MP 7 – A Parser for PicoML

CS 421 – Summer 2013

Revision 1.1

Assigned July 5, 2013
Due July 12, 2013, 11:59 PM
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Added mention of the `%prec` directive in Section 4.

2 Overview

In this MP, we will deal with the process of converting PicoML code into an abstract syntax tree using a parser. We will use the *occamlyacc* tool to generate our parser from a description of the grammar. This parser will be combined with the lexer and type inferencer from previous MPs to make an interactive PicoML parser and type checker, where you can type in PicoML expressions and see a proof tree of the expression's type:

```
Welcome to the Student parser

> val x = 4;
val x : int


final environment:

{ x : int }

proof:
  {  } |= val x = 4 :: { x : int }
  |--{  } |= 4 : int
```

To complete this MP, you will need to be familiar with describing languages with BNF grammars, adding attributes to return computations resulting from the parse, and expressing these attribute grammars in a form acceptable as input to *ocamlyacc*.

3 Given Files

mp7-skeleton.mly: You should copy the file **mp7-skeleton.mly** to **mp7.mly**. The skeleton contains some code to get you started.

picomlIntPar.ml: This file contains the main body of the PicoML executable. It essentially connects your lexer, parser, and type inference code and provides a friendly prompt to enter PicoML expressions.

picomllex.ml: This file contains the ocamllex specification for the lexer. It is a modest expansion to the lexer you wrote for MP7.

mp7common.ml: This file includes the types for expressions, as well as the type inferencing code. Appropriate code from this file will automatically be called by the interactive loop defined in **picomlIntPar.ml**.

4 Overview of `ocamlyacc`

Take a look at the given **mp7-skeleton.mly** file. The grammar specification has a similar layout to the lexer specification of MP6. It begins with a header section (where you can add raw OCaml code), then has a section for directives (these start with a `%` character), then has a section that describes the grammar (this is the part after the `%%`). You will only need to add to the last section. The slides on LR Parsing include an example.

Recall from lecture that the process of transforming program code (i.e, ASCII text) into an *abstract syntax tree* (AST) has two parts. First, the *lexical analyzer* (lexer) scans over the text of the program and converts the text into a sequence of *tokens*. The type of tokens in general may be a preexisting OCaml type, or a user-defined type created for the purpose. In the case where *ocamlyacc* is used, the type is named `token` and the datatype `token` is created implicitly by the `%token` directives. These tokens are then fed into the *parser* created by rules in your input, which builds the actual AST.

Precedence and associativity are the mechanisms by which a grammar is disambiguated. In *ocamlyacc*, precedence is used in the following way to resolve reduce/reduce and shift/reduce conflicts:

- Tokens and rules have precedences. By default, the precedence of a rule is the precedence of its rightmost terminal. You can override this default by using the `%prec` directive in the rule. If you write `%prec` before a symbol in a production, the precedence of the production becomes the precedence of that symbol. See the `ocamlyacc` documentation for examples.
- A reduce/reduce conflict is resolved in favor of the first rule (in the order given by the source file), and `ocamlyacc` outputs a warning.
- A shift/reduce conflict is resolved by comparing the precedence of the rule to be reduced with the precedence of the token to be shifted. If the precedence of the rule is higher, then the rule will be reduced; if the precedence of the token is higher, then the token will be shifted.
- A shift/reduce conflict between a rule and a token with the same precedence will be resolved using the associativity: if the token is left-associative, then the parser will reduce; if the token is right-associative, then the parser will shift. If the token is non-associative, then the parser will declare a syntax error.
- When a shift/reduce conflict cannot be resolved using the above method, then `ocamlyacc` will output a warning and the parser will always shift.

If your specification gives any warnings about shift/reduce or reduce/reduce conflicts, it is likely to fail on some examples, so please try to eliminate all conflicts.

4.1 More Information

Here is a website you should check out if you would like more information on the usage of *ocamlyacc*:

- `http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html`

5 Compiling

5.1 Running PicoML

The given `Makefile` builds executables called `picomlIntPar` and `picomlIntParSol`. The first is an executable for an interactive loop for the parser built from your solution to the assignment, and the second is one built from the reference solution. If you run `./picomlIntPar` or `./picomlIntParSol`, you will get an interactive prompt. You can type in PicoML expressions followed by a double semicolon, and they will be parsed and their types inferred and displayed:

```
Welcome to the Solution parser

> 3;;
val _ : int

final environment:

{}

proof:

  {} |= 3 : int

> let x = 3 + 4;;
val x : int

final environment:

{x : int}

proof:

  {} |= let x = 3  +  4 in x : int
  |--{} |= 3  +  4 : int
  | |--{} |= 3 : int
  | |--{} |= 4 : int
  |--{x : int} |= x : int

> let f = fun y -> y * x;;
val f : int -> int

final environment:

{f : int -> int,x : int}

proof:

  {x : int} |= let f = fun y -> y  *  x in f : int -> int
  |--{x : int} |= fun y -> y  *  x : int -> int
  | |--{y : int,x : int} |= y  *  x : int
  |   |--{y : int,x : int} |= y : int
  |   |--{y : int,x : int} |= x : int
```

```
  |--{f : int -> int,x : int} |= f : int -> int

> f 5;;
val _ : int

final environment:

{f : int -> int,x : int}

proof:

  {f : int -> int,x : int} |= f 5 : int
  |--{f : int -> int,x : int} |= f : int -> int
  |--{f : int -> int,x : int} |= 5 : int
```

Note that your output might have different type variables than shown in the examples; this isn't an error. Notice the accumulation of values in the (type) environment as expressions are entered. To reset the environment, you must quit the program (with CTRL+C) and start again.

6 Problem Setting

The grammar below for PicoML is ambiguous. You are also provided with a table listing the associativity/precedence attributes of the various language constructs. You will be required to use the information given in this table in order to create an *ocamlyacc* specification that generates the same language as the given one, but that is unambiguous and enforces the constructs to be specified as in the table.

The concrete syntax of PicoML that you will need to parse is the following:

```
<main> ::= <exp> ;;
         | let IDENT = <exp> ;;
         | let rec IDENT = <exp> ;;

<exp> ::= IDENT
         | BOOL | INT | FLOAT | STRING | UNIT
         | ( <exp> )
         | ( <exp> , <exp> )
         | let IDENT = <exp> in <exp>
         | let rec IDENT IDENT = <exp> in <exp>
         | <exp> <binop> <exp>
         | <monop> <exp>
         | <exp> && <exp>
         | <exp> || <exp>
         | [ ]
         | [ <list_contents> ]   /* sugar for non-empty lists, extra credit */
         | if <exp> then <exp> else <exp>
         | <exp> <exp>
         | fun IDENT -> <exp>
         | raise <exp>
         | try <exp> with n1 -> e1 | ...   /* extra credit */

<binop> :: = + | - | * | / | +. | *. | /. | ^ | ** | < | > | = | >= | <=
```

```
<monop> ::= print_int | head | tail | ~

<list_contents> ::= <nonempty sequence of expressions separated by double colons>
```

IDENT refers to an identifier token (only one token, takes a string as argument). <binop> refers to some infix identifier token (one for each infix operator). Similarly, <monop> are the unary operators. The nonterminals in this grammar are main, exp, binop, monop, and list_contents, with main being the start symbol.

The rest of the symbols are terminals, and their representations in OCaml are elements of the type token, defined at the beginning of the file mp7.mly. Our OCaml representation of terminals is not always graphically identical to the one shown in the above gramar; we have used concrete syntax in place of tokens for the terminals. For example, :: is represented by DCOLON and + by PLUS. Our OCaml representation of the identifier tokens (IDENT) is achieved by the constructor IDENT that takes a string and yields a token, as constructed by the lexer from MP7.

Recall that identifying the tokens of the language is the job of lexer, and the parser (that you have to write in this MP) takes as input a *sequence of tokens*, such as (INT 3) PLUS (INT 5), and tries to make sense out of it by transforming it into an abstract syntax tree, in this case BinExp(Plus, ConExp(Int 3), ConExp(Int 5)). The abstract syntax trees into which you have to parse your sequences of tokens are given by the following OCaml types ("metatypes", as opposed to PicoML types), present in the file mp7common.ml:

```
type const = Bool of bool | Int of int | Float of float | String of string
           | Nil | Unit

type binop = Add | Sub | Mul | Div | Exp
           | FAdd | FSub | FMul | FDiv
           | Concat | Cons | Comma | Eq | Less

type monop = Head | Tail | Print | Neg | Fst | Snd

type exp =
   | VarExp of string                (* variables *)
   | ConExp of const                 (* constants *)
   | IfExp of exp * exp * exp        (* if exp1 then exp2 else exp3 *)
   | AppExp of exp * exp             (* exp1 exp2 *)
   | BinExp of binop * exp * exp     (* exp1 % exp2
                                        where % is a builtin binary operator *)
   | MonExp of monop * exp           (* % exp1
                                        where % is a builtin monadic operator *)
   | FunExp of string * exp          (* fun x -> exp *)
   | LetExp of string * exp * exp    (* let x = exp1 in exp2 *)
   | RecExp of string * string * exp * exp    (* let rec f x = exp1 in exp2 *)
   | RaiseExp of exp                 (* raise exp *)
   | TryWithExp of exp * (int option * exp) * ((int option * exp) list)
                                (* try exp with n1 -> exp1 | ... | nm -> expm *)

type toplvl = Anon of exp (* f 4;; *)
            | TopLet of (string * exp) (* let f = ... ;; *)
            | TopRec of (string * string * exp) (* let rec f x = ... ;; *)
```

Thus each sequence of tokens should either be interpreted as an element of metatype exp or toplvl, or should yield a parse error. Note that the metatypes exp and toplvl contain abstract syntax trees, and not concrete syntax.

If we do not specify the precedence and associativity of our language constructs and operators, the parsing function is not well-defined. For instance, how should if true then 3 else 2 + 4 be parsed? Depending on how we "read" the above sequence of tokens, we get different results:

- If we read it as the sum of a conditional and a number, we get the same thing as if it were: `(if true then 3 else 2) + 4`
- If we read it as a conditional having a sum in its false branch, we get `if true then 3 else (2 + 4)`

The question is really which of the sum and the conditional binds its arguments tighter, that is, which one has a higher precedence (or which one has precedence over the other). In the first case, the conditional construct has a higher precedence; in the second, the sum operator has a higher precedence.

Another source of ambiguity arises from associativity of operators: how should `true && true && false` be parsed?

- If we read it as the conjunction between true and a conjunction, we get `true && (true && false)`
- If we read it as a conjunctions between a conjunction and false, we get `(true && true) && false`

In the first case, `&&` is right-associative; in the second, it is left-associative.

The desired precedence and associativity of the language constructs and operators (which impose a unique parsing function) are given below, ordered in lines from highest to lowest precedence:

- Highest precedence is application (`_ _`), which is left associative.
- Next is `raise`.
- Next is $**$, which is right associative.
- Next are $*$, $*.$, $/$, and $/.$, which are left associative.
- Next are $+$, $+.$, $-$, $-.$, and $\hat{}$, which are left associative.
- Next is `::`, which is right associative.
- Next are $=$, $<$, $>$, $<=$, and $>=$, which are left associative.
- Next is `&&`, which is left associative.
- Next is `||`, which is left associative.
- Next is `if _ then _ else _`.
- Next is `fun _ -> _`.
- Next is `let _ = _ in _`.
- Next is `let rec _ = _ in _`.
- Next is `try _ with _ -> _ | _ | ...`, where `|` is right associative.

Above, the underscores are just a graphical indication of the places where the various syntactic constructs expect their "arguments". For example, the conditional has three underscores, the first for the condition, the second for the `then` branch, and the third for the `else` branch.

7 Problems

At this point, your assignment for this MP should already be fairly clear. The following problems just break your assignment into pieces and are meant to guide you towards the solution. A word of warning is however in order here: The problem of writing a parser is *not* a modular one, because the parsing of each language construct depends on all the other constructs. Adding a new syntactic category may well force you to go back and rewrite all the categories already present. Therefore you should approach the set of problems as a whole, and always keep in mind the precedences and associativities given for the PicoML constructs. There are two main approaches to disambiguating the grammar: you may use `%left`, `%right`, and `%nonassoc` declarations, or modify the grammar itself, or any combination of the two. You are allowed to add to your grammar new nonterminals (together with new productions) in addition to the one that we require (`main`). In addition, you may find it desirable to rewrite or reorganize the various productions we have given you. The productions given are intended only to be enough to allow you to start testing your additions. Also, you may define the constructs in an order that is different from the order of the problems. For instance, we have gathered the requirements according to the intended semantics of the constructs (e.g., grouping arithmetic operators together and list operators together); you may rather want to group the constructs according to their precedence. However, we require that the non-terminal `main` that we introduced in the problem statement be present in your grammar and that it produce exactly the same set of strings as described by the grammar in Section 6, obeying the precedences and associativities also described in that section. In between each of these examples we have reset the environment.

1. (5 pts) In the file `mp7.mly` add the integer, unit, boolean, float, and string constants.

```
> "hi";;
val _ : string

proof:

  {} |= "hi" : string
```

2. (5 pts) Add parentheses.

```
> (3);;
val _ : int

final environment:

{}

proof:

  {} |= 3 : int
```

3. (5 pts) Add pairs. Note that unlike OCaml, PicoML requires opening and closing parentheses around pairs.

```
> (3,4);;
val it : int * int


final environment:

{}
```

```
proof:

  {} |= 3  ,  4 : int * int
  |--{} |= 3 : int
  |--{} |= 4 : int
```

4. (8 pts) Add unary operators. These should be treated for precedence and associativity as if they were application of single argument functions.

```
> hd [];;
val _ : 'a

final environment:

{}

proof:

  {} |= hd [] : 'a
  |--{} |= [] : 'a list
```

5. (12 pts) Add infix operators. You will need to heed the precedence and associativity rules given in the table above. Add arithmetic and string operators.

```
> 3 + 4 * 8;;
val _ : int

final environment:

{}

proof:

  {} |= 3  +  (4  *  8) : int
  |--{} |= 3 : int
  |--{} |= 4  *  8 : int
    |--{} |= 4 : int
    |--{} |= 8 : int
```

6. (12 pts) Add comparison operators. Since our language only has $=$ and $<$ we'll use the following syntactic sugar:

   - `a > b` $\mapsto$ `b < a`
   - `a <= b` $\mapsto$ `if a < b then true else a = b`
   - `a >= b` $\mapsto$ `if b < a then true else a = b`

```
> 3 > 5;;
val _ : bool

final environment:

{}

proof:

  {} |= 5  <  3 : bool
  |--{} |= 5 : int
  |--{} |= 3 : int
```

7. (10 pts) Add :: (list consing).

```
> 3 :: 2 :: 1 :: [];;
val _ : int list

final environment:

{}

proof:

  {} |= 3  ::  (2  ::  (1  ::  [])) : int list
  |--{} |= 3 : int
  |--{} |= 2  ::  (1  ::  []) : int list
    |--{} |= 2 : int
    |--{} |= 1  ::  [] : int list
      |--{} |= 1 : int
      |--{} |= [] : int list
```

8. (8 pts) Add `let_in_` and `let_rec_in`.

```
> let rec f x = 3 :: x :: (f x) in f 8;;
val _ : int list

final environment:

{}

proof:

  {} |= let rec f x = 3  ::  (x  ::  (f x)) in f 8 : int list
  |--{x : int} |= 3  ::  (x  ::  (f x)) : int list
  | |--{x : int} |= 3 : int
  | |--{x : int} |= x  ::  (f x) : int list
  |   |--{x : int} |= x : int
  |   |--{x : int} |= f x : int list
  |     |--{x : int} |= f : int -> int list
```

```
|     |--{x : int} |= x : int
|--{f : int -> int list} |= f 8 : int list
  |--{f : int -> int list} |= f : int -> int list
  |--{f : int -> int list} |= 8 : int
```

9. (20 pts) Add `fun_->_` and `if_then_else_`.

```
> if true then fun x -> 3 else fun x -> 4;;
val _ : 'a -> int

final environment:

{}

proof:

  {} |= if true then fun x -> 3 else fun x -> 4 : 'a -> int
  |--{} |= true : bool
  |--{} |= fun x -> 3 : 'a -> int
  | |--{x : 'a} |= 3 : int
  |--{} |= fun x -> 4 : 'a -> int
    |--{x : 'a} |= 4 : int
```

10. (10 pts) Add application.

```
> (fun x -> x + x + 3) 4;;
val _ : int

final environment:

{}

proof:

  {} |= (fun x -> (x  +  x)  +  3) 4 : int
  |--{} |= fun x -> (x  +  x)  +  3 : int -> int
  | |--{x : int} |= (x  +  x)  +  3 : int
  |   |--{x : int} |= x  +  x : int
  |   | |--{x : int} |= x : int
  |   | |--{x : int} |= x : int
  |   |--{x : int} |= 3 : int
  |--{} |= 4 : int
```

11. (11 pts) Add `&&` and `||`. Note that e_1`&&`e_2 should parse the same as `if` e_1 `then` e_2 `else false`, and e_2`||`e_2 should parse the same as `if` e_1 `then true else` e_2. In each case, the appropriate OCaml constructor to use is `IfExp`.

```
> true || false && true;;
val _ : bool
```

```
final environment:

{}

proof:

  {} |= if true then true else if false then true else false : bool
  |--{} |= true : bool
  |--{} |= true : bool
  |--{} |= if false then true else false : bool
    |--{} |= false : bool
    |--{} |= true : bool
    |--{} |= false : bool
```

12. (5 pts) Add `raise`.

```
> raise (fun x -> x) 4 + 3;;
val _ : int

final environment:

{}

proof:

  {} |= (raise (fun x -> x) 4)  +  3 : int
  |--{} |= raise (fun x -> x) 4 : int
  | |--{} |= (fun x -> x) 4 : int
  |   |--{} |= fun x -> x : int -> int
  |   | |--{x : int} |= x : int
  |   |--{} |= 4 : int
  |--{} |= 3 : int
```

8 Extra Credit

13. (5 pts) Add syntactic sugar for lists to your expressions. More precisely, add the following production to the grammar:

    - `exp` $\rightarrow$ [*list_contents*]

    where *list_contents* is a non-empty sequence of expressions separated by semicolons. It has to be the case that semicolon binds less tightly than any other language construct or operator.

```
> [3; 2; 1];;
val _ : int list

final environment:
```

```
{}

proof:

  {} |= 3  ::  (2  ::  (1  ::  [])) : int list
  |--{} |= 3 : int
  |--{} |= 2  ::  (1  ::  []) : int list
    |--{} |= 2 : int
    |--{} |= 1  ::  [] : int list
      |--{} |= 1 : int
      |--{} |= [] : int list
```

14. (10 pts) Add `try_with_`. Be sure to notice how the expression is parsed in the example: pipes are associated with the right-most preceding try-with.

    Valid patterns have the form `n -> e`, where `n` is represented by `Some` wrapped around an integer, or `_ -> e`, where `_` is represented by `None`.

```
try 0 with 1 -> 1 | 2 -> try 2 with _ -> 3 | 4 -> 4;;
val _ : int

final environment:

{}

proof:

  {} |= (try 0 with 1 -> 1| 2 -> (try 2 with _ -> 3| 4 -> 4)) : int
  |--{} |= 0 : int
  |--{} |= 1 : int
  |--{} |= (try 2 with _ -> 3| 4 -> 4) : int
    |--{} |= 2 : int
    |--{} |= 3 : int
    |--{} |= 4 : int
```