MP 3 – A Lexer for MiniJava

CS 421 – Spring 2013

Revision 2.0

Assigned Thursday, January 31, 2013
Due Tuesday, February 5, at 9:30 AM
Extension 48 hours (penalty 20% of total points possible)
Total points 30

1 Change Log

1.0 Initial Release.

2.0 Completely new assignment after accidental release of solution.

2 Overview

After completing this MP, you will know how to implement a lexer using a DFA.

You are implementing a lexical analyzer for part of MiniJava, a subset of Java. We will continue to use MiniJava for a number of MPs in this class (including MP4, where you will implement a parser for it).

3 Overview of Lexical Analysis (Lexing)

Recall from lecture that the process of transforming program code (i.e, as ASCII or Unicode text) into an *abstract syntax* tree (AST) has two parts. First, the *lexical analyzer* (lexer) scans the text of the program and converts the text into a sequence of *tokens*, usually as values of a user-defined datatype. These tokens are then fed into the *parser*, which builds the AST.

Note that it is not the job of the lexer to check for correct syntax — that is done by the parser. In fact, our lexer will accept (and correctly tokenize) strings such as "`if if == if if else`" which are not valid programs.

4 Lexing with a DFA

As shown in lecture 5, a lexer can be derived straightforwardly from a DFA for the tokens. The states of the DFA are labelled mostly with tokens, but can also be labelled with "Discard" or "Error" (or "Ident_or_KW" in the notes, but we won't have keywords so that won't come up). Once you've got the DFA written, the rest is simple: repeatedly run the DFA from the start state until there are no moves, then take all the characters that were consumed and perform the action indicated by the label of the state you ended in: emit a token; discard the characters; or signal an error.

In this MP, we will have you write a simple DFA. Your main job will be to implement the DFA; we will provide code to run the DFA (analogous to the code on slide 15 of lecture 5–6, but written in OCaml). Specifically, you will provide the functions:

```
label_of_state: state -> {Discard, Error, Token}
action: state -> string -> token list
transition: char -> state -> state
```

The type `token` will be given below. `transition` gives the transition function of your DFA; we include the state `NoMove`, which is not actually a state but is just a way to say that there is no transition (the way we used -1 in the lecture notes). `action` return a *list* of tokens because that allows it to handle discard actions; that is, it will either return a list with a single token or an empty list.

To simplify the assignment, we will give you the `state` type, so you will only have to figure out the transition and implement the functions above.

Specifically, you will have to handle only the following types of token: identifiers, integers, floats, left bracket, and right bracket; in addition, you need to discard both C++-style and C-style comments.

The code we will provide includes a function `get_all_tokens`: `string` → `token list`, which calls the functions you need to write. Here are some examples:

```
# get_all_tokens " 34. {{} x";;
- : token list = [Errtok; Lbrack; Lbrack; Rbrack; Id "x"; EOF]
# get_all_tokens " 34ij ij34 34.4";;
- : token list = [Intlit "34"; Id "ij"; Id "ij34"; Floatlit "34.4"; EOF]
# get_all_tokens "/* comment */3";;
- : token list = [Intlit "3"; EOF]
# get_all_tokens "/*/* comment */3";;
- : token list = [Intlit "3"; EOF]
# get_all_tokens "/*/* comment */3 // comment\n";;
- : token list = [Intlit "3"; EOF]
# get_all_tokens "/*/* comment */3 // comment";;
- : token list = [Intlit "3"; Errtok; EOF]
# get_all_tokens "/*/* comment *";;
- : token list = [Errtok; EOF]
```

For errors, we are just returning `Errtok`, with no further description of the error (and we continue lexing). Note that a C++-style comment ends prematurely (and gives an error) if it does not end in an end-of-line, and a C-style comment ends prematurely if there is no closing "`*/`".

5 Problems

1. (30 pts) You are given the following types:

```
type token = Id of string | Intlit of string | Floatlit of string
           | Lbrack | Rbrack | Divide | Errtok | EOF

type state = Start
           | BadChar | Whitespace | Lbracket | Rbracket
           | Ident
           | Int | IntDot | Float
           | Slash | CPPcomment | CPPcommentEnd
                   | Ccomment | CcommentStar | CcommentEnd
           | NoMove

type actions = Discard | Error | Token
```

In the class notes, each state was assigned an integer, and each had a label from the type `actions`. Here, instead of using integers, each state will be assigned an element of type `state`; this is simply to make it easier to keep track of the states. We are giving you the state names, so you only have to decide on the transitions and then implement the transition function. We recommend that you draw the DFA on paper. Here is the meaning of the states:

`BadChar` : A character other than one of the accepted characters is seen. The accepted characters are *lower-case* letters, decimal digits, `{`, `}`, `/`, space and newline. Within a comment, all characters are accepted.

`Whitespace` : A single space or newline was seen.

`Lbracket` : `{` was seen.

`Rbracket` : `}` was seen.

`Ident` : In this assignment, an identifier is a lower-case letter followed by zero or more lower-case letters and decimal digits.

`Int` : A digit was seen.

`IntDot` : Some digits were seen, followed by a period.

`Float` : Some digits, a period, and some more digits (at least one) was seen. In this assignment, floats are just two non-empty integers separated by a period; there is no scientific notation.

`Slash` : `/` was seen.

`CPPcomment` : Two `/`'s were seen.

`CPPcommentEnd` : A newline terminating a C++ comment was seen.

`Ccomment` : A `/*` was seen — i.e. we are in the middle of a C-style comment.

`CcommentStar` : We were in the middle of a C-style comment, and we saw a star.

`CcommentEnd` : After being in a C-style comment, we saw a `*/`.

`NoMove` : As noted above, this is not actually a state, and will never be given as an argument to `transition`. But it should be returned from `transition` if there is no move from this state on the given input character.

That description is hopefully enough for you to implement the transition function:

```
transition: char -> state -> state
```

which is the main part of the assignment. The function

```
label_of_state: state -> actions
```

just says for each state what action should be taken (just as described in the lecture). And the function

```
action: state -> string -> token list
```

calls `label_of_state` and then takes the appropriate action. If the action is to return a token, it should return a list containing just that token; if the action is `Discard`, it should return an empty list; and if the action is `Error`, it should return a list containing just the token `Errtok`.

6 Running make and handin

The file mp3-skeleton.ml contains the types listed above and the definition of `get_all_tokens` (and several auxiliary functions). The definitions of the three functions you need to implement are given as stubs for you to fill in. As usual, copy mp3-skeleton.ml to mp3.ml and edit that file. To test against our test cases, run make and ./grader.

It is easy to run MP3 interactively. Since the skeleton is completely self-contained, and the inputs are simple, just start ocaml and enter `#use "mp3.ml";;` and you can run examples. You can enter `#load "solution.cmo";;` to run our solution (using `Solution.get_all_tokens`).

When you're ready — in particular, when you've passed all the tests in the tests file — put `mp3.ml` on the EWS machines and run `/home/cs421/handin -s mp3`.

7 Style Requirements

Starting with this MP, you will be expected to meet the following style requirements. Submissions that do not meet these requirements will not receive a grade until they are resubmitted with correct styling. (Note that these requirements will be checked manually by the grader and will not be enforced by handin, so acceptance by handin does not indicate correct style.)

- No long lines. Lines are 80 characters long.
- No tabs. Use spaces for indentation.
- Indents should be no more than 4 spaces, and must be used consistently.