MYY802: CiScal Compiler

Due on Monday, May 24, 2017

George Manis

G.Zachos, A.Konstantinidis

March 24, 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 Assembly code targeting the MIPS32[1] 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.

[1] From 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] [Delivered]
  - 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 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.

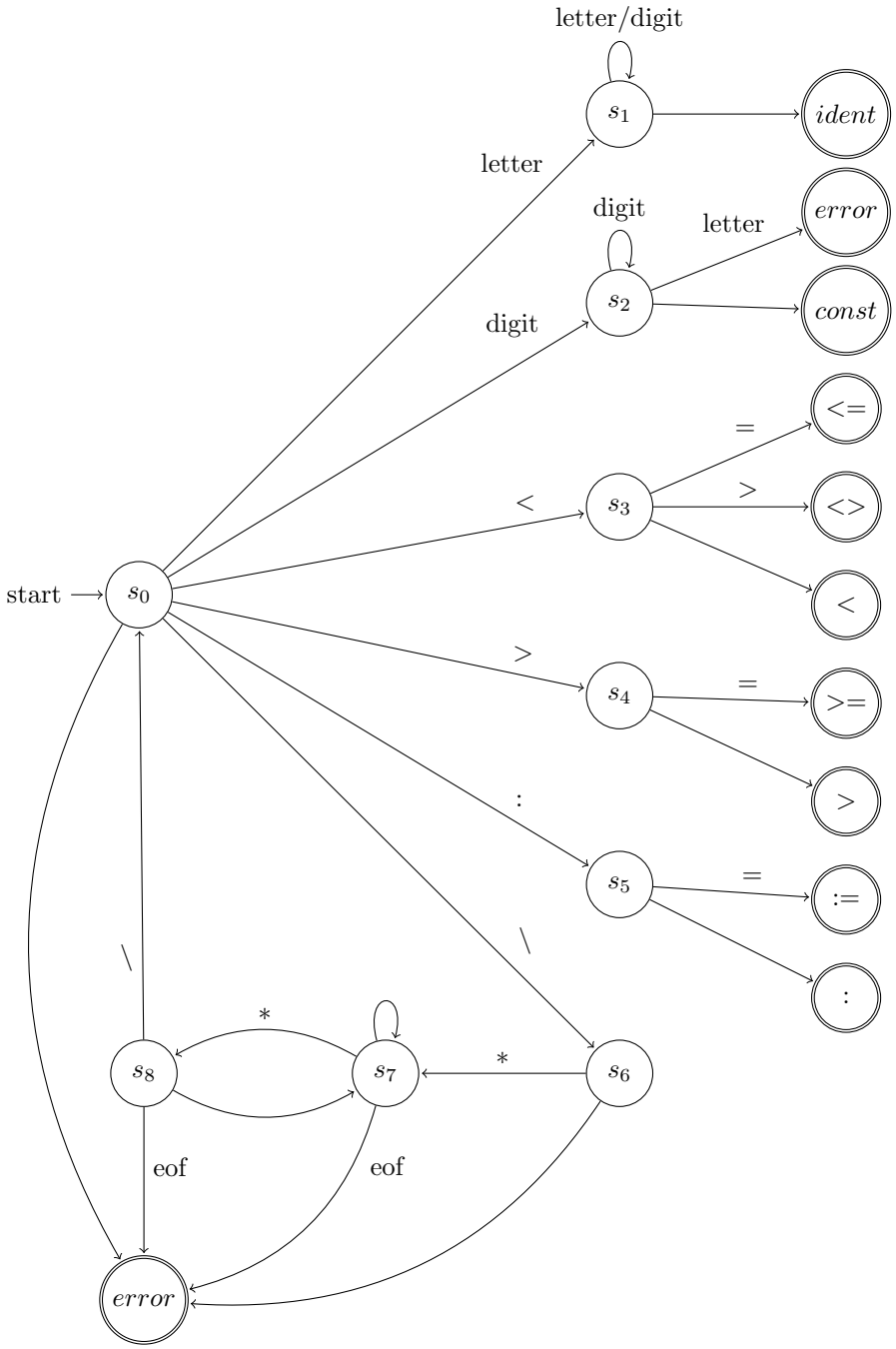

Figure 1: Partial FSM diagram of lexical analyzer's finite automata

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 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.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 `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 theses 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.

6.2 Exit Statement

The `exit` statement is supported in nested do-while loops. When it is encountered, the currently executing loop is immediately terminated and the program control resumes at the next statement following the loop.

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 optimization[2] and translation. The intermediate representation used in the CiScal Compiler is known as **three-address code**. The name derives because its instructions consist of at most three[3] 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 Compiler[4].

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.

[2] CiSal Compiler does not perform any optimazations.
[3] Instructions 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 `T_n`, with $n >= 0$ 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 newely created list.
- `merge(list1, list2)`: Merges two lists holding quad labels.
- `backpatch(list, res)`: Replaces the `res` attribute[5] 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.

[5] Destination/3rd operand.

8 Symbol Table

The symbol table is a data structure used by the compiler, where each identifier (a.k.a. symbol) in a program's source code is associated with information relating to its declaration or appearance in the source.

8.1 Implementation Details

8.1.1 Custom Classes

Implementing the symbol table, included definition of the following classes containing the corresponding attributes:

- `Scope`
  - `nested_level`: Nested level of scope.
  - `enclosing_scope`: Reference to a `Scope` object.
  - `entities`: List containing all the entities of the scope. New entities are appended at the tail of the list.
  - `tmp_offset`: Holds the current offset that results after taking into account all parameters, variables (temporary and explicit). It will later be used to update a function's framelength.
- `Argument`
  - `par_mode`: How the argument to the function is passed ("CV" or "REF").
  - `next_arg`: Reference to the following `Argument` object.
- `Entity`
  - `name`: Entity name (identifier returned by lexical analyzer).
  - `etype`: Entity type ("VARIABLE", "FUNCTION","PARAMETER","TMPVAR").
- `Variable` [Extends `Entity` Class]
  - `offset`: Offset of variable with regard to frame start.
- `Function` [Extends `Entity` Class]
  - `ret_type`: "int" for CiScal functions or "void" for procedures.
  - `start_quad`: Quad label of the first quad of the function.
  - `args`: A list containing all function arguments (`Argument` objects).
  - `framelength`: The function's framelength.
- `Parameter` [Extends `Entity` Class]
  - `par_mode`: "in" and "inout" for `TokenType.INSYM` and `TokenType.INOUTSYM` respectively.
  - `offset`: Offset of parameter with regard to frame start.
- `TmpVar` [Extends `Entity` Class]
  - `offset`: Offset of (temporary) variable with regard to frame start.

8.1.2 Global Data

The following global variables and data structures were used:

- `scopes`: A list holding `Scope` objects. Each new scope is appended at the end of the list.

8.1.3 Function Definitions

The following functions were defined while implementing the symbol table:

- `add_new_scope()`: Add a new scope.
- `print_scopes()`: Print current scope and its enclosing ones.
- `add_func_entity(name)`: Add a new function entity.
- `update_func_entity_quad(name)`: Update the start quad label of a function entity.
- `update_func_entity_framelen(name, framelength)`: Update the framelength of a function entity.
- `add_param_entity(name, par_mode)`: Add a new parameter entity.
- `add_var_entity(name)`: Add a new variable entity.
- `add_func_arg(func_name, par_mode)`: Add a new function argument to a given function.
- `search_entity(name, etype)`: Search for an entity named `name` of type `etype`.
- `search_entity_by_name(name)`: Search for an entity named `name`.
- `unique_entity(name, etype, nested_level)`: Check if entity named `name` of type `etype` at nested level `nested_level` is redefined.
- `var_is_param(name, nested_level)`: Check if a variable entity named `name` already exists as a parameter entity at nested level `nested_level`.

9 Final Code

The final code produced by the CiScal Compiler targets the MIPS[6] architecture and is ready to assemble using MARS 4.5 (MIPS Assembler and Runtime Simulator)[7].

9.1 Implementation Details

9.1.1 Global Data

The following global variables and data structures were used:

- `actual_pars`: A list holding `Quad` objects corresponding to subprogram parameters as discovered while traversing intermediate code. It is then used to validate parameter passing.

9.1.2 Function Definitions

The following functions were defined while implementing the symbol table:

- `gnvlcode(v)`: Load in register `$t0` the address of the non-local variable `v`.
- `loadvr(v, r)`: Load immediate or data `v` from memory to register `$t{r}` (e.g. `$t2`).
- `storerv(r, v)`: Store the contents of register `$t{r}` to the memory allocated for variable `v`.
- `gen_mips_asm(quad, block_name)`: Generate the assembly code for quad `quad`. `block_name` is the name of the block that is currently translated into final code.
- `check_subprog_args(name)`: Check if actual parameters of subprogram `name` are of the same type as typical parameters.

9.2 Implementation Specifics

During the stage of the final code generation, a number of changes have taken place[8].
These are the following:

- A number of assembler directives were added across the output file as shown below:

```
0  .globl L_x    # Declare that symbol L_x is global and can be
1                # referenced from other files. L_x is a text tag
2                # (label) pointing at the start of the main program.
3  .text         # The next items are put in the user text segment.
4  # ...
5  # Assembly code produced while translating intermediate code...
6  # ...
7  .data         # The following data items should be stored in
8                # the data segment.
9  newline: .asciz "\n"    # Store the string "\n" in memory and
10                         # null-terminate it. Use label 'newline'
11                         # to reference it.
```

[6] https://en.wikibooks.org/wiki/MIPS_Assembly/MIPS_Architecture
[7] http://courses.missouristate.edu/KenVollmar/mars/
[8] Without regard to a possible conflict with the handout.

- Instruction `add` (R format[9]) was replaced by `addi` (I format[10]) in all cases that immediates were used.
- Memory access related instructions (`lw`, `sw`)[11] were modified to be of the following format: `OP $rt, offset($rs)`, even when offset equals zero and can be omitted (e.g. `OP $rt, 0($rs)`).
- The final code produced by an `out` quad (`out, x, _, _`) is the following:

```
0  li $v0, 1   # service code 1: print the integer stored in $a0
1  li $a0, x   # load desired value into argument register $a0
2  syscall     # perform the service specified in $v0
```

  Nevertheless, `x` can be either a variable identifier or a numeric constant (immediate). In case of it being a variable name, instruction `li $a0, x` will fail to assemble. To fix this problem the produced code was modified to match:

```
0  # code generated by loadvr(x, '9')
1  li $v0, 1
2  add $a0, $t9, $zero  # move register content from $t9 to $a0
3  syscall
```

- For a more aesthetic result regarding the output of the print statement, three more instructions were added to the produced code of an `out` quad in order for a newline character to be printed after every integer print. The lines added are:

```
0  la $a0, newline  # load the addr. of the string pointed by label 'newline'
1  li $v0, 4        # service code 4: print the null terminated string at Mem[$a0]
2  syscall
```

- In MIPS architecture, stack grows from larger to smaller memory addresses[12]. Because MIPS does not provide an explicit push and pop instruction for manipulating the stack, modifying the stack pointer (`$sp` or `$29`) directly is required. Pushing to stack and poping from stack requires decreasing and increasing the value of `$sp` respectively. For this reason, instructions manipulating the value of stack and frame pointer were changed to follow this specification.
- The final code produced by a `retv` quad was modified to actually return program control to the caller and not only to access its frame and update the return value. This was done by adding:

```
0  lw $ra, 0($sp)   # Restore the value of $ra. In case of leaf procedures
1                   # it is not actually required.
2  jr $ra           # jump to the return address
```

- Finally, the `halt` quad results in the following assembly code:

```
0  li $v0, 10   # service code 10: exit (terminate execution)
1  syscall
```

[9] https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats#R_Format
[10] https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats#I_Format
[11] `lw` and `sw` are of I format
[12] https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html