dOvs - lexical analysis

Group 9
Miran Hasanagić - 20084902
Jakob Graugaard Laursen - 20093220
Steven Astrup Sørensen - 201206081

September 16, 2015

1 Introduction

This report describes the approach in order to develop the lexer, which is the first component in the compilation process. First, it will provide an overview of the concrete solutions to specific problems, such as handling nested comments, escape codes and multi-line strings. Additionally, it provides an overview of problems encountered and interesting experiments during the development process. Finally, five tiger programs are provided as test cases for the various elements of the lexer.

2 How did we implement the lexer

This lexer is implemented by using the existing tool ML-Lex. With this tool it is possible to specify which regular expression shall map to which specific tokens. At first the support for all the basic constructs in the Tiger language where implemented. For example this includes to recognise the keyword. Also the support for non nested comments was implemented in this first version of the lexer.

Following the book by using the ML-Lex tool was ...

2.1 Nested comments

Implementing the nested comments functionality was harder than the first version of the lexer without this functionality. In order to implement this functionality, the possibility to use states in ML-Lex was used. In this case a `COMMENT` state was used. Additionally, a counter was used in order to handle nested comments, and detected when we jump out of the `COMMENT` state. This is accomplished by the following code:

```
"/*"=> (commentLevel := !commentLevel+1; YYBEGIN COMMENT; continue());

and

<COMMENT>"*/"=> (commentLevel := !commentLevel-1; if !commentLevel < 1 then
YYBEGIN INITIAL else (); continue());
<COMMENT>.       => (continue());
```

This code basically shows that for each `/*` we meet, the counter is incremented by one. Afterwards, for each `*/` the counter is decremented by one. When the counter is at zero, we jump out of the `COMMENT` state. Additionally, this counter value is used in order to detect unclosed comments, and issue an error. In this case we use the function `eof`, which is called at the `EOF`. If we reach the `EOF` and the comment counter value is not zero, then an error is issued, as a comment has been made, but not closed.

2.2 Escape codes

The escape codes where implemented when the lexer recognises that it is in the `STRING` state. Afterwards when a backslash is found, it check the following character/sequence in order to recognise if it is legal inside a string according to the tiger language specification.

When inside the `STRING` state, the three basic basics chars are handled by:

```
<STRING> "\\n"|"\\t"|"\\" => (addToCurString yytext;continue());
```

Additionally, the three decimals after backslash are handled the following way:

```
<STRING> "\\"{digit}{3} => (addToCurString (digitEsc yypos yytext);continue());
```

There is also some error handling for the decimal in order to ensure that there are exact three digits after the backslash, as well as the digits being within the correct range of ASCII values of 0 - 127:

```
<STRING> "\\"{digit}{1,2} => (ErrorMsg.error yypos (yytext ^ " is an illformed
ascii decimal escape code. ascii decimal escape code must be of the form
\\ddd, with 0 <= ddd <=" ^ (Int.toString maxAsciiCode) ^ " , and d a digit
between 0 and 9"); continue());
```

At last the escape control chars where handled in order to convert these to their respective escape characteres. This achieved by recognising the character `\^`. Then a function handles to find if the control character is a valid one, and provide the correct mapping using the function `handleCtrl`:

```
<STRING> "\\^". => (addToCurString (handleCtrl yytext yypos);continue());
```

2.3 Multiline strings

For strings also the possibility to split the string over multiple lines is supported by the implemented lexer. This is handle by having a state called `MULTILINE`, which is entered from the before mentioned state `STRING`, when only a "\is typed inside a string. This change happens in the following code:

```
<STRING> "\\"(" "|"\t") =>(inMultiline := true; YYBEGIN MULTILINE; continue());
```

Inside this `MULTILINE` state we only allow space and tab to be used in order to format the string. Afterwards, this state is left when the next "\is received at the input of the lexer.

```
<MULTILINE> "\\" => (inMultiline := false; YYBEGIN STRING; continue());
```

Basically this functionality enables the support of multiline strings inside the lexer. There is, however, some additional error handling for multiline strings.

3 Problems experienced

One of the problems of the more interesting nature is how to handle EOF. While it is rather straightforward to make the lexer jump between relevant states, it quite another matter when wanting to query about it's current state. By looking through the documentation, there does not seem to be any built in way to do so.

This means that we decided to include boolean variables, that we can set when we enter and exit a state, for use in EOF to detect if, for example, a string is unclosed. We also use the commentLevel counter to detect if we still where in the COMMENT state.

A bizarre bug occurred when using a newline inside a comment. It caused everything to be considered a comment. The fix was going from

```
<INITIAL COMMENT MULTILINE>"\n" => (handleNewline yypos; continue());
```

to

```
<INITIAL MULTILINE>"\n" => (handleNewline yypos; continue());

<COMMENT>"\n" => (handleNewline yypos; continue());
```

4 5 tiger programs

These test show five tiger programs, which where used in order to test the lexer. It shall be noted that the main focus of these test are to test if error are reported correctly in different cases. Additionally, it is used in order to test if the lexer "jumps"correctly between the states described above. For this reason the tests are made simple. However, in order to test for more complicated tiger programs, the provided test cases can be used.

4.1 Test 1: Simple Comment+String

Here it is tested if the `COMMENT` state is entered correctly, and that the `STRING` state gets entered when necessary.

```
/* In this part we test a simple comment and string */

"Hello from a String"
```

The lexer behaves as expected, with the following output:

4.2 Test 2: Nested Comment+unclosed string

```
In this test a nested comment is tested, and that an error is issued because of the unclose

/* A nested
/* comment */ */

"This is a unclosed string
```

The lexer behaves as expected, with the following output:

4.3 Test 3: Nested comment, with error

In this part it is check if en error is issued when a nested comment is not closed correctly.

```
/* A nested
/* comment with an error */
```

The lexer behaves as expected, with the following output:

4.4 Test 4: Escape Characters, and Errors

```
"Testing tabs, backslash and newline: \t\n\\"
"Test ctrl char: \^@ \^[ \^? \^G \^H \^I \^J \^K \^L \^M"
"Test ill-formed ctrl char: \^A \^:"
"Testprintable ! # $ % ' ( ) * + , - . / : ;"
"Test digit escape: \0000 \027 \127 \007 \008 \009 \010 \011 \012 \013 u"
"Test printables again: < = > ? @ [ ] ^ _ ` { | } ~"
"Test illformed ascii code: \65 \9"
"\      \hi\
```

```
\ we only see a space between hi and we"
""
```

4.5 Test 5: Multiline Strings

```
/* Test
Some more text for the hell of it
correct
*/
let function hello() = print("Hello"/*a comment /* going */ deep */) in
hello end
```