

MY802: CiScal Compiler

Due on Monday, March 13, 2017

George Manis

G.Zachos, A.Konstantinidis

March 5, 2017

Contents

1	About	3
1.1	CiScal language	3
1.2	CiScal Compiler	3
1.2.1	Using the compiler	3
2	Lexical Analyzer	4
2.1	Implementation details	5
2.1.1	Custom Classes	5
2.1.2	Data Structures	5
2.1.3	Return value	6
3	Syntax Analyzer	6

1 About

1.1 CiScal language

CiScal is a minimal programming language that has borrowed many of its characteristics from C and Pascal. In contrast to its limited capabilities, the development process of a CiScal compiler is quite interesting. In general, the language does support the following features:

- Numeric constants between -32768 and 32767
- if-else, do-while, while, select and assignment statements
- Relational and arithmetic expressions
- Definition of functions and procedures; both nested and not
- Parameter passing by reference and by value
- Recursive function/procedure calls

For more information on CiScal capabilities please refer to `ciscal-grammar.pdf`

On the other hand, the features below are not supported:

- for-loops
- Real numbers
- Characters and strings

1.2 CiScal Compiler

CiScal Compiler (CSC) was developed during the MY802 - Compilers course at the Department of Computer Science and Engineering, University of Ioannina. It is written in Python 3 and serves to transform CiScal source code to MIPS Assembly code.

1.2.1 Using the compiler

To learn how to use CSC run `./csc.py` and the information below will be printed to console:

```
Usage: ./csc.py [OPTIONS] {-i|--input} INFILE
```

```
Available options:
```

<code>-h, --help</code>	Display this information
<code>-v, --version</code>	Output version information
<code>-I, --interm</code>	Keep intermediate code (IC) file
<code>-C, --c-equiv</code>	Keep IC equivalent in C lang file
<code>--save-temps</code>	Equivalent to -IC option
<code>-o, --output OUTFILE</code>	Place output in file: OUTFILE

2 Lexical Analyzer

The Finite-State Machine (FSM) diagram in Figure 1 is a partial graphical representation of the finite automata implemented in the `lex()` function and which is used to convert the input sequence of characters into a sequence of language tokens. In addition to the states shown below, there are fourteen (14) more accepting states that correspond to characters: `'+', '-', '*', '/', '=', ',', ';', '{', '}', '(', ')', '[,]'` and `'eof'`. Transition to these states is triggered only from initial state s_0 .

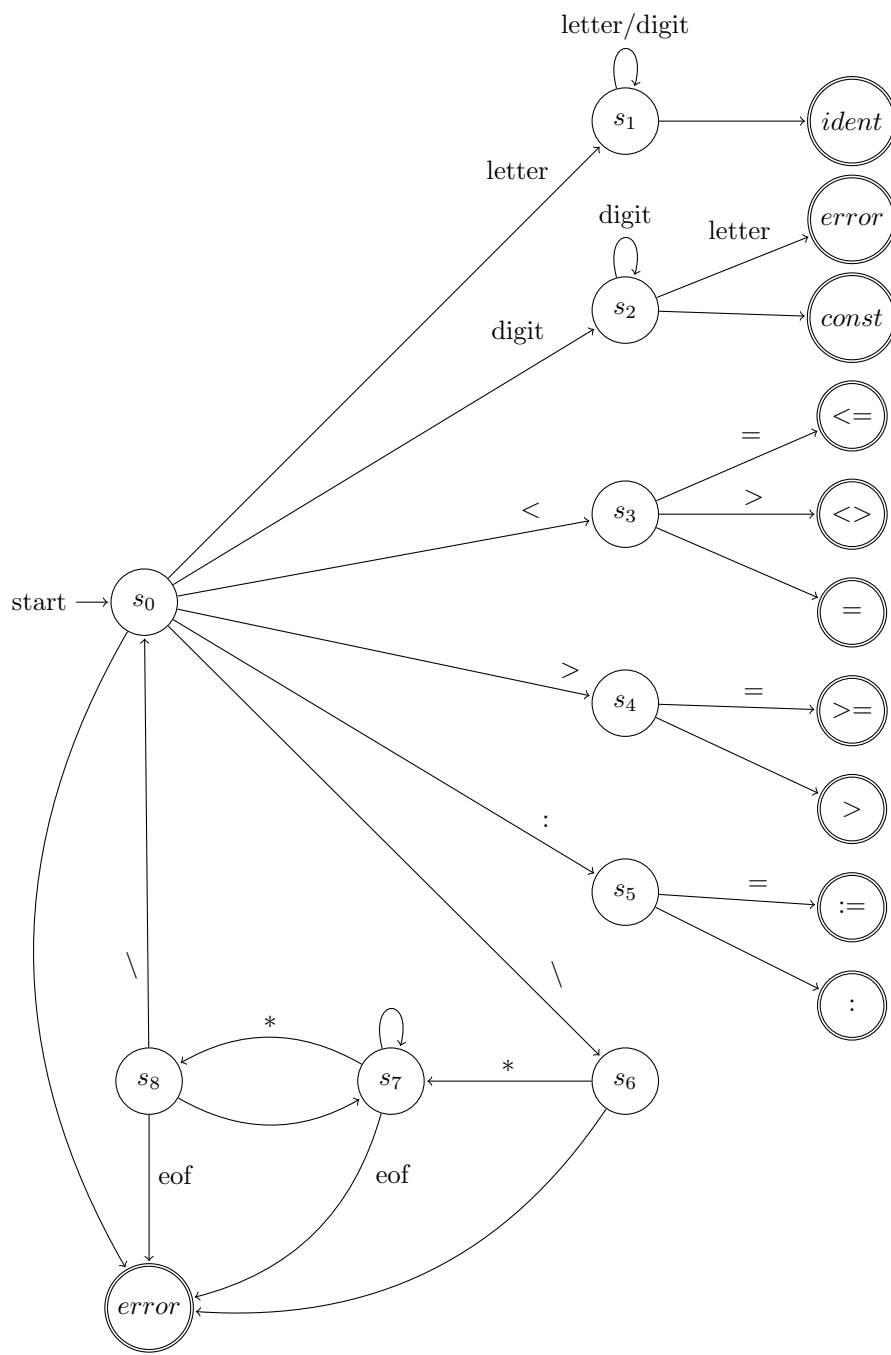


Figure 1: Partial FSM diagram of lexical analyzer's finite automata

2.1 Implementation details

2.1.1 Custom Classes

During the Implementation of the lexical analyzer, a couple of classes were defined. The first class defined was the TokenType class which is an enumeration that maps token types to enumerated constants. The other class was the Token class and was defined to group all useful information related to a token and that should be available to the syntax analyzer. This information includes:

- `tktype`: token type (attribute of the TokenType enumeration)
- `tkval`: the actual token value
- `tkl`: the line of the input file that the token was found
- `tkc`: the offset of the token's first character from the start of line `tkl`

2.1.2 Data Structures

The token dictionary maps actual keyword values to the corresponding TokenType attributes and serves code simplicity.

Listing 1: Token type/value dictionary

```

0 tokens = {
1     '(' : TokenType.LPAREN,
2     ')' : TokenType.RPAREN,
3     '{' : TokenType.LBRACE,
4     '}' : TokenType.RBRACE,
5     '[' : TokenType.LBRACKET,
6     ']' : TokenType.RBRACKET,
7     ',' : TokenType.COMMA,
8     ':' : TokenType.COLON,
9     ';' : TokenType.SEMICOLON,
10    '<' : TokenType.LSS,
11    '>' : TokenType.GTR,
12    '<=' : TokenType.LEQ,
13    '>=' : TokenType.GEQ,
14    '=' : TokenType.EQL,
15    '<>' : TokenType.NEQ,
16    ':=' : TokenType.BECOMES,
17    '+' : TokenType.PLUS,
18    '-' : TokenType.MINUS,
19    '*' : TokenType.TIMES,
20    '/' : TokenType.SLASH,
21    'and' : TokenType.ANDSYM,
22    'not' : TokenType.NOTSYM,
23    'or' : TokenType.ORSYM,
24    'declare' : TokenType.DECLARESYM,
25    'enddeclare' : TokenType.ENDDECLSYM,
26    'do' : TokenType.DOSYM,
27    'if' : TokenType.IFSYM,
28    'else' : TokenType.ELSESYM,
29    'exit' : TokenType.EXITSYM,
30    'procedure' : TokenType.PROCSYM,
31    'function' : TokenType.FUNCSYM,

```

```
32     'print':      TokenType.PRINTSYM,
33     'call':       TokenType.CALLSYM,
34     'in':         TokenType.INSYM,
35     'inout':      TokenType.INOUTSYM,
36     'select':     TokenType.SELECTSYM,
37     'program':    TokenType.PROGRAMSYM,
38     'return':     TokenType.RETURNSYM,
39     'while':      TokenType.WHILESYM,
40     'default':    TokenType.DEFAULTSYM,
41     'EOF':        TokenType.EOF }
```

2.1.3 Return value

The `lex()` function returns an object of type `Token` to the syntax analyzer.

3 Syntax Analyzer

...