TACi: Three-Address Code Interpreter (version 1.0)

David Sinclair

July 27, 2020

1 Introduction

TACi is an interpreter for Three-Address Code, the common *intermediate representation* (IR) used in compilers. **TACi** is written in Java using the compiler tools JavaCC and JJTree. There are many variation of Three-Address Code and this document describes the version of Three-Address Code supported by **TACi**.

2 Usage

To run the Three-Address Code interpreter on a file called *file.tac*, type:

```
java -jar TACi.jar file.tac
```

TACi can be invoked with the `-d` flag to run in debug mode. In this mode **TACi** will display the structure of the parsed AST, the final value of each variable, a list of the labels (entry points) in the program and the final location that each pointer references.

3 Three-Address Code

In Three-Address Code there is at most one operator on the right-hand of an instruction. Hence, in Three-Address Code, the valid instructions for expressions are:

$$\begin{array}{l} \mathrm{x} = \mathrm{y}\ \mathit{op}\ \mathrm{z} \\ \mathrm{x} = \mathit{op}\ \mathrm{y} \\ \mathrm{x} = \mathrm{y} \end{array}$$

Complex expressions in the source language can be translated in a sequence of Three-Address Code instructions using temporary variables. For example $w = x + y - z$ would be translated to:

t1 = y - z
w = x + t1

Three-Address Code is built on the concepts of addresses and instructions.

An *address* can be a *Name* or a *Constant*. The *Names* can include *temporary variables* created by a compiler, which are usually removed in subsequent optimization processes. In **TACi** *Names* are not explicitly typed. They are typed by the values assigned to them. The supported types in **TACi** are:

integers	A whole (non-fractional) number in the range $-2,147,483,648$ to $2,147,483,647$ inclusive.
floats	Real numbers in the range $-1.79769313486231570x10^{308}$ to $1.79769313486231570x10^{308}$ approximately.
booleans	**true** or **false**
strings	A sequence of alphanumeric characters (including the space character, the apostrophe character, the exclamation character and the backslash character) enclosed in double quotes, e.g. `"Hello there!"`.

3.1 Instructions

Each instruction is on a separate line.

3.1.1 Arithmetic & Boolean Instructions

x = y *op* z	Assignments where *op* is a binary arithmetic (e.g. `+,-,*,/`) or binary logical (e.g. `&&, ||`) operation.
x = *op* y	Assignments where *op* is a unary operation. Currently only logical negation is supported.
x = y	Copy instructions

To assign a negative value to a variable, say -4, use

```
x = 0 - 4
```

3.1.2 Branches

name:	Defines *name* as a label. A label must be defined by itself on a line. Every **TACi** program must have a `main:` label.
goto L	Unconditional jump to label *L*
if x *relop* y goto L	Conditional jump where control is passed to label *L* if *x relop y* is **true** and *relop* is a binary relational operator (e.g. `>, >=, == , != , <, <=`, etc.). Otherwise control passes to the next instruction.
ifz x *relop* y goto L	Conditional jump where control is passed to label *L* if *x relop y* is **false** and *relop* is a binary relational operator (e.g. `>, >=, ==, != , <, <=`, etc.). Otherwise control passes to the next instruction.

3.1.3 Functions

param x_n ... param x_1	The arguments to procedure and function calls are defined by the *param* instructions. Parameters are placed on the stack in reverse order.
x = getparam n	Returns a copy of the n-th parameter from the stack.
call p, n	An invocation of procedure p that takes n arguments. After the call to p the n parameters are cleared from the stack.
y = call p, n	An invocation of function p that takes n arguments. The result of the call to p is returned and stored in y. After the call to p the n parameters are cleared from the stack.
return	Passes control to the instruction following the *call* instruction that invoked the procedure p.
return x	Passes control to the instruction following the *call* instruction that invoked the function p. The value of x returned.

3.1.4 Arrays

TACi supports arrays by allowing an array indexed operation as a valid *name*. Hence, reading from an array and writing to an array are valid Three-Address Code instructions.

x = p[i] + 3	Reading from an array by an index value.
p[j] = x *op* y	Writing the result of x op y into an array by index value

Arrays are declared using *.data* directive, e.g.

```
p  .data   24
```

This example declares p as an array of size 24. Array indices start at 0. Accesses to arrays are bounds-checked by **TACi** to ensure that they are between 0 and $(size - 1)$ inclusive. Out-of-bounds array accesses generate a runtime exception.

TACi make no assumption on the size of each element in the array. You should assume that each element of the declared array holds a *byte*. Then if the targeted architectures store an integer in 4 bytes, to store the value 1 in the i-th element of an array called p you should use

```
t1 = i * 4
p[t1] = 1
```

where $t1$ is a temporary variable.

3.1.5 Pointers

TACi supports pointers and basic pointer arithmetic.

x = &y	stores the address of y in x. x is a pointer to y.
z = *x	The contents of what x points to can be accessed by $*x$. $*x$ is a *name* in **TACi**.
*x = y *op* z	Stores the result of y *op* z in the variable that x points to.

If a pointer is assigned the address of an array, it will point to the first element in the array. Hence `x = &p` and `x = &[p0]` are equivalent.

Basic pointer arithmetic using addition and subtraction are supported.

3.2 Library Procedures and Functions

TACi supports some basic library procedures and functions.

`_exit`	`_exit` takes no arguments and exits the parsed program.
`_read`	`_read` takes no arguments and returns the next item read from the console.
`_print`	`_print` takes one argument (from the stack) and displays it on the console.
`_println`	`_println` behaves as `_print` but also adds a newline character after displaying its argument.

3.3 Comments

Comments use the C++ style. They either begin with `/*` and end with `*/`, or they begin with `//` and continue to the end of the current line.

4 Example

The following is an example of a program to calculate the greatest common divisor written in Three-Address Code.

```
mod:                    // modulus function, returns parameter 1 mod parameter 2
   mx = getparam 1
   my = getparam 2
   mt1 = mx / my
   mt2 = mt1 * my
   mt3 = mx - mt2
   return mt3
```

```
gcd:                   // greatest common devisor function, Euclid's algorithm
   ga = getparam 1
   gb = getparam 2
gwb:
   ifz gb != 0 goto gwe
   gt = gb
   param gb
   param ga
   gb = call mod, 2
   ga = gt
   goto gwb
gwe:
   return ga
main:
   s1 = "Enter 1st number "
   param s1
   call _print, 1
   x = call _read, 0
   s2 = "Enter 2nd number "
   param s2
   call _print, 1
   y = call _read, 0
   param y
   param x
   answer = call gcd, 2
   os = "Answer is "
   param os
   call _print, 1
   param answer
   call _println, 1
   call _exit, 0
```

Assuming the file is called *gcd.tac*, then it an be interpreted as follows.

```
$ java -jar TACi.jar gcd.tac
Three Address Code Interpreter (TACi) v1.0
TAC source gcd.tac parsed successfully

Enter 1st number 1071
Enter 2nd number 462
Answer is 21
$
```