MYY802: CiScal Compiler

Due on Monday, March 13, 2017

George Manis

G.Zachos, A.Konstantinidis

March 5, 2017

Contents

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 (integer) constans 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 MYY802 - 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:
        -h, --help              Display this information
        -v, --version           Output version information
        -I, --interm            Keep intermediate code (IC) file
        -C, --c-equiv           Keep IC equivalent in C lang file
        --save-temps            Equivalent to -IC option
        -o, --output OUTFILE     Place output in file: OUTFILE
```

2   Deliverables

- 1st Phase (10%)

  - Lexical Analysis
  - Syntax Analysis

- 2nd Phase (30%)

  - Intermediate Code Generation

- 3rd Phase (10%)

  - Semantic Analysis
  - Symbol Table

- 4th Phase (50%)

  - Final Code Generation (30%)
  - Project Report (20%)

3   Error Handling

To simplify display of error messages, the following functions have been defined:

- `perror_exit()`: Prints an error message to stderr and program exits

- `perror()`: Prints an error message to stderr

- `pwarn()`: Prints a warning to stderr

- `perror_line_exit(ec, lineno, charno, ...)`: Prints line <u>lineno</u> of the inputfile to stderr with character <u>charno</u> highlighted and along with and error message. Finally the program exits.

4 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.
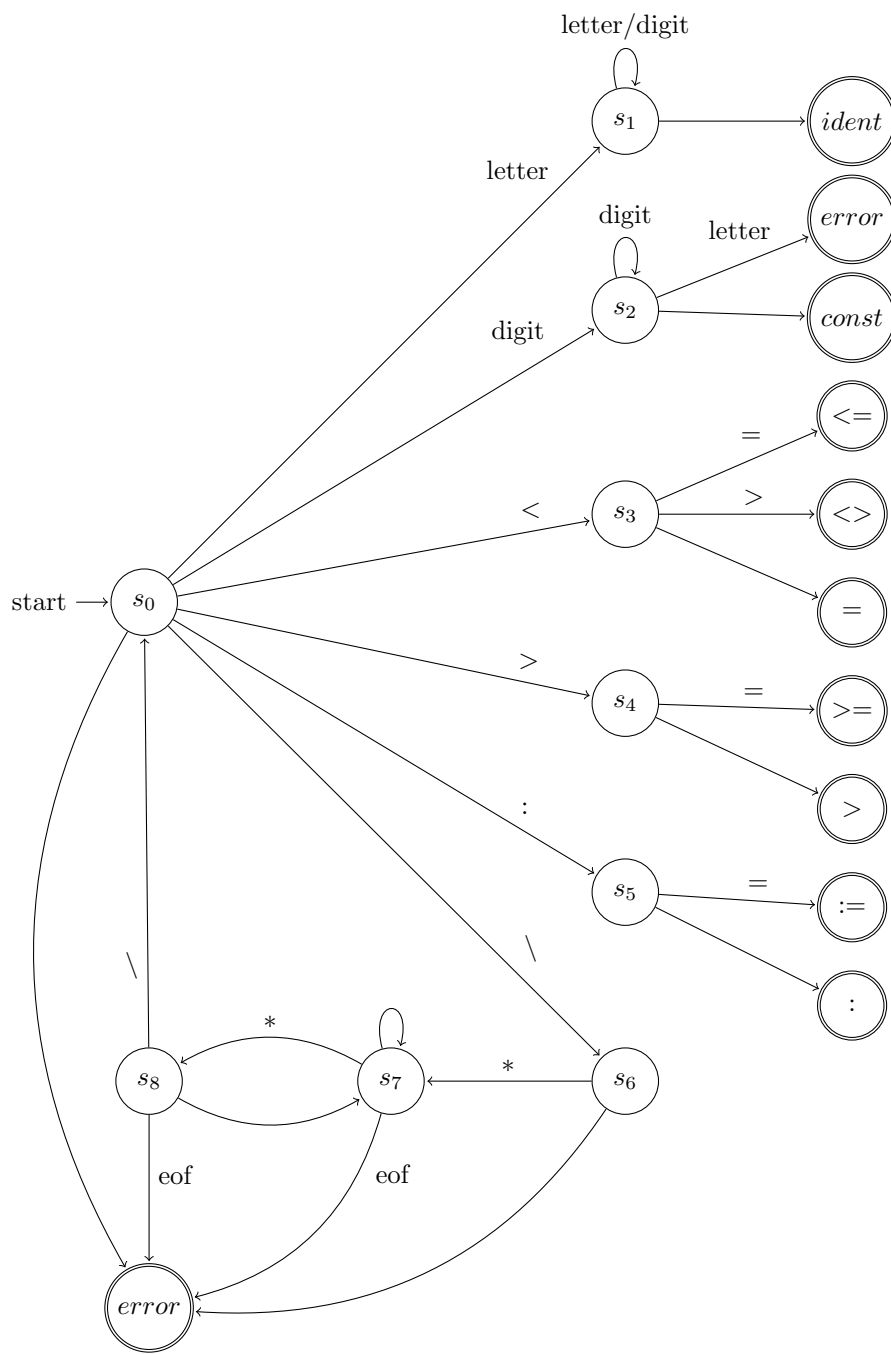
Figure 1: Partial FSM diagram of lexical analyzer's finite automata

4.1 Implementation details

4.1.1 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 `TokeType` 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`

4.1.2 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}
```

4.1.3   Return value

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

5   Syntax Analyzer

In order for the syntax analysis to take place, a function for each grammar rule was defined. In the syntax_analyzer() function, the global variable token is assigned the Token class instance returned by the first call to lex(). Then the program() function that implements the <PROGRAM> grammar rule is called and upon success, a last check takes place to ensure that EOF follows program end. All functions implementing grammar rules expect token to be ready for "consumption" and as a consequence replace it if needed.

6   Implementation specifics

6.1   Handling of numeric constants

CiScal source code files can contain (integer) numeric constants between -32768 and 32767 and an optional sign (+ or −) may precede the constant. In case an illegal constant value is found, an error message should be printed to console. Performing that check is a bit tricky for the following reasons:

- It cannot take place during lexical analysis - This happens because + and − are terminal symbols and as soon as they are identfied, should be returned to the syntax analyzer. Moreover, during lexical analysis there is no way to tell if + and − is a sign or an operator (To put it better, if it is a unary or a binary operator). Nevertheless, if the range $[-32768, 32767]$ was symmetrical, the check would be sign independent and could take place in the lex() function.

- The syntax analyzer expects from lex() to return a signed number - This is because of CiScal's grammar rules and specifically rule <TERM> ::= <FACTOR> (<MUL_OPER> <FACTOR>)* when <FACTOR> ::= CONSTANT. According to these rules, expressions like −2 * −3 will trigger a syntax error. In this example, after the multiplication operator the syntax analyzer expects a numeric constant but the subtraction operator will be encountered.

For the reasons stated, syntax rule <FACTOR> ::= CONSTANT | (<EXPRESSION>) | ID <IDTAIL> was changed to <FACTOR> ::= <OPTIONAL_SIGN> CONSTANT | (<EXPRESSION>) | ID <IDTAIL>. This change has no impact in the manipulation of numeric constants in the <SELECT-STAT> rule and consequently an optional sign preceding a case constant is illegal.