
MP 7 – A Parser for MicroML

CS 421 – Spring 2007

Revision 1.2

Assigned Wednesday March 28, 2007

Due Thursday April 5, 2007 23:59 PM

Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Corrections to add `^` and `div` as infixes.

1.2 Redundant listing of `^` removed from infix table, some minor typos fixed, `andalso` and `orelse` now defined when used in example

2 Overview

In this MP, we will deal with the process of converting MicroML 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 MicroML interpreter (well, it does not interpret yet), where you can type in MicroML expressions and see a proof tree of the expression's type:

```
Welcome to the parser
```

```
> val x = 5;
```

```
final environment:
```

```
{x->int}
```

```
proof:
```

```
{ } |= val x = 5 -> {x->int}  
|--{ } |= 5 : 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 *occamlyacc*.

3 Given Files

mp7yacc.mly: This is the only file you need to modify. It contains the skeleton of an *ocamlyacc* parser specification for MicroML. It also contains some pieces of code that we have started for you, with triple dots indicating places where you should add code.

mp7IntPar.ml: This file contains the main body of the MicroML executable. It essentially connects your lexer, parser, and type inference code and provides a friendly prompt to enter MicroML expressions.

mp7lex.cmo: This file contains the *ocamllex* specification for the lexer. It is a modest expansion to the lexer you wrote for MP6. It has a token *SEMI* to mark the end of a sequence of declarations to be processed. It also has tokens for each infix operator in the pervasive environment.

mp7common.cmo: Most of the content of this file should be familiar to you from MP5. This file includes the types of expressions and declarations. It also contains the type inferencing code from MP5. Appropriate code from this file will automatically be called from *mp7.cmo*.

4 Overview of *ocamlyacc*

Take a look at the given *mp7yacc.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 %%). You will only need to add to the last section.

4.1 Example

The following is the *expr* example from class:

```
%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
%%
expr:
  term                               { Term_as_Expr $1 }
  | term Plus_token expr             { Plus_Expr ($1, $3) }
  | term Minus_token expr            { Minus_Expr ($1, $3) }
term:
  factor                              { Factor_as_Term $1 }
  | factor Times_token term           { Mult_Term ($1, $3) }
  | factor Divide_token term          { Div_Term ($1, $3) }
factor:
  Id_token                            { Id_as_Factor $1 }
  | Left_parenthesis expr Right_parenthesis { Parenthesized_Expr_as_Factor $2 }
main:
  expr EOL                            { $1 }
```

Recall from lecture that the process of transforming program code (i.e. as 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 should be named `token` and the datatype `token` is created implicitly by the `%token` directives. These tokens are then fed into the *parser* created by entry points in your input, which builds the actual AST.

The first five lines define the types of tokens of the language. These directives are converted by *ocamlyacc* into an OCaml disjoint type declaration defining the type `token`. Notice that the `Id_token` token has data associated with it (this corresponds to writing `type token = ... | Id_token of string` in OCaml). The sixth line says that the goal nonterminal is the nonterminal called `main`. After the `%%` directive comes the important part: the productions. The format of the productions is fairly self-explanatory. The above specification describes the grammar:

$$\begin{aligned}
 S &::= E \text{ eol} \\
 E &::= T \quad E ::= T + E \quad E ::= T - E \\
 T &::= F \quad T ::= F * T \quad T ::= F / T \\
 F &::= id \quad F ::= (E)
 \end{aligned}$$

The important thing about *ocamlyacc* is that **each production returns a value** that is to be put on the stack. We call this the *semantic value* of the production. It is described in curly braces by the *semantic action*. The semantic action is actual OCaml code that will be evaluated when this parsing algorithm reduces by this production. The result of this code is the semantic value, and it is placed on the stack to represent the nonterminal.

What do `$1`, `$2`, etc., mean? These refer to the positional values on the stack, and are replaced in the OCaml code by the semantic values of the subexpressions on the right-hand side of the production. Thus, the symbol `$1` refers to the semantic value of the first subexpression on the right-hand side, and so on.

As an example, consider the following production:

```

expr :
  ...
  | term Plus_token expr           { Plus_Expr ($1, $3) }

```

When the parser reduces by this rule, `$1` holds the semantic value of the `term` subexpression, and `$3` holds the value of the `expr` subexpression. The semantic rule generates the AST representing the addition of the two, and the result becomes the semantic value for this production and is put on the stack to replace the top three items.

Also note that when tokens have associated data (like `Id_token`, which has a string), that associated data is treated as the semantic value of the token:

```

factor :
  Id_token           { Id_as_Factor $1 }

```

Thus, the above `$1` corresponds to the string component of the token, and not the token itself.

4.2 More Information

Here is a website you should check out if you would like more information or an alternate explanation of *ocamlyacc* usage:

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

5 Compiling

A `Makefile` is provided for this MP. After you make changes to `mp7yacc.mly`, all you have to do is type `gmake` (or possibly `make` if you are using a non-linux machine) and the two needed executables will be rebuilt.

5.1 Running MicroML

The given `Makefile` builds executables called `mp7IntPar` and `mp7IntParSol`. 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 standard solution. If you run `./mp7IntPar` or `./mp7IntParSol`, you will get an interactive screen, much like the OCaml interactive screen. You can type in MicroML expressions followed by a double semicolon, and they will be parsed and their types inferred and displayed:

6 Important Notes

- The BNFs below for MicroML's grammar are ambiguous, and it is just a description of the *concrete* syntax of MicroML. You are also provided with a table listing the associativity/precedence attributes of the various language constructs. You are supposed to use the information given in this table in order to create a grammar that generates the same language as the given one, but that is unambiguous and enforces the constructs to be specified as in the table. Your actual *ocaml yacc* specification will consist of the latter grammar.
- **The BNFs do not show the stratification needed to eliminate ambiguity. That is your job!** This will likely involve reorganizing things.
- For this MP, you must implement precedence using stratification. *ocaml yacc* has some shortcut directives (`%left`, `%right`) for defining operator precedence, but these are off-limits for this MP! Do not use them!
- Even though the work in this MP is split into several problems, you should really have the overall view on how the disambiguated grammar will look like, because precedence makes the choices for the productions corresponding to the language constructs conceptually interdependent. You might want to read through all the expression types first and try to organize your stratification layers before starting. 90 percent of your intellectual effort in this MP will consist of disambiguating the grammar properly.

Stratification means breaking something up into layers. In the example 4.1, We could have expressed the grammar more simply and succinctly by

$$\begin{aligned} S &::= E \text{ eol} \\ E &::= id \mid E + E \mid E - E \mid E * E \mid E / E \mid (E) \end{aligned}$$

This grammar, while compact, and comprehensible to humans, is highly ambiguous for the purposes of parsing. To render it unambiguous, we introduced intermediate non-terminals (layers, or strata) to express associativity and precedence of operators. You will need to perform similar transformations on the description given here to remove ambiguities and avoid shift-reduce or reduce-reduce conflicts.

7 Problem Setting

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

```
<main> ::= <dec> ;

<dec> ::= val ID = <exp>
      | val rec ID = <exp>
      | fun ID ID = <exp>
      | local <dec> in <dec> end
      | <dec> <dec>
```

```

<exp> ::= ID
        | UNIT | BOOL | INT | FLOAT | STRING
        | ( <exp> )
        | ( <exp> , <exp> )
        | let <dec> in <exp> end
        | <exp> <infix> <exp>
        | <exp> andalso <exp>
        | <exp> orelse <exp>
        | [ ]
        | [ <list_contents> ] /* sugar for non-empty lists, extra credit */
        | if <exp> then <exp> else <exp>
        | <exp> <exp>
        | fn ID => <exp>
        | ID
        | op ID
        | op <infix>

```

<list_contents> ::= <nonempty sequence of expressions separated by commas>

ID refers to an identifier token (only one token, takes a string as argument) <infix> refers to some infix identifier token (one for each infix operator). These new tokens are: STRCAT (“^”), EXPO (“**”), DIV (“div”), MOD(“mod”), TIMES (“*”), RTIMES (“*.”), FRAC (“/”), RFRAC (“/.”), PLUS (“+”), RPLUS (“+.”), MINUS (“-”), RMINUS (“-.”), LESS (“<”), GR (“>”), LEQ (“<="), GEQ (“>="). (<infix> is not a syntactic category that you must parse. It is only used here to express <exp>.)

The nonterminals in this grammar are main, exp, list_contents, and dec, with main being the start symbol. A main consists of a dec ended by a semicolon.

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 yacc.mly. Our OCaml representation of terminals is not always graphically identical to the one shown in the above grammar; for example, :: is represented by DBLCOLON and + by PLUS. Our OCaml representation of the variable tokens is achieved by the constructor Var that takes a string and yields a token, as constructed by the lexer from MP6.

Recall that identifying the tokens of the language is the job of lexers, and the parser (that you have to write in this MP) takes as input a *sequence of tokens*, such as TRUE ANDALSO FALSE and tries to make sense out of it by transforming it into an abstract syntax tree, in this case IfThenElse(Bool true, Bool false, Bool false). The abstract syntax trees that you have to parse your sequences of tokens into are given by the following OCaml types (metatypes, to avoid confusion with MicroML types), present in the file mp7common.ml:

```

type dec = Valbind of (string * exp) | ValbindRec of (string * exp)
         | Local of (dec * dec) | Seq of (dec * dec)
and exp =
  (*special constants*)
  Unit | Bool of bool | Char of char | Int of int | Real of float
        | String of string
  (*pair*)
  | Pair of exp * exp
  (*identifier*)
  | Id of string
  (*let in, application, functions*)
  | LetIn of dec * exp | App of exp * exp | Fn of string * exp
  | IfThenElse of exp * exp * exp

```

Thus each sequence of tokens should either be interpreted as an element of metatype `exp` or `dec`, or should yield a parse error. Note that the metatypes `exp` and `dec` contain abstract, and not concrete syntax. Thus, for instance, an expression is here the AST that a concrete sequence of tokens might be parsed into, but not that concrete sequence of tokens itself. For instance, `TRUE ANDALSO FALSE` is *not* an expression, but a sequence of tokens belonging to the set of terminal words generated in the grammar by the nonterminal `exp`,¹ which will be parsed into an expression.

Recall from MP5 that our abstract syntax encodes any operator of MicroML (of one or more arguments) using application. This is why `3 + 4` parses to `App(Id "+", Pair(Int 3, Int 4))`, and `[2, 3]` parses to `App(Id "::", Pair(Int 2, App(Id "::", Pair(Int 3, Id "nil"))))`.

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
`App(Id "+", Pair(IfThenElse(Bool true, Int 3, Int 2), Int 4))`
- If we read it as a conditional having a sum on its false branch, we get
`IfThenElse(Bool true, Int 3, App(Id "+", Pair(Int 2, Int 4)))`.

The question is really which of the sum and 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: consider the operators `andalso` and `orelse`, which are short-circuit versions of `and` and `or`.² How should `true andalso true andalso false` be parsed?

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

In the first case, `andalso` 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, where a left-associative operator is preceded by "left", a right-associative operator by "right", and precedence decreases downwards on the lines (thus two items listed on the same line have the same precedence).

```

op _
left _ _      (that is to say, application binds tighter than anything but
               "op" and "let_in_end" and associates to the left)
right **
left * left *. left / left /. left mod left div
left + left +. left - left -. left ^
right ::
left = left < left > left <= left >=
left _andalso_
left _orelse_
if_then_else_
fn_=>_
right _ _      (for sequences of declarations)

```

¹We use the same name, `exp`, for two different (but related) things: for a nonterminal in the grammar, and for the meta-type of (abstract) expressions.

²`x andalso y` is equivalent to `if x then y else false`. Furthermore, `x orelse y` is equivalent to `if x then true else y`.

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.

8 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 MicroML constructs. You are allowed, and even encouraged, to add to your grammar new nonterminals (together with new productions) in addition to the ones that we require (`dec` and `exp`). 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 it is allowed that you define the constructs in an order which is different from the one we have given here. 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; you are absolutely free to do that. However, we require that the nonterminals `main`, `dec` and `exp` that we introduce in the problem statements be present in your grammar and that they produce exactly the same set of strings as described by the grammar in Section 7, obeying the precedences and associativities also described in that section. For example, your grammar should contain a nonterminal called `dec` that produces all the declarations in MicroML (even though, for associativity and precedence reasons, you might want to introduce new intermediate nonterminals that split the declarations into subcategories).

1. (5 pts) In the file `mp7yacc.mly`, we have already included a few productions of `exp` and `dec`, including integer constant expressions in the grammar. Add the unit, boolean, float, and string constants.

```
> val x = @"hi";

final environment:

{x->string}

proof:
  {} |= val x = "hi" -> {x->string}
  ...

> val x = true;

final environment:

{x->bool}

proof:
  {} |= val x = true -> {x->bool}
  ...
```

2. (5 pts) Add parentheses.

```
> val x = (3 + 4);
```

```
final environment:
```

```
{x->int}
```

```
proof:
```

```
{ } |= val x = (op +)(3,4) -> {x->int}
```

```
...
```

3. (8 pts) Add `let_in_end`.

```
> val x = let val x = 3 in x + x end;
```

```
final environment:
```

```
{x->int}
```

```
proof:
```

```
{ } |= val x = let val x = 3 in (op +)(x,x) end -> {x->int}
```

```
...
```

4. (10 pts) Add application. (You can't execute this code until you do Problem 9.)

```
> val f = fn x => x + x val res = f 3;
```

```
final environment:
```

```
{res->int, f->int -> int}
```

```
proof:
```

```
{ } |= val f = fn x => (op +)(x,x) val res = f3 -> {res->int, f->int -> int}
```

```
...
```

5. (12 pts) We have already included the "op" command, which allows one to treat an infix operator as a standard identifier. Example: `(op +) (3,4) -> 3 + 4`. However, you will need to treat each infix operator also as an infix, and you will need to heed the precedence and associativity rules given in the table above.

Start by adding arithmetical and string operators.

```
> val x = 3 + 4 * 2;
```

```
final environment:
```

```
{x->int}
```

```
proof:
```

```
{ } |= val x = (op +)(3, (op *)(4,2)) -> {x->int}
```

```
...
```

```
> val x = 2.0 *. 3.1 *. 2.5;
```

```
final environment:
```

```
{x->real}
```

```
proof:
```

```
{ } |= val x = (op *.)((op *.)(2.,3.1),2.5) -> {x->real}  
...
```

```
> val z = 2.0 ** 3.0 ** 2.0;
```

```
final environment:
```

```
{z->real}
```

```
proof:
```

```
{ } |= val z = (op **)(2.,(op **)(3.,2.)) -> {z->real}  
...
```

6. (12 pts) Add comparison operators. Note that these only work on integers, except for = which is polymorphic.

```
> val x = 3 < 4;
```

```
final environment:
```

```
{x->int}
```

```
proof:
```

```
{ } |= val x = (op <)(3,4) -> {x->bool}  
...
```

```
> val x = @"hi" = @"hi";
```

```
final environment:
```

```
{x->bool}
```

```
proof:
```

```
{ } |= val x = (op =)("hi","hi") -> {x->bool}  
...
```

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

```
> val x = 3 :: 2 :: 1 :: nil;
```

```
final environment:
```

```
{x->list(int)}
```

```
proof:
```

```
{} |= val x = (op ::)(op ::) (3,(op ::)(op ::)) (2,(op ::)(op ::))  
(1,nil)) -> {x->list(int)}  
...
```

8. (11 pts) Add `andalso` and `orelse`. Note that `x andalso y` is equivalent to `if x then y else false`. Furthermore, `x orelse y` is equivalent to `if x then true else y`.

Note: These are not treated as infix operators. Rather, they are special constructs and get much lower precedence as a result.

```
> val x = true orelse false andalso true;
```

```
final environment:
```

```
{x->bool}
```

```
proof:
```

```
{} |= val x = if true then true else if false then true else false -> {x->bool}  
...
```

9. (20 pts) Add `fn=>.` and `if.then.else.`

```
> val f = fn x => if x then 3 else 4;
```

```
final environment:
```

```
{f->bool -> int}
```

```
proof:
```

```
{} |= val f = fn x => if x then 3 else 4 -> {f->bool -> int}  
...
```

10. (10 pts) Add function declarations and local declarations. A function declaration is a declaration like this:

```
fun f x = x
```

This is equivalent to `val f = fn x => x`.

```
> fun f x = x;
```

```
final environment:
```

```
{f->'d -> 'd}
```

```
proof:
```

```
{} |= val rec f = fn x => x -> {f->'d -> 'd}  
...
```

```

> val x = @"hi" local val x = 3 in val y = x end;

final environment:

{y->int,x->string}

proof:
  {} |= val x = "hi" local val x = 3 in val y = x end -> {y->int,x->string}
  ...

```

Note: 'd is a type variable. f is a polymorphic function.

9 Extra Credit

11. (5 pts) Add syntactic sugar for lists to your expressions. More precisely, add the following expressions to the grammar:

- $\text{exp} \rightarrow [\text{list_contents}]$

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

```

> val x = [1, 2, 3];

does not parse
> val x = 1 :: 2 :: 3 :: [];

final environment:

{x->list(int)}

proof:
  {} |= val x = (op ::)(op ::) (1,(op ::)(op ::) (2,(op ::)(op ::) (3,nil))) ->
  {x->list(int)}
  ...

```

12. (5 pts) Add syntactic sugar for functions with multiple arguments to your expressions. More precisely, add the following expressions to the grammar:

- $\text{dec} \rightarrow \text{fun } f1 \text{ fun_vars } = \text{exp}$

where *fun_vars* is a non-empty sequence of variables separated by white-space.

```

> fun add x y = x + y;

final environment:

{add->int -> int -> int}

```

```
proof:
  {} |= val rec add = fn x => fn y => (op +) (x,y) -> {add->int -> int -> int}
  ...
```

10 Additional tests

1. Can you pass the following test?

```
> val x = head(true :: (if true then false else true andalso true) :: nil);
```

```
final environment:
```

```
{x->bool}
```

```
proof:
```

```
{ } |= val x = head (op ::) (true,(op ::) (if true then false else if
  true then true else false,nil)) -> {x->bool}
```

```
...
```

2. How about this one?

```
> val x = 3 - 4 - 2 * 9 > 10 andalso true andalso head(true::nil);
```

```
final environment:
```

```
{x->bool}
```

```
proof:
```

```
{ } |= val x = if if (op >) ((op -) ((op -) (3,4),(op *) (2,9)),10)
  then true else false then head (op ::) (true,nil) else false -> {x->bool}
```

```
...
```

3. This one?

```
> val x = 3 - 4 - 2 * 9 > 10 andalso true andalso head(true::nil);
```

```
final environment:
```

```
{x->bool}
```

```
proof:
```

```
{ } |= val x = if if (op >) ((op -) ((op -) (3,4),(op *) (2,9)),10)
  then true else false then head (op ::) (true,nil) else false -> {x->bool}
```

```
...
```