

## 1 Introduction

In this lab assignment, we are going to implement an interpreter for the  $\lambda$ -calculus language we defined in HW6. This will require *lexing* and *parsing* to turn a textual description of a program into an expression. We will be using `ocamllex` for lexing, and `ocaml yacc` for parsing. This lab description contains a rough outline of what you will need to know, but you will also need to *read the documentation*.

Your interpreter has three total parts.

1. **Lexing** separates the text in an input file into *tokens* (terminal symbols) that represent things like numbers, identifiers, special characters, keywords, etc. It also removes comments and whitespace. We will implement the lexer using the `ocamllex` program.
2. **Parsing** produces the program expression from the sequence of tokens returned by the lexer. The parser is specified as a *context-free grammar* that includes a *semantic action* for each production in the grammar. We will implement the parser using the `ocaml yacc` program.
3. **Evaluation** uses the evaluator you generated in HW6.

## 2 Source language

First, we have to define the language we are compiling.

### 2.1 Lexical conventions

#### 2.1.1 Comments

Comments begin with the characters `(*` and are terminated with `*)`. Any text is allowed in a comment, but comments can't be nested.

#### 2.1.2 Identifiers

Identifiers (for variable, type, and function names) are a sequence of letters and digits, starting with a letter. The underscore `_` counts as a letter.

#### 2.1.3 Keywords

The following words are used as keywords.

```
fun  let  in
if   then else
```

## 2.1.4 Constants

- *integers* have the following form:

**decimal** `['0'-'9']+`

**octal** `"0o" ['0'-'7']*`

**hex** `"0x" ['0'-'9' 'a'-'f' 'A'-'F']*`

## 2.2 Grammar

## 2.3 Program

A toplevel program is a sequence of expressions and let-definitions. Each definition is followed by a double-semicolon ; ;.

*prog* ::= *exp* ; ; | *let* *v* = *exp* ; ; | *let* *f* *v* = *e* ; ;

## 2.4 Expressions

The hard part of expressions is defining application correctly. To make it easier, we *factor* the grammar into “atomic” expressions and normal “compound” expressions. The atomic expressions are the numbers, variables, and parenthesized expressions.

*atomic* ::= *number* | *var* | ( *exp* )

Given the atomic expressions, we define an *application* as a concatenation of atomic expressions.

*apply* ::= *atomic* | *apply atomic*

The expressions correspond directly to the expression type.

*exp* ::= *apply*

| *exp* + *exp* | *exp* - *exp* | *exp* \* *exp* | *exp* / *exp*

| *exp* < *exp* | *exp* <= *exp* | *exp* > *exp* | *exp* >= *exp* | *exp* = *exp* | *exp* <> *exp*

| **if** *exp* **then** *exp* **else** *exp*

| **fun** *var* -> *exp*

| **let** *var* = *exp* **in** *exp*

| **let** *var* *var* = *exp* **in** *exp*

Note, there was a bug in an earlier version of this grammar where the **let**-expressions did not have a body defined with the **in** keyword. You should use the correct definition above.

## 3 Using ocamllex

The `ocamllex` program is a lexer-generator. Given a description of terminal symbols as regular expressions, `ocamllex` generates a nondeterministic finite automaton to parse the input file (this is similar to the lexing we performed in lab3).

The lexing description is defined in the file `lex.mll`. If you run the command `ocamllex lex.mll`, you will generate the NFA code in the file `lex.ml`. This code is fairly unreadable, but you may be interested in looking at the construction of the NFA.

The `lex.mll` file contains four general sections, listed as follows.

```
{ (* Normal ML code *) }

(* Regular expression definitions *)
let white = [' ' '\t' '\n']+
let decimal = ['0'-'9']+
let octal = "0o" ['0'-'7']*
let number = decimal | octal

(* Lexing definitions *)
rule main = parse
  white
  { main lexbuf }
| number
  { TokInt (int_of_string (Lexing.lexeme lexbuf)) }
| "if"
  { TokIf }
| ...
| eof
  { TokEof }

{ (* More ML code *) }
```

**header** The header is enclosed in curly braces, and contains arbitrary ML code.

**regular expressions** For convenience, `ocamllex` allows you to define symbols for regular expressions. For example, the regular expressions define here are used to specify white-space (blanks) and decimal/octal numbers. The documentation describes the syntax for regular expressions.

**lexing definitions** The lexing definitions specify code that should be executed when a regular expression is matched. The definitions starts with the keyword **rule**; `ocamllex` will produce a function with the same name that takes a value of type `Lexing.lexbuf` and returns values computed by the rule clauses.

For example, consider the rules above. The first clause matches sequences of white-space characters. We want to discard them, so the clause ignores the match and recursively executes the main rule. The second clause matches expressions that look like numbers. The string that was matched is called the *lexeme*. The function `Lexing.lexeme lexbuf` gets the lexeme, and we use the `int_of_string` function to turn this into a number. Note, the `lexbuf` variable is a special variable that is implicitly defined in all rules.

The fourth clause matches a keyword. There are better ways to perform this matching; think about it.

Finally, the last clause matches the special end-of-file pattern.

**footer** Finally, the last part of the description contains some more arbitrary ML code.

## 4 Parsing

For parsing, we'll use the `ocamlyacc` parser-generator, which is a lot like `ocamllex`, but it takes a context-free grammar as input and produces a deterministic PDA. The parsing file is specified in the file `parse.mly`, which has the following general sections.

```

%{ (* Normal ML code *) %}

(* Token definitions (terminal symbols) *)
%token <int> TokInt
%token <string> TokId
%token TokAdd
%token TokSub
...

(* Precedence definitions *)
%nonassoc TokIf TokThen TokElse
%left TokLt TokLe TokGt TokGe TokEq TokNeq
...

(* Start symbol definition *)
%start prog
%type <Ast.prog option> prog

%%

(* Context-free grammar, with semantic actions *)
prog:      TokEof
          { None }
        | prog_exp TokDoubleSemi
          { Some $1 }
        ...
        ;

atomic:    TokInt
          { Int $1 }
        | TokId
          { Var $1 }
        | TokLeftParen exp TokRightParen
          { $2 }
        ;

apply:     atomic
          { $1 }
        | apply atomic
          { Apply ($1, $2) }
        ;

exp:       apply
          { $1 }

          /*
           * Binary operations.
           */
        | exp TokAdd exp
          { Binop (AddOp, $1, $3) }
        ...
        ;

```

**header** The first part of the file contains arbitrary ML code delimited with `%{` and `%}`.

**tokens** The second part defines the *terminal symbols* using the `%token` command. Each `%token` defines a tag for a terminal symbol with an optional value. For example, in this case, we

specify that the symbol `TokInt` has an `int` value, `TokId` is a string, and the remaining tokens do not have a value. If you look back at the lexing definition, you'll see that `TokInt` takes a `int`, and `TokIf` does not have a value.

**precedences** Note that the grammar is highly ambiguous. Yacc deals with this by defining *precedences* to help determine whether the PDA should shift or reduce. Here are the precedences of the operators. You will have to add these using the `%left` declaration for left-associative operators, and `%nonassoc` for non-associative operators.

Note, once you define precedence declarations, `ocaml yacc` should report no shift/reduce conflicts for this grammar.

Prec	Operator	Assoc
0	<b>let, in</b>	none
1	<b>fun, -&gt;</b>	none
2	<b>if, then, else</b>	none
3	<code>&lt;, ≤, &gt;, ≥, =, ≠</code>	left
4	<code>+, -</code>	left
5	<code>*, /</code>	left

**start symbol** The `%start` command is used to specify the start symbol. The `%type` specifies the value returned by the production. For this lab, the start symbol is `prog`, and the value of this production is `Ast.prog option`. Your parse should return `None` on end-of-file, or a value `Some p` where `p` has type `Ast.prog` otherwise.

**productions** The remainder of the file specifies the grammar as a set of productions. The syntax is a little different from the syntax we have used in class (it uses a single colon instead of the `::=` that we have been using), but it is essentially the same.

Each production has a *semantic action* that specifies code to be executed when the production is derived. For this lab, we want to use this code to build the expression that be being parsed. During parsing, each symbol on the right-hand-side of a production has a value. The values are denoted `$1`, `$2`, `...`. So, for example, if we parse the production `exp: exp TokAdd exp`, we want to produce a `Binop` expression. We get the two sub-expressions with `$1` and `$3`, and we combine them with the code `{ Binop (AddOp, $1, $3) }`.

## 5 Getting started

Check out the code as normal. You will need to implement parts of the `parse.mly` file and the `lex.mll` file.

The file `main.ml` defines a `ocamlc`-style toplevel that

- parses an expression,
- evaluates it, and
- prints the value.

Note that the programs have been extended to include lets that have no body (just like toplevel lets in OCaml). The environment is used to collect these values.

Here is an example session. Note that you don't have to use the `rec` keyword in function definitions. Also, you can leave the evaluator with `Control-D`.

```
<jyh:kenai 2469>./eval
# 1 + 2;;
```

```

3
# if 1 then 2 + 4 else 4 * 5;;
6
# let f x = x + 5;;
<fun>
# f 10;;
15
# let fact n =
if n = 0 then
  1
else
  n * fact (n - 1);;
<fun>
# fact 10;;
3628800
# <jyh:kenai 2470>

```

One more thing. This interpreter has terrible behavior on errors.

```

<jyh:kenai 2483>./eval
# let s = 1;;
1
# a + 2;;
Unbound variable "a"
Exit 2
<jyh:kenai 2484>./eval
# let f n = if n = 0 then 1 else n * g (n - 1);;
<fun>
# f 2;;
Unbound variable "g"
Exit 2
<jyh:kenai 2485>./eval
# let x$ = 1;;
Fatal error: exception Failure("illegal character: $")
Exit 2

```

As a general rule, you *should never* write an interpreter that does error handling as poorly as this. For example, you should report the line number for each error, or other helpful information. We won't do it for this lab, but imagine what would happen if we were parsing 1000 line files...

## 6 What to turn in

You should turn in your entire `eval` directory.

In addition, you should include the following.

- A README file explaining what you did, how it works, and whether you had any problems.
- The files `fact.out` and `fib.out` generated using your compiler with the `-print_ast` option.

```

% ./eval -print_ast fact.ml > fact.out
% ./eval -print_ast fib.ml > fib.out

```

- As usual, you should also include any additional tests that you think are important.