IIT CS440: Programming Languages and Translators

Homework 3: Top-down and Shift-Reduce Parsing

Prof. Stefan Muller
TA: Xincheng Yang

Out: Monday, Mar. 1
Due: Thursday, Mar. 11 11:59pm CST

This assignment contains 8 written tasks and 4 programming tasks, for a total of 65 points, in addition to a maximum of 1 bonus point.

0 Logistics and Submission - Important

The same rules as on HW2 apply. In particular:

1. Make sure you read and understand the updated/clarified collaboration policy on the course website.
2. The complicated skeleton code of this assignment will make testing in the `ocaml` toplevel or on Try-OCaml very difficult. Instructions for testing your code are provided in the writeups of the programming problems.
3. Your answers to the programming problems will go in the file `parse.ml`. You only need to submit this file (and your written pdf). Do not rename `parse.ml`.

1 LR Parsing: Mamma Mia!

Let $\Sigma = \{a, b\}$ and let L be the language generated by the grammar:

$$S \to \epsilon \mid a\ S\ a \mid b\ S\ b$$

Task 1.1 (Written, 6 points).

Give the parse tree for the string *abba*.

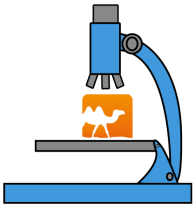
Task 1.2 (Written, 10 points).

Complete the table of configurations of a shift-reduce parser for the string *abba*.

Stack	Input	Action
\$	*abba*\$	shift

2 MicrOCaml

In this section, you will build a predictive parser for a tiny subset of OCaml, appropriately called MicrOCaml. The grammar of MicrOCaml is shown below:



$$
\begin{array}{rcl}
\textit{op} & \rightarrow & (+) \mid (-) \mid (*) \mid (/) \mid (<) \mid (>) \mid (<=) \mid (>=) \mid (=) \mid (\&\&) \mid (||) \\
\textit{var} & \rightarrow & [a-z, A-Z][a-z, A-Z, 0-9]^* \\
\textit{num} & \rightarrow & [0-9]^+ \\
\textit{const} & \rightarrow & \texttt{true} \mid \texttt{false} \mid \textit{num} \\
\textit{value} & \rightarrow & \textit{const} \mid \textit{op} \mid \texttt{fun}\ \textit{var} - > \textit{exp} \\
\textit{exp} & \rightarrow & \textit{var} \mid \textit{value} \mid \texttt{let}\ \textit{var}\ =\ \textit{exp}\ \texttt{in}\ \textit{exp} \mid \texttt{if}\ \textit{exp}\ \texttt{then}\ \textit{exp}\ \texttt{else}\ \textit{exp} \mid \texttt{app}\ \textit{exp}\ \texttt{to}\ \textit{exp}
\end{array}
$$

Note that the grammar above is the concrete syntax of MicrOCaml (without whitespace and comments, both of which are allowed and ignored by the lexer): it's what you'd type to write a program. MicrOCaml supports Booleans and integers, as well as operators on them, lambdas, let-bindings, if statements and function application. There are a few imporant major differences between MicrOCaml and OCaml: application is explicit, using, for example, `app f to x` instead of `f x`. In addition, there are no infix operators: while MicrOCaml supports many of the integer and Boolean operations of OCaml (addition, subtraction, multiplication, division, less than, greater than, less-or-equal, greater-or-equal, equal, Boolean and, and Boolean or), these must be applied as functions using the `(+)` syntax of OCaml.

We've already implemented a lexer for MicrOCaml in `parse.ml`. The lexer turns an input into a list of tokens. The tokens used are defined in the type `token` at the top of `parse.ml`. The grammar for MicrOCaml as seen by the parser is then:

$$
\begin{array}{rcl}
\textit{op} & \rightarrow & \textsc{plus} \mid \textsc{minus} \mid \textsc{times} \mid \textsc{div} \mid \textsc{lt} \mid \textsc{gt} \mid \textsc{le} \mid \textsc{ge} \mid \textsc{eqop} \mid \textsc{and} \mid \textsc{or} \\
\textit{const} & \rightarrow & \textsc{true} \mid \textsc{false} \mid \textsc{num} \\
\textit{value} & \rightarrow & \textit{const} \mid \textit{op} \mid \textsc{fun var arrow}\ \textit{exp} \\
\textit{exp} & \rightarrow & \textsc{var} \mid \textit{value} \mid \textsc{let var equal}\ \textit{exp}\ \textsc{in}\ \textit{exp} \mid \textsc{if}\ \textit{exp}\ \textsc{then}\ \textit{exp}\ \textsc{else}\ \textit{exp} \mid \textsc{app}\ \textit{exp}\ \textsc{to}\ \textit{exp}
\end{array}
$$

Note that the nonterminal *num* has been replaced by token NUM, which carries an integer, and the nonterminal *var* has been replaced by token VAR, which carries a string. The = sign in the "let" syntax has its own token (which is distinct from the token used for (=) as an operator, since these are two distinct language features even though they use the same concrete syntax!), as does the arrow `->` in the lambda syntax.

Task 2.1 (Written, 8 points).

Calculate the following (if the set is large, you may describe it rather than giving it explicitly):

(a) First(*const*)

(b) First(*op*)

(c) First(FUN VAR ARROW *exp*)

(d) First(VAR)

(e) First(*value*)

(f) First(LET VAR EQUAL *exp* IN *exp*)

(g) First(IF *exp* THEN *exp* ELSE *exp*)

(h) First(APP *exp* TO *exp*)

Task 2.2 (Written, 2 points).

Calculate Follow(exp).

Task 2.3 (Written, 3 points).

Explain why the grammar above for MicrOCaml is $LL(1)$.

It happens that the grammar of MicrOCaml is unambiguous, even without parentheses. You may have guessed that the reason for this is the odd `app` f `to` x syntax.

Task 2.4 (Written, 6 points).

Explain, in a short paragraph, why the grammar of MicrOCaml would become ambiguous and not $LL(1)$ if we replaced the production APP *exp* TO *exp* with *exp exp*, the equivalent construct for application in real OCaml. (Explain both why it would be ambiguous and why it would be not $LL(1)$.

2.1 Language Features and Compiler Pipeline

Before we write the parser, let's discuss a few more features of MicrOCaml that go beyond the syntax. For one thing, MicrOCaml is dynamically typed. This isn't necessarily obvious from the syntax (even though there's no syntax for types): remember, because of type inference, you can write an entire OCaml program without writing down any types, but the language is still statically typed. So why is MicrOCaml dynamically typed? Well, because we haven't written a type checker. This may seem like a silly answer, but it's more or less correct: we could write a type checker for MicrOCaml and make it a statically typed language. Instead, we choose to allow all syntactically valid programs and allow type errors to arise at runtime—this makes it dynamically typed. It also leads to some interesting results, as we'll now see.

We've provided several example MicrOCaml programs in the `examples` folder. Take a look at these to get a sense of how to write code in the language. You may also want to write a couple of your own programs for testing. MicrOCaml may seem very simple, but it is actually incredibly powerful[1]. One somewhat surprising feature is that, even though there is no syntax for `let rec`, we can still write recursive functions. The program `prog4.uml` is an infinite loop. The program `fact.uml` contains a factorial function (as written, the program calculates 5!, but you can change this by editing the last line). You don't really need to understand how or why this works right now (we'll cover it in class in a few weeks), but you're welcome to take some time to puzzle it out if you want. As a hint, the fact that MicrOCaml is dynamically typed is important here.

Bonus Task 2.5 (Written, 1 points).

Why can't we write `prog4.uml` in regular OCaml?
Hint: Think about what type `f` would have.

We also have not written an interpreter for MicrOCaml. This is not a principled decision so much as a logistical one: you're going to write this interpreter on the next assignment, so we can't give you the code for it now :). Instead, we've done the next best thing and given you a compiler from MicrOCaml to another dynamically typed functional language, Python[2].

Once you've written the parser (following these instructions now will just raise the `ImplementMe` exception), you can compile the compiler by running `make`, or if you don't have `make`:

```
ocamlc -o microml types.ml parse.ml print.ml topy.ml main.ml
```

Then run the compiler with

```
./microml <filename>.uml
```

where `<filename>.uml` is a MicrOCaml program (one of the examples we gave you or one you write yourself). The program will produce `<filename>.py`, which will contain entirely unreadable, but working, Python code. You can run it with

```
python3 <filename>.py
```

if you have Python 3 installed, or copy and paste it into an online Python interpreter, such as `https://www.python.org/shell/`.

[1] If you're familiar with this terminology, it's Turing-complete.
[2] Bet you've never thought of Python this way.

2.2 Writing the Parser

The parser's job is to convert a list of tokens (actually, you get slightly more than just the token, as we'll explain in a minute) into an abstract syntax tree. Abstract syntax trees for MicrOCaml are represented using the type definitions in `types.ml`:

```
type const = True | False | Int of int
type var = string
type op = Plus | Minus | Times | Div | Lt | Gt | Le | Ge | Eq | And | Or
type value = Const of const
           | Fun of var * exp (* fun x -> e *)
           | Op of op
and exp = Var of var
        | Value of value
        | Let of var * exp * exp (* Let (x, e1, e2) means let x = e1 in e2 *)
        | If of exp * exp * exp (* If (e1, e2, e3) means if e1 then e2 else e3 *)
        | App of exp * exp (* App (e1, e2) means app e1 to e2 *)
```

The file `parse.ml` contains a few other definitions you'll want. Remember how on Homework 0, we said one of the hardest parts of writing a compiler is producing good error messages? To help you do this, we've defined a type of locations `loc`, which is a pair of a line and column. Instead of producing just a list of tokens, the lexer actually gives you a list of values of type `tl`. A value of this type is a pair of a token and the location (line and column) in the code where that token starts. You'll want these for producing error messages.

```
type loc = int * int (* line, column *)
type tl = token * loc
```

We also define an exception `SyntaxError`, which you'll raise when a program doesn't parse correctly.
`exception SyntaxError of loc option * string`
You can include the location in the code where the error occurs and a string with a helpful message. We won't grade you on the helpfulness of your error messages, but believe us, spending a few minutes making them give you useful information will save you a lot of frustration when you're debugging.

Now, finally, you're ready to start writing the parser.

Task 2.6 (Programming, 6 points).

We've written a function `is_op : token -> bool` that returns true if and only if the provided token is an operator. Write two similar functions:

(a) `is_const : token -> bool` should return true if and only if the provided token is a constant (see the nonterminal *const* of the grammar).

(b) `is_first_val : token -> bool` should return true if and only if the provided token is in First(*value*).

You'll use these functions in writing the parser.

The predictive parser is, as usual, a collection of functions, each of which parses a particular nonterminal. Each function is of the form `parse_X : tl list -> X * tl list`, where `X` is one of the nonterminals of the grammar (var, const, op, value, exp)[3]. Each takes in a list of (token, loc) pairs, matches the appropriate nonterminal from the start of the list, and returns the appropriate abstract syntax tree node, along with the remaining tokens. We've implemented one of these, `parse_const` for you as an example. If the list is empty, it reports a syntax error. Otherwise, it takes the first token from the list and returns the appropriate constant, together with the tail of the list. If the first token is not a constant, it again raises a syntax error: this time, it reports the location where the constant should have been.

[3]OK, var isn't a nonterminal of the grammar we ended up with, but it's convenient to break it out into a separate function

Task 2.7 (Programming, 6 points).

Implement the functions `parse_var` and `parse_op`, following the pattern of `parse_const`. You may find the helper function `op_of_token` saves you some typing.

Task 2.8 (Programming, 4 points).

Implement the function `parse_exact : token -> tl list -> tl list`. This function doesn't exactly fit the pattern of the others: it takes a token as an additional input. If the first token of the input `tl list` is exactly the given token, it returns the rest of the tokens. Otherwise, it should report a syntax error.

The functions `parse_value` and `parse_exp` are more complicated and are mutually recursive, because the nonterminals *value* and *exp* are mutually recursive.

Task 2.9 (Programming, 14 points).

Implement the functions `parse_value` and `parse_exp`. We've implemented one case of `parse_value` for you (which also shows why you might want the `parse_exact` function you wrote above) and given you some of the structure of the rest. The implementation of `parse_exp` is up to you. Good luck.

2.2.1 Testing the Parser

Once you're done, you test your implementation using the instructions in Section 2.1. For debugging, you may, as usual, want to do unit testing using `assert` as well as print results to the screen. You can use `assert` statements as usual and these will be checked when you run `./microml <filename>`. You can also just check asserts (and not actually compile an input file) by running:

```
make assert
./microml-test
```

This will do nothing except run your asserts (and will produce no output if your asserts pass).

We've added one assert for you: parsing the expression

```
let f = fun x -> app x to x in app f to f
```

(the contents of `prog4.uml`) should give the result

```
Let ("f", Value (Fun ("x", App (Var "x", Var "x"))),
          App (Var "f", Var "f"))
```

If you wish to print expressions to the screen to help debug, you may find the function `pprint_exp : exp -> unit` in `print.ml` helpful: it prints the given expression to the screen as code and returns `()`. You can use it by calling `Print.pprint_exp`.

3 Standard Written Questions

Task 3.1 (Written, 0 points).

How long (approximately, in hours/minutes of actual working time) did you spend on this homework, total? Your honest feedback will help us with future homeworks.

Task 3.2 (Written, 0 points).

Who, if anyone, did you collaborate with (and in what way), and what outside sources, if any, did you consult in working on this homework?