

MY802: CiScal Compiler

Due on Monday, May 24, 2017

George Manis

G.Zachos, A.Konstantinidis

March 20, 2017

Contents

1	About	3
1.1	CiScal language	3
1.2	CiScal Compiler	3
1.2.1	Using the compiler	3
2	Deliverables	4
3	Error Handling	4
4	Lexical Analyzer	5
4.1	Implementation Details	6
4.1.1	Custom Classes	6
4.1.2	Data Structures	6
4.1.3	Return Value	7
5	Syntax Analyzer	7
6	Implementation Specifics	7
6.1	Handling of numeric constants	7
7	Intermediate Code Generation	8
7.1	Implementation Details	8
7.1.1	Custom Classes	8
7.1.2	Global Data	8
7.1.3	Function Definitions	9

1 About

1.1 CiSca language

CiSca 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 CiSca compiler is quite interesting. In general, the language does support the following features:

- Numeric (integer) 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 CiSca 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 CiSca Compiler

CiSca 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 CiSca source code to Assembly code targeting the MIPS32¹ architecture. Moreover, transformation of the intermediate code to ANSI C code is available when possible.

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
```

Note: For the course purposes, the `--save-temps` option is enabled by default.

¹From this point on, we will refer to MIPS32 as MIPS

2 Deliverables

- 1st Phase (10%) [Due on March 13, 2017] [Delivered]
 - Lexical Analysis
 - Syntax Analysis
- 2nd Phase (30%) [Due on April 28, 2017] [Delivered]
 - Intermediate Code Generation
- 3rd Phase (10%) [Due on April 28, 2017] [Delivered]
 - Semantic Analysis
 - Symbol Table
- 4th Phase (50%) [Due on May 24, 2017]
 - 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 then program exits
- `perror()`: Prints an error message to `stderr`
- `pwarn()`: Prints a warning to `stderr`
- `perror_line_exit(ec, lineno, charno, ...)`: Prints line lineno of the inputfile to `stderr` with character charno highlighted and along with an 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 .

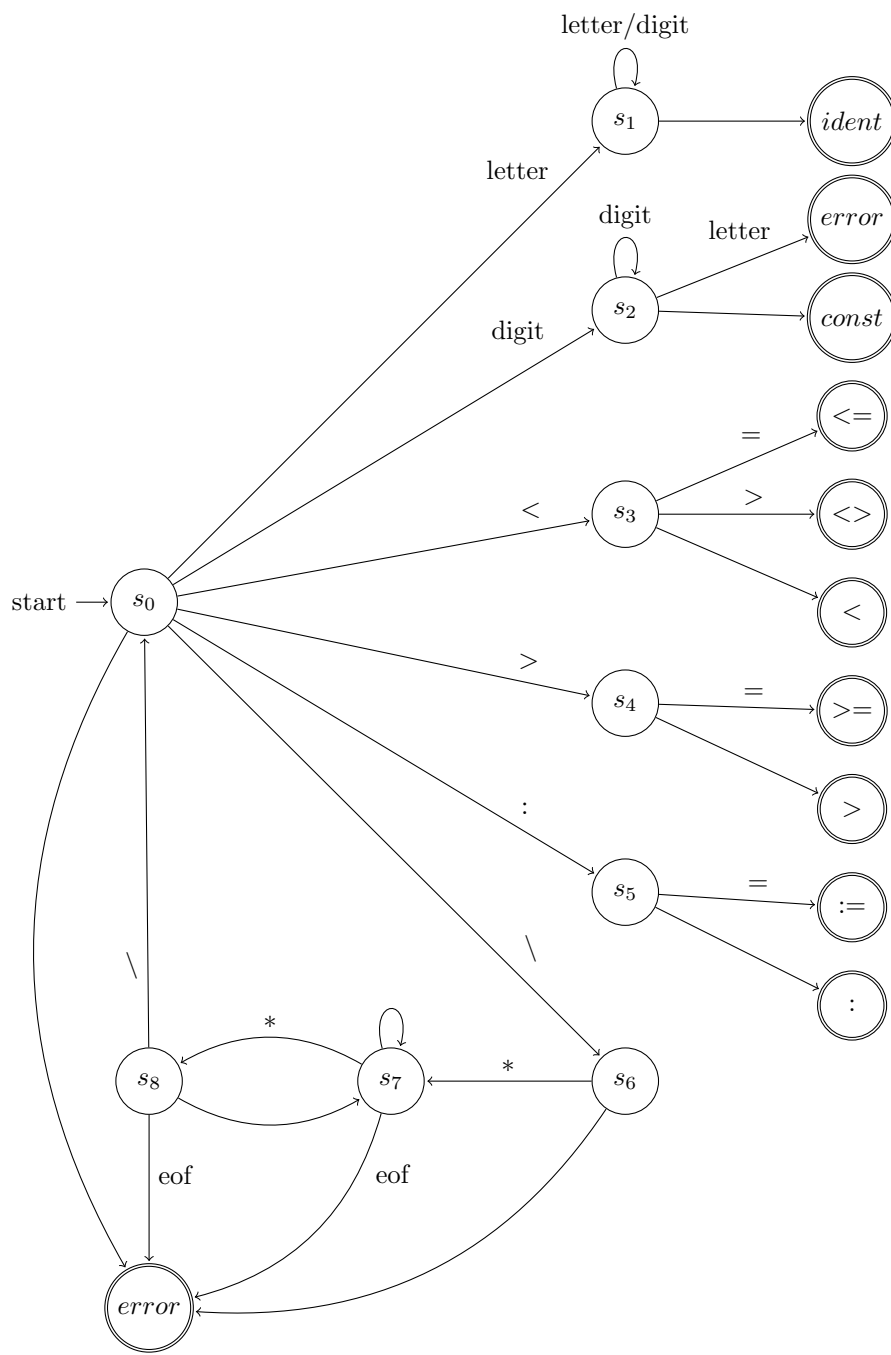


Figure 1: Partial FSM diagram of lexical analyzer's finite automata

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 `TokenType` enumeration)
- `tkval`: the actual token value
- `tkl`: the line number 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.RETURN_SYM,
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 `parser()` 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 identified, 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 these reasons, syntax rule `<FACTOR> ::= CONSTANT | (<EXPRESSION>) | ID <IDTAIL>` was changed to `<FACTOR> ::= <OPTIONAL_SIGN> CONSTANT | (<EXPRESSION>) | ID <IDTAIL>`.

Important: 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.

7 Intermediate Code Generation

Intermediate code, also known as intermediate representation is a code used internally by the compiler in order to represent source code. This code is then used to generate assembly instructions that will finally be converted to executable machine code. This intermediate representation is machine independent and is designed to be conducive for further processing, such as optimization² and translation. The intermediate representation used in the CiScaI Compiler is known as **three-address code**. The name derives because its instructions consist of at most three³ operands. The combination of three operands and one operator constitute a **quadruple** to which we will refer to as a quad. Each quad can be referenced using a label (non-negative integer). In addition to the three-address code generation, CSC supports generation of intermediate code equivalent in ANSI C and the generated code is ready to compile using the underlying system's C compiler. The transformation to ANSI C code takes place only when nested function definitions do not exist in user program, as it is not supported by the language specifications. On a later version of CSC, we plan to support nested functions using a GNU C extension provided by the GNU C Compiler⁴.

7.1 Implementation Details

7.1.1 Custom Classes

While implementing intermediate code generation, the `Quad` class was defined which serves to hold the information of a quadruple. This information is the following:

- `label`: the label to reference the quad.
- `op`: the operator to be applied to the source operands.
- `arg1`: operand #1, 1st source.
- `arg2`: operand #2, 2nd source.
- `res`: operand #3, destination.

7.1.2 Global Data

The following global variables and data structures were used:

- `nextlabel`: Holds the label (non-negative integer) which will be used to reference the next `Quad` to be generated.
- `next_tmpvar`: Used to implement the naming convention of temporary variables. (Temporary variable format: `T_<next_tmpvar>`).
- `tmpvars`: A dictionary holding temporary variable names.
- `quad_code`: A list that holds the `Quad` objects generated while parsing the source code of the user program.

²CiScaI Compiler does not perform any optimizations.

³Instructions with fewer operands may occur.

⁴<https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Nested-Functions>

7.1.3 Function Definitions

The following functions were defined while implementing the intermediate code generation:

- `next_quad()`: Returns the label of the next quad to be generated.
- `gen_quad(op, arg1, arg2, res)`: Generates a new quad and appends it to the current quad list representing input source code.
- `new_temp()`: Creates a new temporary variable named `Tn`, with $n \geq 0$ and returns its name.
- `empty_list()`: Creates a new (empty) list that will be used to hold quad labels and returns a reference to it.
- `make_list(label)`: Has the same effect of `empty_list()` but also adds `label` to the newly created list.
- `merge(list1, list2)`: Merges two lists holding quad labels.
- `backpatch(list, res)`: Replaces the `res` attribute⁵ of all the quadruples referenced by the labels in `list`.
- `generate_int_code_file()`: Generates the file containing the intermediate code of the user program.

For the transformation of the three-address code to ANSI C equivalent code, the following functions were further defined:

- `transform_to_c(quad)`: Transforms a quad to ANSI C code.
- `find_var_decl(quad)`: A naive way to find which variables are used as operands, starting from quad `quad` and until the end of the block. It returns a list with all the variable identifiers encountered. This will help during variable declaration.
- `transform_decls(vars)`: Given a list of variable identifiers, returns a string with a valid C variable declaration.
- `generate_c_code_file()`: Generates the file containing the ANSI C equivalent code of the intermediate code.

⁵Destination/3rd operand.