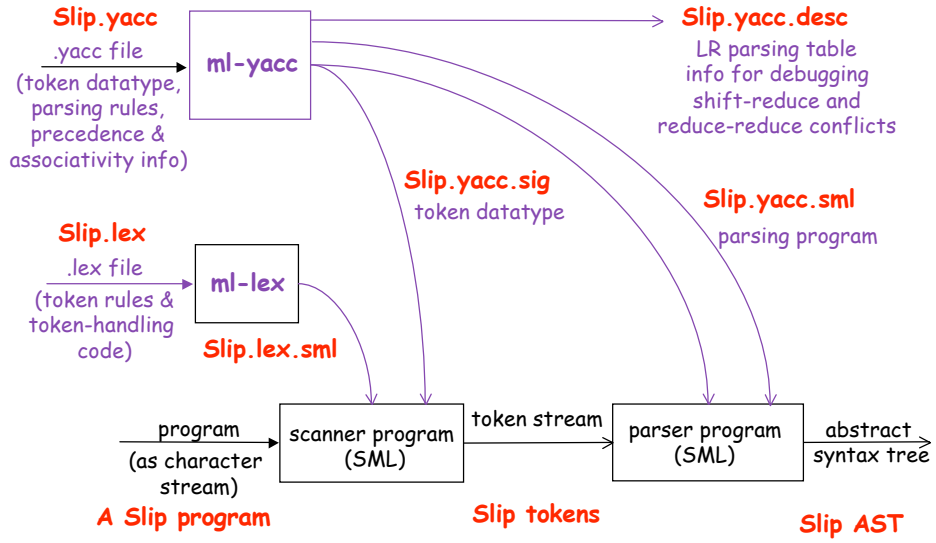
Department of Computer Science
Wellesley College

Compiler Structure



* YACC = Yet Another Compiler Compiler

ml-yacc: A Parser Generator



Format of a .yacc File

Header section with SML code
%%
Specification of terminals & variables and various declarations
(including precedence and associativity)
%%
Grammar productions with semantic actions

IntexpUnambiguousAST.yacc (Part 1)

```
(* no SML code in this case *)
%% (* separates initial section from middle section of .yacc file *)
%name Intexp
%pos int
%term
   INT of int
 | ADD | SUB | MUL | DIV | EXPT
 | LPAREN (* "(" *)
 | RPAREN (* ")" *)
 | EOF  (* End of file*)
%nonterm
   Start of AST.exp
 | Exp of AST.exp
 | Term of AST.exp
 | Factor of AST.exp
 | Unit of AST.exp
%start Start

(* Middle section continued on next slide *)
```

Name used to prefix various names created by ml-yacc

Declares the type of positions for terminals. Each terminal has a left and right position. Here, position is just an int (a character position in a string or file)

Specifies the terminals of the language. ml-yacc automatically constructs a Tokens module based on this specification.
Tokens specified without "of" will have a constructor of two args: (1) its left position and (2) its right position.
Tokens specified with "of" will have a constructor of three args: (1) the component datum (whose type follows "of"); (2) its left position; and (3) its the right position.

Specifies the non-terminals of the language and the kind of value that the parser will generate for them (an AST in this case).
The "start" non-terminal should not appear on the right-hand sides of any productions.

Start symbol of the grammar

IntexpUnambiguousAST.yacc (Part 2)

```
(* Middle section continued from previous slide *)
%keyword
%eop EOF
%noshift EOF
%nodefault
%verbose
%value INT(0)
```

Lists the keywords of the language. The error handler of the ml-yacc generated parser treats keywords specially. In this example, there aren't any keywords.

Indicates tokens that end the input

Tells the parser never to shift the specfied tokens

Suppresses generation of default reduction

Generates a description of the LALR parser table in *filename*.yacc.desc. This is helpful for debugging shift/reduce and reduce/reduce conflicts

Specifies default values for value-bearing terminals. Terminals with default values may be created by an ml-yacc-generated parser as part of error-correction.

Note; in this middle section, you can also specify associativity and precedence of operators. See IntexpPrecedence.yacc (later slide) for an example.

```
(* Grammar productions and semantic actions on next slide *)
```

IntexpUnambiguousAST.yacc (Part 3)

```
%%  (* separates middle section from final section *)
(* Grammar productions and associated semantic actions (in parens) go here *)

 Start: Exp (Exp)

 Exp : Term (Term)
    (* The following rules specify left associativity *)
    | Exp ADD Term (AST.BinApp(AST.Add,Exp,Term))
    | Exp SUB Term (AST.BinApp(AST.Sub,Exp,Term))

 Term : Factor (Factor)
    (* The following rules specify left associativity *)
    | Term MUL Factor (AST.BinApp(AST.Mul,Term,Factor))
    | Term DIV Factor (AST.BinApp(AST.Div,Term,Factor))

 Factor : Unit (Unit)
        (* The following rule specifies right associativity *)
      | Unit EXPT Factor (AST.BinApp(AST.Expt,Unit,Factor))

 Unit : INT  (AST.Int(INT))
     | LPAREN Exp RPAREN (Exp)
```

Within parens, nonterminals stand for the value generated by the parser for the nonterminal, as specified in the %nonterm declaration (an AST expression in this case).

Within parens, terminals stand for the component datum specified after "of" in the %term declaration. E.g., the datum associated with INT is an integer.

AST module for integer expressions

```
structure AST =

struct

  datatype exp = Int of int
               | BinApp of binop * exp * exp

  and binop = Add | Sub | Mul | Div | Expt

  (* Various AST functions omitted from this slide *)

end
```

Interfacing with the Lexer

```
(*contents of Intexp.lex *)
open Tokens

type pos = int
type lexresult= (svalue,pos) token

fun eof () = Tokens.EOF(0,0)

fun integer(str,lexPos) =
   case Int.fromString(str) of
     NONE => raise Fail("Shouldn't happen: sequence of digits not recognized as integer -- " ^ str)
   | SOME(n) => INT(n,lexPos,lexPos)

(* For scanner initialization (not needed here) *)
fun init() = ()
%%
%header (functor IntexpLexFun(structure Tokens: Intexp_TOKENS));
alpha=[a-zA-Z];
digit=[0-9];
whitespace=[\ \t\n];
symbol=[+*^()\[];
any= [^];
%%
{digit}+  => (integer(yytext,yypos));
"+" => (ADD(yypos,yypos));
(* more lexer rules go here *)
```


Parser interface code, Part 1

```
structure Intexp = struct  (* Module for interfacing with parser and lexer *)

(* Most of the following is "boilerplate" code that needs to be slightly edited on a per parser basis. *)

 structure IntexpLrVals = IntexpLrValsFun(structure Token = LrParser.Token)
 structure IntexpLex = IntexpLexFun(structure Tokens = IntexpLrVals.Tokens);
 structure IntexpParser =
     Join(structure LrParser = LrParser
          structure ParserData = IntexpLrVals.ParserData
          structure Lex = IntexpLex)

 val invoke = fn lexstream => (* The invoke function invokes the parser given a lexer *)
   let val print_error = fn (str,pos,_) =>
               TextIO.output(TextIO.stdOut,
                   "***Intexp Parser Error at character position "  ^ (Int.toString pos)
                   ^ "***\n" ^ str^ "\n")
   in IntexpParser.parse(0,lexstream,print_error,())
   end

 fun newLexer fcn = (* newLexer creates a lexer from a given input-reading function *)
      let val lexer = IntexpParser.makeLexer fcn
          val _ = IntexpLex.UserDeclarations.init()
       in lexer
      end

 (* continued on next slide *)
```

Parser interface code, Part 2

```
(* continued from previous slide *)
fun stringToLexer str = (* creates a lexer from a string *)
    let val done = ref false
     in newLexer (fn n => if (!done) then "" else  (done := true; str))
    end

fun fileToLexer filename = (* creates a lexer from a file *)
    let val inStream = TextIO.openIn(filename)
     in newLexer (fn n => TextIO.inputAll(inStream))
    end

fun lexerToParser (lexer) = (* creates a parser from a lexer *)
   let val dummyEOF = IntexpLrVals.Tokens.EOF(0,0)
            val (result,lexer) = invoke lexer
            val (nextToken,lexer) = IntexpParser.Stream.get lexer
    in if IntexpParser.sameToken(nextToken,dummyEOF) then
         result
       else (TextIO.output(TextIO.stdOut,
                           "*** INTEXP PARSER WARNING -- unconsumed input ***\n");
            result)
   end

val parseString = lexerToParser o stringToLexer (* parses a string *)

val parseFile = lexerToParser o fileToLexer (* parses a file *)

end (* struct *)
```


Putting it all together: the load file

```
(* This is the contents of load-intexp-unambiguous-ast.sml *)
CM.make("$/basis.cm");          (* loads SML basis library *)
CM.make("$/ml-yacc-lib.cm");  (* loads SML YACC library *)
use "AST.sml"; (* datatype for integer expression abstract syntax trees *)
use "IntexpUnambiguousAST.yacc.sig";   (* defines Intexp_TOKENS
                                          and other datatypes *)
use "IntexpUnambiguousAST.yacc.sml";  (* defines shift-reduce parser *)

use "Intexp.lex.sml";   (* load lexer *after* parser since it uses
                           tokens defined by parser *)
use "Intexp.sml"; (* load top-level parser interface *)
Control.Print.printLength := 1000;   (* set printing parameters so that     *)
Control.Print.printDepth  := 1000;   (*  we'll see all details of parse trees *)
Control.Print.stringDepth := 1000;  (*   and strings  *)
open Intexp;  (* open the parsing module so that we can use parseString
                 and parseFile without qualification. *)
```

Taking it for a spin

```
[fturbak@cardinal intexp-parsers] ml-lex Intexp.lex

Number of states = 14
Number of distinct rows = 3
Approx. memory size of trans. tabl
e = 387 bytes
[fturbak@cardinal intexp-parsers] ml-yacc IntexpUnambiguousAST.yacc
[fturbak@cardinal intexp-parsers]
```

Linux shell

```
- use "load-intexp-unambiguous-ast.sml";
(* … lots of printed output omitted here … *)

- parseString "1+2*3^4^5*6+7";
val it =
  BinApp
   (Add,
    BinApp
     (Add,Int 1,
      BinApp
       (Mul,
        BinApp (Mul,Int 2,BinApp (Expt,Int 3,BinApp (Expt,Int 4,Int 5))),
        Int 6)),Int 7) : IntexpParser.result
```

SML intepreter

IntexpUnambiguousAST.yacc.desc

```
state 0:

        Start : . Exp

        INT             shift 6
        LPAREN          shift 5

        Start           goto 19
        Exp             goto 4
        Term            goto 3
        Factor          goto 2
        Unit            goto 1

        .               error


state 1:

        Factor : Unit .  (reduce by rule 7)
        Factor : Unit . EXPT Factor

        ADD             reduce by rule 7
        SUB             reduce by rule 7
        MUL             reduce by rule 7
        DIV             reduce by rule 7
        EXPT            shift 7
        RPAREN          reduce by rule 7
        EOF             reduce by rule 7


        .               error
```

```
… lots of states omitted …

state 18:

        Unit : LPAREN Exp RPAREN .  (reduce by rule 10)

        ADD             reduce by rule 10
        SUB             reduce by rule 10
        MUL             reduce by rule 10
        DIV             reduce by rule 10
        EXPT            reduce by rule 10
        RPAREN          reduce by rule 10
        EOF             reduce by rule 10


        .               error


state 19:


        EOF             accept


        .               error

72 of 104 action table entries left after compaction
21 goto table entries
```

IntexpUnambiguousCalc.yacc

We can calculate the result of an integer expression rather than generating an AST.

```
(* An integer exponentiation function *)
fun expt (base,0) = 1
  | expt(base,power) =
      base*(expt(base,power-1))


%%

(* Only changed parts shown *)

%nonterm
    Start of int
  | Exp of int
  | Term of int
  | Factor of int
  | Unit of int

)
```

```
%%

 Start: Exp (Exp)

Exp : Term (Term)
    (* The following rules specify left associativity *)
    | Exp ADD Term (Exp + Term)
    | Exp SUB Term (Exp - Term)

Term : Factor (Factor)
    (* The following rules specify left associativity *)
    | Term MUL Factor (Term * Factor)
    | Term DIV Factor (Term div Factor)
          (* div is integer division *)

Factor : Unit (Unit)
       (* The following rule specifies right associativity *)
      | Unit EXPT Factor (expt(Unit,Factor))

Unit : INT  (INT)
     | LPAREN Exp RPAREN (Exp)
```


Testing IntexpUnambiguousCalc

```
[fturbak@cardinal intexp-parsers] ml-yacc IntexpUnambiguousCalc.yacc
[fturbak@cardinal intexp-parsers]
```

Linux shell

```
- use "load-intexp-unambiguous-calc.sml";
(* … lots of printed output omitted here … *)

- parseString "1+2*3";
val it = 7 : IntexpParser.result

- parseString "1-2-3";
val it = ~4 : IntexpParser.result
```

SML intepreter

IntexpAmbiguousAST.yacc

We can use an ambiguous grammar.
(In this case, it leads to shift/reduce conflicts.)

```
%%
(* Only changed parts shown *)

%nonterm
   Start of AST.exp
  | Exp of AST.exp
)

%%

 Start: Exp (Exp)

 Exp : INT  (AST.Int(INT))
     | Exp ADD Exp (AST.BinApp(AST.Add,Exp1,Exp2))
     | Exp SUB Exp (AST.BinApp(AST.Sub,Exp1,Exp2))
     | Exp MUL Exp (AST.BinApp(AST.Mul,Exp1,Exp2))
     | Exp DIV Exp (AST.BinApp(AST.Div,Exp1,Exp2))
     | Exp EXPT Exp (AST.BinApp(AST.Expt,Exp1,Exp2))
```


IntexpAmbiguousAST.yacc.desc

```
25 shift/reduce conflicts

error:  state 8: shift/reduce conflict (shift EXPT, reduce by rule 6)
error:  state 8: shift/reduce conflict (shift DIV, reduce by rule 6)
error:  state 8: shift/reduce conflict (shift MUL, reduce by rule 6)
error:  state 8: shift/reduce conflict (shift SUB, reduce by rule 6)
error:  state 8: shift/reduce conflict (shift ADD, reduce by rule 6)
error:  state 9: shift/reduce conflict (shift EXPT, reduce by rule 5)
error:  state 9: shift/reduce conflict (shift DIV, reduce by rule 5)
error:  state 9: shift/reduce conflict (shift MUL, reduce by rule 5)
error:  state 9: shift/reduce conflict (shift SUB, reduce by rule 5)
error:  state 9: shift/reduce conflict (shift ADD, reduce by rule 5)
error:  state 10: shift/reduce conflict (shift EXPT, reduce by rule 4)
error:  state 10: shift/reduce conflict (shift DIV, reduce by rule 4)
error:  state 10: shift/reduce conflict (shift MUL, reduce by rule 4)
error:  state 10: shift/reduce conflict (shift SUB, reduce by rule 4)
error:  state 10: shift/reduce conflict (shift ADD, reduce by rule 4)
error:  state 11: shift/reduce conflict (shift EXPT, reduce by rule 3)
error:  state 11: shift/reduce conflict (shift DIV, reduce by rule 3)
error:  state 11: shift/reduce conflict (shift MUL, reduce by rule 3)
error:  state 11: shift/reduce conflict (shift SUB, reduce by rule 3)
error:  state 11: shift/reduce conflict (shift ADD, reduce by rule 3)
error:  state 12: shift/reduce conflict (shift EXPT, reduce by rule 2)
error:  state 12: shift/reduce conflict (shift DIV, reduce by rule 2)
error:  state 12: shift/reduce conflict (shift MUL, reduce by rule 2)
error:  state 12: shift/reduce conflict (shift SUB, reduce by rule 2)
error:  state 12: shift/reduce conflict (shift ADD, reduce by rule 2)

... state descriptions omitted ...
```

IntexpPrecedenceAST.yacc

We can resolve conflicts with precedence/associativity declarations

```
%%
(* Only changed parts shown *)

%nonterm
   Start of AST.exp
  | Exp of AST.exp
)

(* Specify associativity and precedence from low to high *)
%left ADD SUB
%left MUL DIV
%right EXPT

%%

 Start: Exp (Exp)

 Exp : INT  (AST.Int(INT))
     | Exp ADD Exp (AST.BinApp(AST.Add,Exp1,Exp2))
     | Exp SUB Exp (AST.BinApp(AST.Sub,Exp1,Exp2))
     | Exp MUL Exp (AST.BinApp(AST.Mul,Exp1,Exp2))
     | Exp DIV Exp (AST.BinApp(AST.Div,Exp1,Exp2))
     | Exp EXPT Exp (AST.BinApp(AST.Expt,Exp1,Exp2))
```