MP 8 – A Parser for MicroML

CS 421 – su 2011

Revision 1.0

Assigned July 24, 2011
Due July 31, 2011 23:59 PM
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

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 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

mp8-skeleton.mly: You should copy the file **mp8-skeleton.mly** to **mp8.mly**. The skeleton contains some pieces of code that we have started for you, with triple dots indicating places where you should add code.

micromlIntPar.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.

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

mp8common.ml: This file includes the types of expressions and declarations. It also contains the type inferencing code. Appropriate code from this file will automatically be called by the interactive loop defined in **micromlIntPar.ml:**. You will want to use the types defined in this file, and probably some of the functions, when you are creating your attributes in **mp8.mly**.

4 Overview of `ocamlyacc`

Take a look at the given **mp8-skeleton.mly** file. The grammar specification has a similar layout to the lexer specification of MP7. 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 `exp` example from class (lecture 19 LR Parsing, slides 52 – 59):

```
%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
%%
exp:
   term                                   { Term_as_Expr $1 }
 | term Plus_token exp                    { Plus_Expr ($1, $3) }
 | term Minus_token exp                   { 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 exp Right_parenthesis { Parenthesized_Expr_as_Factor $2 }
main:
 exp 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 in the example above define the sorts 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 start symbol for the grammar 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 following extended BNF grammar:

$$
\begin{array}{lll}
S ::= E\ \textbf{eol} & & \\
E ::= T & \mid T + E & \mid T - E \\
T ::= F & \mid F * T & \mid F / T \\
F ::= id & \mid (E) &
\end{array}
$$

An important fact 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:

```
exp:
  ...
  | term Plus_token exp                      { 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 `exp` 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 `mp8.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 `micromlIntPar` and `micromlIntParSol`. 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 `./micromlIntPar` or `./micromlIntParSol`, you will get an interactive screen, much like the OCaml interactive screen. You can type in MicroML expressions followed by a (single) semicolon, and they will be parsed and their types inferred and displayed:

```
Welcome to the Solution parser

> 3;
```

```
val it : int


final environment:

{ it : int }

proof:
  {  } |= val it = 3 :: { it : int }
  |--{  } |= 3 : int

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


final environment:

{ x : int, it : int }

proof:
  { it : int } |= val x = 3 + 4 :: { x : int }
  |--{ it : int } |= 3 + 4 : int
    |--{ it : int } |= (op +) 3 : int -> int
    | |--{ it : int } |= (op +) : int -> int -> int
    | |--{ it : int } |= 3 : int
    |--{ it : int } |= 4 : int

> val f = fn y => y * x;
val f : int -> int


final environment:

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

proof:
  { x : int, it : int } |= val f = fn y => y * x :: { f : int -> int }
  |--{ x : int, it : int } |= fn y => y * x : int -> int
    |--{ y : int, x : int, it : int } |= y * x : int
      |--{ y : int, x : int, it : int } |= (op *) y : int -> int
      | |--{ y : int, x : int, it : int } |= (op *) : int -> int -> int
      | |--{ y : int, x : int, it : int } |= y : int
      |--{ y : int, x : int, it : int } |= x : int

> f 5;
val it : int


final environment:

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

proof:
  { f : int -> int, x : int, it : int } |= val it = f 5 :: { it : int }
  |--{ f : int -> int, x : int, it : int } |= f 5 : int
    |--{ f : int -> int, x : int, it : int } |= f : int -> int
    |--{ f : int -> int, x : int, it : int } |= 5 : int

>
```

Note: your output might have different type variables than those shown is subsequent examples.

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 Important Notes

- The BNF below for MicroML's grammar is 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 *ocamlyacc* specification will consist of the latter grammar.
- **The BNF does not show the stratification needed to eliminate ambiguity. That is your job!** This will likely involve reorganizing things.
- *ocamlyacc* has some shortcut directives (`%left`, `%right`) for defining operator precedence. You may use them in this MP. But be warned that, while using these features *may* decrease time to write the grammar, it takes some time to read documentation and understand these features. Our solution does not use these, and the staff will not be obliged to help you learn how to 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 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.
- **You will lose points for shift/reduce and reduce/reduce conflicts in your grammar**. Reduce/reduce conflicts will be penalized more heavily. `ocamlyacc -v mp8.mly` will inform you of any conflicts in your grammar, and produce a file `mp8.output` giving the action and goto table information and details of how conflicts arise and which productions are involved.

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

$$S ::= E \textbf{ eol}$$
$$E ::= id \mid E + E \mid E - E \mid E * E \mid E/E \mid (E)$$

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 and reduce-reduce conflicts.

7 Problem Setting

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

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

<dec> ::= <dec> <dec>
        | local <dec> in <dec> end
        | val IDENT = <exp>
        | val _ = <exp>
        | val rec IDENT = <exp> { and IDENT = <exp> }*
        | fun IDENT IDENT { IDENT }* = <exp> and { IDENT IDENT { IDENT }* = <exp> }*

<exp> ::= IDENT
        | BOOL | INT | REAL | STRING | UNIT | NIL
        |  ~ | fst | snd | hd | tl
        | ( <exp> )
        | ( <exp> { , <exp> }* )
        | ( <exp> { ; <exp> }* )
        | [ ]
        | [ <exp> { , <exp> }* ]
        | let <dec> in <exp> { ; <exp> }* end
        | op <infid>
        | <exp> <infid> <exp>
        | <exp> andalso <exp>
        | <exp> orelse <exp>
        | <exp> <exp>
        | if <exp> then <exp> else <exp>
        | fn IDENT => <exp>
        | raise <exp>
        | handle <exp> with <pat> => <exp> { PIPE <pat> => <exp> }*   /* extra credit */

<pat> ::= INT | _
```

Note: We use { A }* as a shorthand for a sequence of 0 or more As, and we use `PIPE` token to distinguish | in MicroML with the one in BNF syntax.

`IDENT` refers to an identifier token (only one token, taking a string as argument). `<infid>` refers to some infix identifier token (one for each infix operator). Tuples should have at least 2 elements in it. (`<infid>` is not an explicit syntactic category that you must parse. It is only used here to express `<exp>`.)

The nonterminals in this grammar are `main`, `dec`, `exp`, and `pat`, 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 `mp8.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 most 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.

Some of productions in the above grammar do not have a corresponding direct representation in abstract syntax. These language constructs are syntactic sugar and are summarized in the following table.

Expression	*Desugared form*	*Notes*
x `andalso` y x `orelse` y	`if` x `then` y `else false` `if` x `then true else` y	
x < y x <= y x >= y x <> y	(y > x) (y > x) `orelse` (x = y) (x > y) `orelse` (x = y) `if` (x = y) `then false else true`	with further expansion of orelse
x `<infix>` y	`(op <infix>)` x y	`+ - * / +. -. *. /. ^ :: = <`
`fun` $f_1\ a_1 \ldots a_n = exp_1$ ... `and` $f_m\ a_1 \ldots a_k = exp_m$	`val rec` f_1 `=` `fn` a_1 `=>` ... `=>` `fn` a_n `=>` exp_1 ... `and` f_m `=` `fn` a_1 `=>` ... `=>` `fn` a_k `=>` exp_m	
`(`exp_1`,`exp_2`,`...`,`exp_n`)`	`(`exp_1`, (`exp_2`, (`...`,`exp_n`))`...`)`	tuples are nested pairs, `,` is right-assoc.

`[]`
`[`exp_1`,`exp_2`,`...`,`exp_n`]`
	`nil`
	exp_1 `::` (exp_2 `::` (... (exp_n `:: nil))`...`)`
		
		`::` is right associative

exp_1`;`exp_2	`let val _ =` exp_1 `in` exp_2 `end`	`;` is right associative
exp`;`	`val it =` exp`;`	At the top level only (`<main>`)

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
`AppExp(AppExp(ConstExp(PrimOp(BinOp(IntPlusOp))), ConstExp(IntConst 3)), ConstExp(IntConst 5))`. The abstract syntax trees into which you have to parse your sequences of tokens are given by the following OCaml types (metatypes, to avoid confusion with MicroML types), present in the file `mp8common.ml`:

```
type bin_op =
     IntPlusOp
   | IntMinusOp
   | IntTimesOp
   | IntDivOp
   | RealPlusOp
   | RealMinusOp
   | RealTimesOp
   | RealDivOp
   | ConcatOp
   | ConsOp
   | CommaOp
   | EqOp
   | GreaterOp

type prim_op = BinOp of bin_op | IntNeg | HdOp | TlOp | FstOp | SndOp

type const =
     BoolConst of bool
   | IntConst of int
   | RealConst of float
   | StringConst of string
   | NilConst
   | UnitConst
   | PrimOp of prim_op

(* declarations *)
type dec =
     Val of string option * exp
```

```
    | Rec of (string * exp) nelist
    | Seq of dec * dec
    | Local of dec * dec

(* expressions for MicroML *)
and exp =
    | VarExp of string
    | ConstExp of const
    | IfExp of exp * exp * exp
    | AppExp of exp * exp
    | FnExp of string pattern
    | LetExp of dec * exp
    | RaiseExp of exp
    | Handle of exp * (int option pattern nelist)

and 'a pattern = 'a * exp
```

The above is a slight variation (allowing underscore in val bindings, fst, snd, hd and tl operators) of the OCaml types with which you worked in MP5.

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. Recall from MP5 that our abstract syntax encodes any operator of MicroML (except for `andalso`, `orelse` and <=, <, >=, <>, which are syntactic sugar for corresponding combination of >, = and if/then/else) using application. This is why `3 + 4` parses to

`AppExp(AppExp(ConstExp(PrimOp(BinOp(IntPlusOp))), ConstExp(IntConst 3)), ConstExp(IntConst 4))`.

Similarly, `[2,3]` parses to

`AppExp(AppExp(ConstExp(PrimOp(BinOp(ConsOp))), ConstExp(IntConst 2)), AppExp(AppExp(ConstExp(PrimOp(BinOp(ConsOp))), ConstExp(IntConst 3)), ConstExp(NilConst)))`

Notice that we here discuss how to parse *sequences of items*, and not the original text that the MicroML programmer writes - thus, one programming in MicroML one writes `3+4`, which will be lexed to `(INT 3) PLUS (INT 4)`, and then parsed to `AppExp(AppExp(ConstExp(PrimOp(BinOp(IntPlusOp))), ConstExp(IntConst 3)), ConstExp(IntConst 4))`.

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 andalso true andalso false` be parsed?

- If we read it as the conjunction between true and a conjunction, we get `true andalso (true andalso false)`
- If we read it as a conjunctions between a conjunction and false, we get `(true andalso true) andalso 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 _        (op_ binds tighter than anything else)
left _ _    (application is left associative)
~_ hd_ tl_ fst_ snd_
left *  left *.  left /  left /.
left +  left +.  left -  left -.  left ^
right ::    (cons is right associative)
left =  left <  left >  left <=  left >= left <>
left _andalso_
left _orelse_
if_then_else_
fn_=>_
raise_
handle_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.

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 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, it is allowed that you define the constructs in an order that is different from the one we have given here. For instance, we have gathered the requirements according to overall syntactic and semantic similarities (e.g., grouping arithmetic 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 non-terminal main that we introduced in the problem statement be present in your grammar and that it produces exactly the same set of strings as described by the grammar in Section 7, obeying the precedences and associativities also described in that section.

1. (3 pts) In the file mp8.mly add the integer, unit, nil, boolean, real, and string constants.

```
> val x = "hi";
val x : string


final environment:

{ x : string }

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

2. (3 pts) Add ˜ (negation), fst, snd, hd, tl. These are represented as primitive operations in abstract syntax.

```
> fst;
val it : ('c * 'd) -> 'c


final environment:

{ it : ('c * 'd) -> 'c }

proof:
  {  } |= val it = fst :: { it : ('c * 'd) -> 'c }
  |--{  } |= fst : ('c * 'd) -> 'c
```

3. (3 pts) Add parentheses.

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


final environment:

{ it : int }

proof:
  {  } |= val it = 3 :: { it : int }
  |--{  } |= 3 : int
```

4. (5 pts) Add tuples. Note that unlike OCaml, MicroML requires opening and closing parentheses around tuples. Notice that (a1,a2,...,an) is a syntactic sugar for (a1,(a2,(...,an))), with `,` associating to the right.

```
> (3, 9);
val it : int * int


final environment:

{ it : int * int }

proof:
  ...
```

5. (8 pts) Add `let_in_end` and expression sequencing.

```
> let val x = 3 in x end;
val it : int


final environment:
```

```
{ it : int }

proof:
  {  } |= val it = let val x = 3 in x end :: { it : int }
  ...

> ("hi"; 3);
val it : int


final environment:

{ it : int }

proof:
  {  } |= val it = let val _ = "hi" in 3 end :: { it : int }
  ...
```

6. (5 pts) Add syntactic sugar for lists to your expressions. More precisely, add the following expressions to the grammar:

   - `<exp> ::= []`
   - `<exp> ::= [ <exp>` { `, <exp>` }⋆ `]`

   It has to be the case that comma binds less tightly than any other language construct or operator.

```
> val x = [1, 2, 3];
val x : int list


final environment:

{ x : int list }

proof:
  {  } |= val x = 1 :: (2 :: (3 :: nil)) :: { x : int list }
  ...
```

7. (5 pts) Add `andalso` and `orelse`. Note that e_1 `andalso` e_2 should parse the same as `if` e_1 `then` e_2 `else false`, and e_2 `orelse` 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 orelse false;
val it : bool


final environment:

{ it : bool }

proof:
  {  } |= val it = if true then true else false :: { it : bool }
  ...
```

8. (12 pts) Add comparison operators.

```
> "a" < "b";
val it : bool


final environment:

{ it : bool }

proof:
  {  } |= val it = "b" > "a" :: { it : bool }
  ...
```

9. (10 pts) Add :: (list consing).

```
> 3 :: 2 :: 1 :: nil;
val it : int list


final environment:

{ it : int list }

proof:
  {  } |= val it = 3 :: (2 :: (1 :: nil)) :: { it : int list }
  ...
```

10. (12 pts) Add infix operators for arithmetic and also for string concatenation. You will need to heed the precedence and associativity rules given in the table above.

```
> 3 + 4 * 2;
val it : int


final environment:

{ it : int }

proof:
  {  } |= val it = 3 + (4 * 2) :: { it : int }
  ...
```

11. (10 pts)

Add application.

```
> hd nil;
val it : 'e
```

```
final environment:

{ it : 'e }

proof:
  {  } |= val it = hd nil :: { it : 'e }
  |--{  } |= hd nil : 'e
    |--{  } |= hd : 'e list -> 'e
    |--{  } |= nil : 'e list
```

12. (5 pts) Add op_ operator. This operator takes an infix binary operator and converts it to a function of two arguments.

```
> op +;
val it : int -> int -> int


final environment:

{ it : int -> int -> int }

proof:
  {  } |= val it = (op +) :: { it : int -> int -> int }
  |--{  } |= (op +) : int -> int -> int
```

13. (20 pts) Add `fn_=>_`, `if_then_else_` and `raise_`

```
> fn x => if x then 3 else raise 4;
val it : bool -> int


final environment:

{ it : bool -> int }

proof:
  {  } |= val it = fn x => if x then 3 else raise 4 :: { it : bool -> int }
  ...
```

14. (2 pts) Add `val _ = <exp>`.

```
> val _ = "hi";


final environment:

{  }

proof:
  {  } |= val _ = "hi" :: {  }
  |--{  } |= "hi" : string
```

15. (3 pts) Add local declarations.

```
> local val x = 2 in val y = x + 3 end;
val y : int


final environment:

{ y : int }

proof:
  {  } |= local val x = 2
in val y = x + 3 end :: { y : int }
  ...
```

16. (10 pts) Add `val rec`. The `and` operator associates to the right.

```
> val rec f = fn x => f x;
val f : 'c -> 'd


final environment:

{ f : 'c -> 'd }

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

17. (15 pts) Add `fun`. Note that `fun` is syntactic sugar for `val rec` and `and` associates to the right.

```
> fun f x = g x x and g a b = f b;
val f : 'f -> 'g
val g : 'f -> 'f -> 'g


final environment:

{ g : 'f -> 'f -> 'g, f : 'f -> 'g }

proof:
  {  } |= val rec f = fn x => g x x
and g = fn a => fn b => f b :: { g : 'f -> 'f -> 'g, f : 'f -> 'g }
  ...
```

9 Extra Credit

18. (10 pts) Add `handle_with_`. Be sure to notice how the expression is parsed in the second example: pipes are associated with the right-most preceding handle-with (the ambiguity this fixes is analogous to the dangling-else problem.)

Valid patterns have the form `n => e`, where `n` is to be represented by `Some` of an integer, or `_ => e`, where `_` is represented is by `None`.

```
> handle "hi" with 1 => "one" | 2 => "two";
val it : string


final environment:

{ it : string }

proof:
  {  } |= val it = handle "hi" with 1 -> "one" | 2 -> "two" :: { it : string }
  ...
```

10 Additional tests

1. Can you pass this test? Make sure your parser parses the expression as in the example.

   ```
   > 3 - 4 - 2 * 9 > 10 andalso true;
   val it : bool


   final environment:

   { it : bool }

   proof:
     {  } |= val it = if 3 - 4 - (2 * 9) > 10 then true else false :: { it : bool }

     ...
   ```

2. This one?

   ```
   > if true then 1 else 0 + 2;
   val it : int


   final environment:

   { it : int, it : bool }

   proof:
     { it : bool } |= val it = if true then 1 else 0 + 2 :: { it : int }
     |--{ it : bool } |= if true then 1 else 0 + 2 : int
       |--{ it : bool } |= true : bool
       |--{ it : bool } |= 1 : int
       |--{ it : bool } |= 0 + 2 : int
     ...
   ```

3. This one?

```
> (fn x => ()) 3;
val it : unit


final environment:

{ it : unit, it : int, it : bool }

proof:
  { it : int, it : bool } |= val it = (fn x => ()) 3 :: { it : unit }

  ...
```