University of Arizona, Department of Computer Science

CSc 453 — Assignment 1 — Due Mon Sep 14, 23.59 — 10%

Christian Collberg
September 3, 2009

1 Introduction

Your task is to write a lexical analyzer for the language Luca. Your program should be named `luca_lex`. `luca_lex` takes a Luca source program as input and produces a list of tokens as output, like this:

```
$ luca_lex prog1.luc > prog1.luc.out
```

The result is a list of tokens in XML format:

```
PROGRAM␣P;
␣␣␣CONST␣C␣:␣BOOLEAN␣=␣true;
BEGIN
␣␣␣WRITE␣"Hello!";
␣␣␣WRITE␣666;
␣␣␣WRITE␣6.66;
␣␣␣--␣Here's␣comment!
END.
```

⇒

```
<block>
   <TOKEN kind="PROGRAM" line="1"/>
   <TOKEN kind="IDENT" line="1" value="P"/>
   <TOKEN kind="SEMICOLON" line="1"/>
   <TOKEN kind="CONST" line="2"/>
   <TOKEN kind="IDENT" line="2" value="C"/>
   <TOKEN kind="COLON" line="2"/>
   <TOKEN kind="IDENT" line="2" value="BOOLEAN"/>
   <TOKEN kind="EQ" line="2"/>
   <TOKEN kind="IDENT" line="2" value="true"/>
   <TOKEN kind="SEMICOLON" line="2"/>
   <TOKEN kind="BEGIN" line="3"/>
   <TOKEN kind="WRITE" line="4"/>
   <TOKEN kind="STRINGLIT" line="4" value="Hello!"/>
   <TOKEN kind="SEMICOLON" line="4"/>
   <TOKEN kind="WRITE" line="5"/>
   <TOKEN kind="INTLIT" line="5" value="666"/>
   <TOKEN kind="SEMICOLON" line="5"/>
   <TOKEN kind="WRITE" line="6"/>
   <TOKEN kind="REALLIT" line="6" value="6.66"/>
   <TOKEN kind="SEMICOLON" line="6"/>
   <TOKEN kind="END" line="8"/>
   <TOKEN kind="PERIOD" line="8"/>
   <TOKEN kind="EOF" line="8"/>
</block>
```

2 Output Format

The output of `luca_lex` is a text file in *XML* format. There are essentially two kinds of entries: those that represent literals and identifiers contain an argument `value`, the rest do not:

```
<block>
   <TOKEN kind="PROGRAM" line="1"/>
```

```
   <TOKEN kind="EQ" line="2"/>
   <TOKEN kind="IDENT" line="1" value="P"/>
   <TOKEN kind="STRINGLIT" line="4" value="Hello!"/>
   <TOKEN kind="INTLIT" line="5" value="666"/>
   <TOKEN kind="REALLIT" line="6" value="6.66"/>
   <TOKEN kind="CHARLIT" line="6" value="C"/>
   <TOKEN kind="EOF" line="8"/>
</block>
```

The last token generated is always `EOF`. A complete list of tokens is given in Appendix A.

Line numbers are allowed to be "approximately" correct, i.e. they should be ± 1 from the "correct" line.

Be careful to get the output syntactically correct! Avoid extra spaces between `<` and `TOKEN` and `/` and `>`, for example.

3 Luca Lexical Rules

- Luca line comments start with a `--`-sign and extend to the end of the line. They are not included in the output of `luca_lex`.
- Luca structured comments start with a `(*` and must end with `*)`. They are not allowed to be nested. They are not included in the output of `luca_lex`.
- Luca is case-sensitive.
- Strings start and end with a `"`-character and cannot contain a `"`-character. They cannot extend past the end of a line.
- Character literals start and end with a `'`-character and must contain exactly one character (not a `'`).
- Identifiers consist of letters and digits, and must start with a letter.
- Integer literals consist of a sequence of digits.
- Real literals have the syntax

$$((\texttt{digit}*.\texttt{digit}+)|(\texttt{digit}+.\texttt{digit}*))(\texttt{E}(\backslash+|-)?\texttt{digit}+)?$$

  Examples of valid floating-point numbers:

  ```
  0.5    .5    5.   5.0   5.E-6   100.587E99
  ```

- Control characters other than tabs and newlines are not allowed in Luca source files.

4 Luca Lexical Errors

Instead of printing error messages, `luca_lex` generates these special *error tokens*:

```
ERROR_UNTERMINATED_STRING
ERROR_REALLIT
ERROR_ILLEGAL_CHARACTER
ERROR_UNTERMINATED_COMMENT
ERROR_UNTERMINATED_CHAR
ERROR_EMPTY_CHAR
```

Error messages can then be generated by the parser instead.

You don't have to do any error recovery. We won't test any of your output that appears after an error token.

5 Implementation Notes

- **This assignment can be coded in Java, C, or C++.** If you want to use another language, ask me first.
- Make sure that your `Makefile` is working properly. The TA will do the following, and nothing else:

  ```
  $ make
  $ luca_lex test1.luc > test1.luc.out
  ```

  In other words, if you're coding in Java you must provide a shell script called `luca_lex` that calls Java with the appropriate parameters.
- **You cannot use `lex` or any other similar lexical analysis generator for this assignment, either directly or indirectly.** I expect you to build the finite state machine by hand, and code it by hand.
- **You should build your scanner the way we've discussed in class, using a <u>table-driven</u> finite state automaton.** If you try anything else you will get 0 points.

6 Submission and Assessment

- The deadline for this assignment is Mon Sep 14, 23.59. It is worth 10% of your final grade.
- You should submit the assignment electronically to `d2l.arizona.edu`.
- You can work alone or in teams of 2. You must submit a `README` file that lists the members of your team and how much each team member contributed to the assignment.
- If you work in a team you should only submit one copy of the assignment.
- You can download 20 test cases from the class website: `http://www.cs.arizona.edu/~collberg/Teaching/453/2009/Assignments/index.html`. Each test case will give you 0 points if you get it wrong, 4 points if you get it right. No partial credits. You can see some of the test cases in Appendix B You can get an additional 20 points from more complicated "secret" test cases, for a total of 100 points.
- Two files `Char.java` and `Token.java` have been provided for you. Use them or not, it's up to you.

Don't show your code to anyone outside your team, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the TA or the instructor.

A Luca Tokens

token	**token_name**
+	PLUS
-	MINUS
*	STAR
/	SLASH
%	PERCENT
:=	COLONEQ
!	BANG
:	COLON
,	COMMA
[	LBRACK
]	RBRACK
(	LPAREN
)	RPAREN
.	PERIOD
;	SEMICOLON
^	CARET
@	ATCHAR
`	BACKQUOTE
AND	AND
OR	OR
=	EQ
>=	GE
>	GT
<	LT
<=	LE
#	NE
ISA	ISA
NARROW	NARROW
TRUNC	TRUNC
FLOAT	FLOAT
NOT	NOT
PROGRAM	PROGRAM
PROCEDURE	PROCEDURE
VAR	VAR
BEGIN	BEGIN
END	END

token	**token_name**
FOR	FOR
NEW	NEW
TYPE	TYPE
WRITE	WRITE
READ	READ
WRITELN	WRITELN
ENDFOR	ENDFOR
EXTENDS	EXTENDS
REF	REF
ENUM	ENUM
CONST	CONST
ARRAY	ARRAY
RECORD	RECORD
METHOD	METHOD
CLASS	CLASS
OF	OF
IN	IN
TO	TO
DO	DO
BY	BY
IF	IF
THEN	THEN
ELSE	ELSE
ENDIF	ENDIF
LOOP	LOOP
ENDLOOP	ENDLOOP
EXIT	EXIT
WHILE	WHILE
REPEAT	REPEAT
UNTIL	UNTIL
ENDDO	ENDDO
integer literal	INTLIT
real literal	REALLIT
string literal	STRINGLIT
char literal	CHARLIT
identifier	IDENT
	EOF

B Examples

```
[␣]␣(␣)
[]()
```

$\Longrightarrow$

```
<block>
  <TOKEN kind="LBRACK" line="1"/>
  <TOKEN kind="RBRACK" line="1"/>
  <TOKEN kind="LPAREN" line="1"/>
  <TOKEN kind="RPAREN" line="1"/>
  <TOKEN kind="LBRACK" line="2"/>
  <TOKEN kind="RBRACK" line="2"/>
  <TOKEN kind="LPAREN" line="2"/>
  <TOKEN kind="RPAREN" line="2"/>
  <TOKEN kind="EOF" line="2"/>
</block>
```

```
:
:=
```

$\Longrightarrow$

```
<block>
  <TOKEN kind="COLON" line="1"/>
  <TOKEN kind="COLONEQ" line="2"/>
  <TOKEN kind="EOF" line="3"/>
</block>
```

```
.12
34.
45.67
8.E5
.9E5
1.2E5
1.2e5
1.2e-5
1.2E+5
1.2E+99
```

$\Longrightarrow$

```
<block>
  <TOKEN kind="REALLIT" line="1" value=".12"/>
  <TOKEN kind="REALLIT" line="2" value="34."/>
  <TOKEN kind="REALLIT" line="3" value="45.67"/>
  <TOKEN kind="REALLIT" line="4" value="8.E5"/>
  <TOKEN kind="REALLIT" line="5" value=".9E5"/>
  <TOKEN kind="REALLIT" line="6" value="1.2E5"/>
  <TOKEN kind="REALLIT" line="7" value="1.2e5"/>
  <TOKEN kind="REALLIT" line="8" value="1.2e-5"/>
  <TOKEN kind="REALLIT" line="9" value="1.2E+5"/>
  <TOKEN kind="REALLIT" line="10" value="1.2E+99"/>
  <TOKEN kind="EOF" line="11"/>
</block>
```

```
1
12
123
1234
```

$\Longrightarrow$

```
<block>
  <TOKEN kind="INTLIT" line="1" value="1"/>
  <TOKEN kind="INTLIT" line="2" value="12"/>
  <TOKEN kind="INTLIT" line="3" value="123"/>
  <TOKEN kind="INTLIT" line="4" value="1234"/>
  <TOKEN kind="EOF" line="5"/>
</block>
```

```
"foo"
""
'a'
```

$\Longrightarrow$

```
<block>
  <TOKEN kind="STRINGLIT" line="1" value="foo"/>
  <TOKEN kind="STRINGLIT" line="2" value=""/>
  <TOKEN kind="CHARLIT" line="3" value="a"/>
  <TOKEN kind="EOF" line="4"/>
</block>
```

```
--␣this␣is␣a␣comment␣--
-␣--␣and␣here␣is␣another␣one
```

$\Longrightarrow$

```
<block>
  <TOKEN kind="MINUS" line="2"/>
  <TOKEN kind="EOF" line="3"/>
</block>
```

```
(*␣comment␣here␣*)
(␣*␣*␣)
(*␣(␣*␣comment␣here␣*␣)
*6␣and␣here␣*)
*)
```

$\Longrightarrow$

```
<block>
  <TOKEN kind="LPAREN" line="2"/>
  <TOKEN kind="STAR" line="2"/>
  <TOKEN kind="STAR" line="2"/>
  <TOKEN kind="RPAREN" line="2"/>
  <TOKEN kind="STAR" line="5"/>
  <TOKEN kind="RPAREN" line="5"/>
  <TOKEN kind="EOF" line="5"/>
</block>
```

```
''
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_EMPTY_CHAR" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
?
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_ILLEGAL_CHARACTER" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
.5E-x
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_REALLIT" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
.5E
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_REALLIT" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
'
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_CHAR" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
(*␣fooo
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_COMMENT" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
(*␣fooo␣*␣)
lots␣of␣stuff
here␣42


blah
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_COMMENT" line="1"/>
   <TOKEN kind="EOF" line="7"/>
</block>
```

```
"blah␣blah␣blah
42
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_STRING" line="1"/>
   <TOKEN kind="INTLIT" line="2" value="42"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
"blah␣blah␣blah
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_STRING" line="1"/>
   <TOKEN kind="EOF" line="1"/>
</block>
```

C For honor's section only

These extensions are for the honor's section only. Submit extensive test cases for each extension.

C.1 Nested comments

Add Modula-2-style nested comments:

```
(*
   This is a comment.
   (*
      This is a comment within a comment.
   *)
*)
```

It is a static error for comments to be unbalanced. That is, every `(*` must have a matching `*)`. Nested comments are not included in the output of `luca_lex`.

C.2 C-style escape-characters

Allow C-style escape-characters within string literals. You should support at least `\n`, `\"`, and `\t`.

C.3 Hexadecimal, octal, and binary integer constants

Allow hexadecimal, octal, and binary integer constants. Hexadecimal literals consist of the prefix `0x` followed by a sequence of hexadecimal digits (`0-9`, `A-F`, `a-f`). Upper and lower case hexadecimal digits may be freely mixed. Octal constants consist of the prefix `0o` followed by a sequence of octal digits (`0-7`). Binary constants consist of the prefix `0b` followed by a sequence of binary digits (`0,1`).