# Advanced Compiler Coursework

Harry Martin

Bachelor of Science in Computer Science with Honours
The University of Bath
December 2016

## 1 Section 1 Overview

For this coursework I created a interpreter, three address code (TAC) compiler and a MIPS compiler. The MIPS compiler using the TAC to create MIPS code. The generated MIPS code was designed to run on the emulator QTSPIM. The interpreter, TAC compiler and MIPS compiler were developed in parallel. It was especially important to develop the MIPS and TAC compilers in parallel as I could design TAC code that made the design challenges in the MIPS compiler easier. In this document I will explain how the interpreter and compilers work, show test cases of the interpreter and compiler working, criticise them and suggest further work.
The interpreter does a tree walk on the abstract syntax tree created by the provided code. A function called interpret recursively parses the tree. There are two types of branches an answer branch or a state changing branch. An Answer branch returns a value while a state changing branch only change the state of the program. The interpreter keeps track of what branches are answer branches and returns the value of an answer branch. An answer branch is a branch whose leaf is a RETURN statement. To implement environments(frame) I used a list of variables and there values. The list made dynamic assignment of variables possible. The frame of a child function is linked to the frame of it's parent function so that the child function can access variables in the parent function. Function pointer are placed on the frame. The name of the function pointer is mapped to the function's abstract syntax tree and the parent's frame if there is one.
The TAC compiler does a tree walk on the abstract syntax tree. TAC lines are created by the TAC compiler and placed in a list for the MIPS compiler. Therefore the TAC lines can be directly mapped to assembly code by the MIPS compiler. Therefore I purposely wrote the TAC code to map easily to MIPS assembly code. For example I did not allow integers to be added together without placing one of them into a register first. The MIPS assembly can only do arithmetic operations on two integers at a time therefore it was important for the TAC compiler to break down arithmetic operations. In the MIPS code

variables are transformed into memory addresses. The MIPS compiler uses the interpreter frame code to store variables and there memory addresses. Frames are created when functions are called and parent function frame are attached to child frames like in the interpreter. I used the interpreter frame system to reduce repeated code and because it meant implementing a feature only once.

# 2 Section 2: Description of Implementation

## 2.1 Interpreter

**Variable Binding** Variables bindings are store in a list called a frame. This frame is a list of symbols which corresponds to a value. A symbol and value pair can be updated or new symbol, value pairs added to the frame.

**Assignment Statements** When a node of type '=' is reached the interpreter evaluates the right node using the evalExp function. This function evaluates arithmetic expressions. This function is recursive and has two base cases. If the node is a integer then that value is returned. If that node is a variable then the value of that variable in the frame is returned. The recursive case is if the node has a type of an arithmetic operation (e.g. add). The operator is applied to the value of left and right nodes. Once this expression has been evaluated the value is bound to the symbol to the left of the '=' node.

**If statement** When a node is of type IF then the interpreter will evaluate the it's condition. The condition is calculated in a function evalCondition using a similar method to the evalExp function except it applies boolean operators rather than arithmetic operators. If the condition is true then the node to the left is evaluated. If the condition is false and there is an else statement then the node to the right is evaluated. If there is no else statement then this branch terminates.

**While statement** When a node is of type WHILE then the interpreter will firstly evaluate the condition. If the condition is true then the node to the left will be evaluated. This code will be continuously evaluated in till the condition evaluates as false.

**Return Statement** When a node with the type RETURN is parsed by the interpreter, it evaluates the value of the return statement by parsing the node to the left of the return statement. The branch is set to an answer branch. This done using an integer pointer. If the integer pointer is set to one then the branch is an answer branch if the pointer is set to zero then it's a state changing branch. This integer pointer is passed to the interpreter at each evaluation as an argument. When the interpreter is evaluating the program at a node with an non-leaf type it parses both the left and right nodes. It will then return the value in the node whose answer branch pointer is true. If both branches are

not answer branches then the answer branch pointer at this level of evaluation is set to zero.

**Functions** When a function is called the function evalFunction is called. Firstly the arguments are evaluated using the evalExp function. This will produce a set of values that need to be passed to the function. The function pushStack() in the interpreter is called which creates a new frame.The values of the passed arguments are bound to the parameter names in the function definition and added to the new frame.
When the program is first interpreted all of the function names and there abstract syntax trees are stored on a global frame, with exception to the main function which is evaluated intermediately. When a function is called the function is taken from the global frame and it's abstract syntax tree evaluated.
When a function returns the function popStack() is called which causes the program to go back to the frame that the program was using before the function was called.

**Nested Functions** Nested functions are functions that exist inside a parent functions. When the interpreter sees a node labelled as a function while parsing a function it will store that function name and abstract syntax tree on the parent function's frame. When that nested function is called it's frame is given the parent's function frame as it's context. This makes nested functions closures as they are functions with an environment. If a variable is not found in the nested function's frame then the parent's function frame will be searched. This is done recursively such that if the parent function is itself a child of other function then that function will have it's frame searched.

**Function pointers** Functions can be past or returned. The interpret function returns a union with two types of values. A integer or a closure. If a function pointer is being returned then the interpreter will return it's closure (abstract syntax tree and pointer to the parent function's frame). This closure can be called like any other function. With the abstract syntax tree being evaluated and it's parent frame being attached to a new frame. Function parameters evaluate to integers and closures. The passed function is then placed on the new frame and can be called inside the called function or any of it's child functions.

## 2.2 Three address code Compiler

**Arithmetic calculations** Arithmetic calculations are broken down into two forms: A = B + C or A = B. Temporaries are denoted as $tn(where n is a integer) are used to break down large arithmetic expressions. For example:

C–:
int a = 10 + 2 + 5 + 2;

Three address code:
$t2 = 10;
$t1 = $t2 + 2;
$t3 = $t1 + 5;
$t4 = $t3 + 2;
$t5 = $t4;
a = $t5;

When the TAC compiler encounters an arithmetic expression in the tree the left and right expression are evaluated. The base case are either a temporary being directly loaded with a variable or a integers . The recursive case are temporaries being added, subtracted, divided or multiplied to other temporaries or integers.

**If Statements** When an node of type IF is encountered then the compiler prints an if statement in the form if $tm. Where m is an integer. This temporary holds the value of the condition (1 or 0). The condition is evaluated before the if statement. The else blocked is printed before the then block. This makes the if statement more convenient to convert to assembly as a false if condition with cause the program to keep executing the code sequentially into the else block. The end of the then block is indicated by END THEN and the end of else block is indicated by END ELSE.

**While Statements** Firstly the condition is calculated above the while statement and placed in a temporary. Leaving a statement WHILE $tn where n is an integer. Underneath this line is the code to be executed with the block ending with the statement END WHILE.

**Functions** A node of type D indicates the start of a function. The name of the function is printed. Each of the function parameters are printed as PopArg n where n is the name of the parameter. The code inside is compiled and printed. At the end of the function the end of the function is indicated .end FUNCTIONNAME.

**Function calls** Firstly the arguments passed to the function are calculated and place into temporaries using the arithmetic methods described early. The temporaries containing the arguments are printed out with the command PushParam. A function call is described with the statement LCall and the return value placed in a temporary.

**Closures and Function Pointers** A child function is printed inside there parent function in similar fashion to how they were in the c– code. Whether a variable is located in a parent or child function is not indicated directly in the TAC code. This makes the TAC compiler simpler, leaving this task to the MIPS compiler that has the benefit of pre-compiled TAC code to help it understand

the code. The value returned by a function is placed inside a temporary. If the return value of a function is called in the C– code then the TAC compiler assumes that this is a function pointer. The closure of the function pointer is not directly indicated by the TAC code, this keeps this phase of compilation simple.

**Temporaries** There is a variable in the compiler that is incremented either time a new temporary is created. The temporaries are mapped directly to MIPS registers therefore the temporaries numbers cannot go over 9. When a new function abstract syntax tree is encountered the temporary counter is set back to zero. Every time a then block, else block and while block is entered the temporary counter is set back to zero. This reduces the number of registers used so that programs that use a lot of variabls can be run on QTSPIM.

## 2.3 MIPS Compiler

**TAC Compiler's output for the MIPS compiler** The TAC compiler as it prints TAC code places those TAC lines into a structure called TACLINE. This structure contains all of the information needed for each TAC line. One TAC line of code can produce more than one TACLINE structure. These TACLINEs are placed into a list and the MIPS compiler compiles these TACLINEs one by one.

**Variable** Variables are stored as memory locations. The interpreter frame code is used to map variables to address by the MIPS compiler. When a variable is initialized the last memory location used in the frame has four added to it (MIPS word size). This memory location is mapped to the variable. When a TAC line contains a variable the variable is substituted for the memory location.

**Arithmetic Expressions** Each TAC arithmetic expression is converted into a MIPS instruction. Below are the conversions. As discussed earlier the temporaries in the TAC code are directly mapped to the $t registers.
variable = register  sw register (variable's memory location)

register = integer  li register integer

register1 = register2  move register2 register 1

register = variable  lw register (variable's memory location)

register1 = register2 op integer  op_Instruction register1 register 2 integer
(where op is either +,-,/,*,¡,¿,== and op_Instruction is the corresponding MIPS instruction)

register1 = register 2 op register3  op_Instruction register1 register2 register3

The TAC compiler has been designed to only generate the above as QTSPIM cannot operate on two integers. For example if it sees a = 10 + 3. It will change it to $t1 = 10; a = $t1 + 3. This is a naturally recursively process hence was done by the TAC compiler which is compiled by a recursive function rather than by the MIPS compiler which is compiled in a linear fashion. Integers can not be directly placed into memory. Therefore a = 1; is changed to $t1 = 1; a = $t1 by the TAC compiler so that the integer is first placed into a register before it's saved into memory.

**IF Statement** Each if statement in the program is named with an integer. This means that the label of each if statement is uniquely named. The TAC compiler places the results of the if condition in a temporary. This makes testing the condition simpler for the MIPS compiler. The assembly instruction used is beq which branches if the two arguments are equal. The if statement is a beq statement testing if the value in the temporary containing the value of the condition is equal to one. If so then it jumps to the then block by using the label if_n(n is the if statement number). If the beq condition returns false then it will continue onto the else code directly below. At the end of the else code is a jump statement that jumps to the end of the then code. Below the then block is the next piece of code to execute once the if statement has been executed. If there is not an else block then then the jump statement is placed underneath the beq statement. The labels are noted in a TACLINE structure during the compilation of the TAC using a char that described what type of label it is.

**While Loop** The conditional statement of the while loop is compiled in a similar fashion to the if statement conditional. However the beq statement will branch if the temporary is equal to zero. The beq statement jumps to the end of the loop. The loop body is underneath the beq statement as it is executed if the temporary is equal to one. At the end of the loop body the assembly jumps back to the beq statement to repeat the process.

**Functions** When a TACLINE contains a function name then the compiler creates a frame. A frame is a segment of memory that a function can use to store it's passed parameters and variables. The frames are dynamically allocated. This so that this memory persists and can be used by closures. Meta-data about each function is calculated before the MIPS compiler starts. For each function the parameters, and variables counted. This tells the compiler how much memory to allocate a function. The frame pointer in $fp points to the current frame. The frame is allocated with a syscall with the pre-calculated memory allocation size. The frame pointer ($fp) and the return address($ra) are stored in the frame so that at the end of the function they can be recalled. Then the the address of the allocated memory is placed in $fp.
The compiler counts the maximum number of parameters passed to a function. A static piece of data is set aside in the .data section of the program with

enough memory for the maximum number of parameters. This static piece of memory's address is placed in the register $a1. The PopArg TAC line will cause the compiler to place the value in the address of $a1 into the frame. It needs to be placed inside the frame in case a function is called inside this function, which overwrites the static piece of data.
The function's code is then turned into MIPS. Once a return statement is compiled, the old values of $fp and $ra stored in the frame are placed back into $fp and $ra. The jal instruction is used on the $ra register to return to the calling function. The value being returned is placed inside $a0.

**Function Calls** The TAC line PushParam with cause the compiler to set up the parameters. If this is the first parameter then the memory location of the static parameter data is loaded into $a1. The parameters are then stored consecutively in the static parameters memory spaces each time a PushParam is seen. The values of parameters have to be in registers due to MIPS rules. This is enforced by the TAC compiler. The assembly to place a parameter in memory is sw $tn m($a1). Where n is the register number and m is the parameters number times 4 (word size in MIPS).
When the TAC line LCall is parsed the compiler prints the assembly command JAL with the function name. This will jump to the function's label. When control returns to the calling function the returned value in $a0 is placed inside a $t register.

**Programs Entry Point** QTSPIM runs the first function in the program. However in C– main is at the bottom therefore the compiler creates a function at the top of the program. This function is called _main. All _main does is call main then terminates the program with a syscall once main returns.

**Closures** Closures are a function with an attached environment. The compiler attaches the frame of the parent function to a child function. When a child function is being compiled the MIPS compiler frame containing variable to memory locations mappings from the parent function is attached to the child function's frame. When the compiler can not find a variable in the child's frame it will look in the parent's frame. This process is recursive for nested children functions. If the compiler finds the variable in the parent's frame then it will have to find the parent's frame pointer. To do this it loads the stored parent frame pointer from the child function frame into the register $v1. If the variable is in the frame of the parent function of the parent function then it will repeat this process to find the grandparent's frame. It will then find the variable in $v1 using the mappings in the relevant frame.

**Frame pointer in While and If statements** The frame pointer is stored at the next available space on the frame and the frame pointer is incremented to this address. This simulates a new frame without actually creating a new frame. Variables outside the if and while blocks are accessed in a similar method to

finding variables in a closure with the frame pointer to the enclosing function stored at the excepted place. When an if and while block are started a new compiler frame is created. At the end of the block the frame pointer is set back to the old frame pointer that was stored. When outside the if and while blocks the function code can overwrite this code as it's now out of scope.

**Function Pointers** Function pointers are a pointer to a closure. This consists of two parts. The address of the function's label and the parent's frame pointer. 8 bytes are dynamically allocated for the function address and parent function frame pointer. This binds these two address together and allows them to be both accessed by an single address. When a function pointer is returned the function pointer can be used in three ways. It can be placed in a variable where the function pointer is placed in a function's memory, it can be called like a normal function or it can be passed into another function as an argument. To call a function pointer the current frame pointer is saved in the register $s2. Then the frame pointer is set the environment in the function pointer. The function's address is placed in a register then called with the instruction Jalr. Once the control returns the frame pointer it is set back to the address saved in $s2. The address in $s2 is saved into the current frame before being overwritten. This allows for function pointers to be called in the function initially called. A function pointer can be passed as a parameter. The function pointer is an address so it takes up 4 bytes like integers and be placed in the static piece of data that store arguments. In the called function this passed function pointer can be called.

**Optimization** I have implemented a function that optimizes the TAC code before being compiled into MIPS. It uses the copy optimization techniques where statements such as $tn = $tm (where n and m are integers) are removes and all $tn are replaced with $tm in the basic block. Where a basic block is a block of code between a label and a jump. This reduces the number of loads and makes the MIPS code more efficient. This optimization phase is done after the initial TAC compilation as the compiler now has access to the entire TAC code representing the program.

# 3 Section 3: Three address code syntax and semantics

## 3.1 Arithmetic Expressions

Syntax: $tn = m (where n and m are integers)
Semantics: The temporary $tn is set to the value m.
Syntax: $tn = b (where n is an integer and n a variable)
Semantics: The temporary $tn is set to the value in the variable b.
Syntax: $tn = $tm (where n and m are integers)

Semantics: The temporary $tn's value is assigned to the value in temporary $tm.
Syntax: b = n (where b is an variable and n an integer)
Semantics: The variable b's value is changed to the integer n.
Syntax: b = $tn (where b is an variable and n an integer)
Semantics: The variable b's value is changed to the value in temporary $tn.
Syntax: M = A + B (where M is either a variable or temporary and A and B are either temporaries, variables or integers)
Semantics: The value of M is assigned to the sum of A and B.
Syntax: M = A - B (where M is either a variable or temporary and A and B are either temporaries, variables or integers)
Semantics: The value of M is assigned to the subtraction of A and B.
Syntax: M = A * B (where M is either a variable or temporary and A and B are either temporaries, variables or integers)
Semantics: The value of M is assigned to the product of A and .
Syntax: M = A / B (where M is either a variable or temporary and A and B are either temporaries, variables or integers)
Semantics: The value of M is assigned to the division of A and B.
Syntax: M = A ¡ B (where M is either a variable or temporary and A and B are either temporaries, variables or integers)
Semantics: The value of M is assigned to 1 if A is less than B and 0 otherwise.
Syntax: M = A ¿ B (where M is either a variable or temporary and A and B are either temporaries, variables or integers)
Semantics: The value of M is assigned to 1 if A is greater than B and 0 otherwise.
Syntax: M = A == B (where M is either a variable or temporary and A and B are either temporaries, variables or integers)
Semantics: The value of M is assigned to 1 if A is equal to B and 0 otherwise.

## 3.2 IF Statement

Syntax: IF $tn: (where n is an integer)
Semantics: If the value in $tn is 1 then jump to THEN otherwise jump to ELSE if an ELSE block has been defined.
Syntax: THEN
Semantics: Indicates the start of a THEN block.
Syntax: END THEN
Semantics: Indicates the end of a THEN block.
Syntax: ELSE
Semantics: Indicates the start of a ELSE block.
Syntax END ELSE
Semantics: Indicates the end of a ELSE block.

## 3.3 WHILE Loop

Syntax: WHILE $tn: (where n is an integer)

Semantics: If the value in $tn is 1 then execute the while loop code below otherwise jump to the code below the statement END WHILE.
Syntax: END WHILE
Semantics: Indicates the end of the while loop jump back to the while loop to retest the condition.

## 3.4 Functions

Syntax: FUNCTION_NAME: (where FUNCTION_NAME is the name of the function)
Semantics: Indicates the start of a function. Below is the function's code.
Syntax: END FUNCTION_NAME (where FUNCTION_NAME is the name of the function)
Semantics: Indicates the end of a function.
Syntax: RETURN A (where A is a function pointer, a integer, temporary or variable)
Syntax: Returns A to the function caller. A is an atomic element with all arithmetic calculation done above the return statement.
Syntax: PopArg b ( where b is variable)
Semantics: Pop's a parameter that was push by the calling function and places that integer or function pointer inside the variable b.

## 3.5 Function calling

Syntax: PushParam b (where b is a variable or temporary)
Semantics: Push's the value in b for the function being called to use as an argument.
Syntax: $tn = LCall FUNCTION_NAME (where FUNCTION_NAME is the name of a function and n is an integer)
Semantics: The function FUNCTION_NAME is called and it's return value is placed in the temporary $tn.
Syntax: $tn = LCall $tm (where FUNCTION_NAME is the name of a function and n and m are integers)
Semantics: Same as above except the function pointer in $tm is being called.

## 3.6 Function Children

Syntax:

```
FUNCTION_PARENT:
... code ...
FUNCTION_CHILD:
... code ...
END FUNCTION_CHILD
... code ...
END FUNCTION PARENT
```

(where FUNCTION_PARENT is the name of the parent function and FUNCTION_CHILD is the name of the child function)
Semantics: The child function is placed inside the parent function. Whether variables in the function child are in the child function or in it's parent function is not indicated in the TAC code.

# 4 Section 4 Test Cases

To keep this document relatively short I have only include four test cases for both the interpreter and the compiler. For all test cases both the interpreter and compiler produced correct solutions. The compiler tests in the appendices are done with the optimization turned off as they focus on the basic functions of the compiler. Below is a program compiled with the optimization on and off without _main code.
Optimization On:

```
f:
li $v0, 9
li $a0,16
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
lw $t2 8($fp)
add $t1 $t2 10
add $t3 $t1 2
sw $t3 12($fp)
lw $a0 12($fp)
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end f
main:
li $v0, 9
li $a0,20
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
li $t2 10
add $t1 $t2 5
add $t3 $t1 2
sw $t3 8($fp)
lw $t5 8($fp)
```

```
la $a1, params
sw $t5, 0(\$a1)
jal f
move $t6, $a0
move $a0, $t6
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end main
```

Optimization Off:

```
f:
li $v0, 9
li $a0,16
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
lw $t2 8($fp)
add $t1 $t2 10
add $t3 $t1 2
move $t4 $t3
sw $t4 12($fp)
lw $a0 12($fp)
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end f
main:
li $v0, 9
li $a0,20
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
li $t2 10
add $t1 $t2 5
add $t3 $t1 2
move $t4 $t3
sw $t4 8($fp)
lw $t5 8($fp)
la $a1, params
sw $t5, 0($a1)
jal f
```

```
move $t6, $a0
move $a0, $t6
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end main
```

The two instances of the line move $t4 $t3 have been removed with references to $t4 changed to $t3. Therefore this program will run slightly faster with the optimization on and a register has been freed for use.

# 5 Section 5 Evaluation of work

Global static memory was used to pass arguments. It would have been quicker to pass parameters in registers. Child functions are currently placed inside their parent function meaning that there is a jump in the parent function which is inefficient. If the child function was moved out of the parent function then there would be less jumps in the MIPS code. The MIPS compiler could have been made faster if instead of using a list for variable to memory location mappings it used a hash table. The address of the static memory is currently placed into $a1 multiple times in some programs. The compiler could have placed this once in the _main function.
The compiler could have used more optimization techniques. To optimize the MIPS code the compiler would have broken the code up into basic blocks where each block starts with a label and ends with a jump. Breaking the MIPS code up into basic blocks would make some optimization techniques easier to implement. For example the compiler could have used redundant instruction elimination where values that are loaded, stored in memory, then loaded again are replace by a single load. Or control flow optimizations were jumps to jumps are replace by a single jump. The compiler could used loop jamming where similar loop are merged. This would improve caching and reduces the loop overhead as you only have to run the loop overhead once.

## 5.1 Section 6 Further Work

Function cloning could have improved the compiler. Function cloning creates difference versions of the same function with each instance of that function being optimized for a certain set of parameters. This paper[1] describes function cloning that optimises functions using optional parameters. Different cases of the same function are created based on whether certain optional parameters are used. Some of these functions would contain less instructions than the original function. At compile time each function is created with a present() function to check if a optional parameter is used. At run-time the present function indicates which function to use. For this optimization to be worth it the number of instructions saved must be more than the number of functions in the present function. In future work the compiler could be expanded to use

optional parameters and use a similar optimization technique to improve the speed of the programs with optional parameters.
The MIPS compiler could use delay slots. Delay slots are instructions that are executed while the program branches as branching take 2 cycles. The instruction above a branch would be moved below the jumps. This would make programs more efficient as they complete more instruction in a given number of cycles.
Currently if there is a program that uses a lot of variables then the compiled code will not run as there will be more registers used than exists in QTSPIM. To fix this problem the compiler would initially compile the program as it does now but then go though the code changing registers that do not exist to memory locations in a function's allocated memory.

# 6 Appendices

## 6.1 Interpreter Tests

```
function cplus(int a){
        int add(int b){return a + b;}
        return add;
}

int main(){

        return cplus(5)(5);
}
```

—C INTEPRETER

Answer: 10

```
int fact(int n) {
        int inner_fact(int n, int a) {
                if (n==0) return a;
                return inner_fact(n-1,a*n);
        }
        return inner_fact(n,1);
}

int main(){

        return fact(5);
}
```

—C INTEPRETER

Answer: 120

```
function twice(function f,int n) {
        int g(int x) { return f(f(x*n)); }
        return g;
}

int m(int a){

        return a*5;
}

int main(){

        return twice(m,5)(5);
}
```

—C INTEPRETER

Answer: 625

```
function doNTimes(int n,function f){

        int innerDoNTimes(int a){
                while(n > 0){

                        a = f(a);
                        n = n - 1;
                }

                return a;
        }

        return innerDoNTimes;
}

int m(int a){

        return a*5;
}

int main(){
```

```
        return doNTimes(5,m)(5);
}
```

—C INTEPRETER

Answer: 15625

## 6.2 Compiler Tests

```
function cplus(int a){
int add(int b){return a + b;}
return add;
}

int main(){

return cplus(5)(5);
}
```

—C COMPILER

```
cplus:
PopArg a;
add_:
PopArg b;
$t2 = a;
$t3 = b;
$t1 = $t2 + $t3;
Return  $t1;
.end add_
Return  add_;
.end cplus
main:
$t1 = 5;
PushParam $t1;
$t2 = LCall cplus;
$t3 = 5;
PushParam $t3;
$t4 = LCall $t2;
Return  $t4;
.end main



.data
params: .word 0
```

```
.text
.globl  main
_main:
jal main
li $v0,10
syscall
.end _main
cplus:
li $v0, 9
li $a0,12
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
j endadd_
add_:
li $v0, 9
li $a0,12
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
move $v1, $fp
lw $v1, 0($v1)
lw $t2 8($v1)
lw $t3 8($fp)
add $t1 $t2 $t3
move $a0, $t1
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end add_
endadd_:
li $v0, 9
li $a0, 8
syscall
la $t0, add_
sw $t0, ($v0)
sw $fp, 4($v0)
la $a0, ($v0)
lw $ra 4($fp)
lw $fp 0($fp)
```

```
jr $ra
.end cplus
main:
li $v0, 9
li $a0,24
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
li $t1 5
la $a1, params
sw $t1, 0($a1)
jal cplus
move $t2, $a0
li $t3 5
la $a1, params
sw $t3, 0($a1)
sw $s2,8($fp)
move $s2, $fp
lw $fp, 4($t2)
lw $t0, 0($t2)
jalr $t0
move $fp, $s2
lw $s2, 8($fp)
move $t4, $a0
move $a0, $t4
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end main
```

Run: $a0 = 10

```
int fact(int n) {
int inner_fact(int n, int a) {
if (n==0) return a;
return inner_fact(n-1,a*n);
}
return inner_fact(n,1);
}

int main(){

return fact(5);
}
```

```
--C COMPILER
fact:
PopArg n;
inner_fact:
PopArg n;
PopArg a;
$t2 = n;
$t1 = $t2 == 0;
IF $t1:
THEN:
Return  a;
END THEN
$t3 = n;
$t2 = $t3 - 1;
PushParam $t2;
$t5 = a;
$t6 = n;
$t4 = $t5 * $t6;
PushParam $t4;
$t7 = LCall inner_fact;
Return  $t7;
.end inner_fact
$t8 = n;
PushParam $t8;
$t8 = 1;
PushParam $t8;
$t9 = LCall inner_fact;
Return  $t9;
.end fact
main:
$t1 = 5;
PushParam $t1;
$t2 = LCall fact;
Return  $t2;
.end main



.data
params: .word 0, 0
.text
.globl  main
_main:
jal main
li $v0,10
syscall
```

```
.end _main
fact:
li $v0, 9
li $a0,12
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
j endinner_fact
inner_fact:
li $v0, 9
li $a0,20
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
lw $t0, 4($a1)
sw $t0,12($fp)
lw $t2 8($fp)
seq $t1 $t2 0
beq $t1 1 if_1
j end_1
if_1:
sw $fp, 16($fp)
add $fp, $fp, 16

move $v1, $fp
lw $v1, 0($v1)
lw $a0 12($v1)
move $v1, $fp
lw $v1, 0($v1)
lw $ra 4($v1)
move $v1, $fp
lw $v1, 0($v1)
lw $fp 0($v1)
jr $ra
lw $fp, 0($fp)
end_1:
lw $t3 8($fp)
sub $t2 $t3 1
la $a1, params
sw $t2, 0($a1)
```

```
lw $t5 12($fp)
lw $t6 8($fp)
mul $t4 $t5 $t6
sw $t4, 4($a1)
jal inner_fact
move $t7, $a0
move $a0, $t7
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end inner_fact
endinner_fact:
lw $t8 8($fp)
la $a1, params
sw $t8, 0($a1)
li $t8 1
sw $t8, 4($a1)
jal inner_fact
move $t9, $a0
move $a0, $t9
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end fact
main:
li $v0, 9
li $a0,8
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
li $t1 5
la $a1, params
sw $t1, 0($a1)
jal fact
move $t2, $a0
move $a0, $t2
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end main
```

Run: $a0 = 120

```
function twice(function f,int n) {
int g(int x) { return f(f(x*n)); }
```

```
return g;
}

int m(int a){

return a*5;
}

int main(){

return twice(m,5)(5);
}
```

—C COMPILER

```
twice:
PopArg f;
PopArg n;
g:
PopArg x;
$t4 = x;
$t5 = n;
$t3 = $t4 * $t5;
PushParam $t3;
$t6 = LCall f;
PushParam $t6;
$t7 = LCall f;
Return  $t7;
.end g
Return  g;
.end twice
m:
PopArg a;
$t2 = a;
$t1 = $t2 * 5;
Return  $t1;
.end m
main:
$t1 = m;
PushParam $t1;
$t1 = 5;
PushParam $t1;
$t2 = LCall twice;
$t3 = 5;
PushParam $t3;
$t4 = LCall $t2;
Return  $t4;
```

```
.end main



.data
params: .word 0, 0
.text
.globl  main
_main:
jal main
li $v0,10
syscall
.end _main
twice:
li $v0, 9
li $a0,16
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
lw $t0, 4($a1)
sw $t0,12($fp)
j endg
g:
li $v0, 9
li $a0,28
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
lw $t4 8($fp)
move $v1, $fp
lw $v1, 0($v1)
lw $t5 12($v1)
mul $t3 $t4 $t5
la $a1, params
sw $t3, 0($a1)
move $v1, $fp
lw $v1, 0($v1)
lw $v1, 8($v1)
sw $s2,12($fp)
move $s2, $fp
```

```
lw $fp, 4($v1)
lw $t0, 0($v1)
jalr $t0
move $fp, $s2
lw $s2, 12($fp)
move $t6, $a0
la $a1, params
sw $t6, 0($a1)
move $v1, $fp
lw $v1, 0($v1)
lw $v1, 8($v1)
sw $s2,16($fp)
move $s2, $fp
lw $fp, 4($v1)
lw $t0, 0($v1)
jalr $t0
move $fp, $s2
lw $s2, 16($fp)
move $t7, $a0
move $a0, $t7
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end g
endg:
li $v0, 9
li $a0, 8
syscall
la $t0, g
sw $t0, ($v0)
sw $fp, 4($v0)
la $a0, ($v0)
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end twice
m:
li $v0, 9
li $a0,12
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
lw $t2 8($fp)
```

```
mul $t1 $t2 5
move $a0, $t1
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end m
main:
li $v0, 9
li $a0,24
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
li $v0, 9
li $a0, 8
syscall
la $t0, m
sw $t0, ($v0)
sw $fp, 4($v0)
la $t1, ($v0)
la $a1, params
sw $t1, 0($a1)
li $t1 5
sw $t1, 4($a1)
jal twice
move $t2, $a0
li $t3 5
la $a1, params
sw $t3, 0($a1)
sw $s2,8($fp)
move $s2, $fp
lw $fp, 4($t2)
lw $t0, 0($t2)
jalr $t0
move $fp, $s2
lw $s2, 8($fp)
move $t4, $a0
move $a0, $t4
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end main
```

Run: $a0 = 625

```
int doNTimes(int n,int a,function f){
```

```
while(n > 0){

a = f(a);
n = n - 1;
}

return a;
}

int m(int a){

return a*5;
}

int main(){

return doNTimes(5,5,m);
}
```

—C COMPILER

```
doNTimes:
PopArg n;
PopArg a;
PopArg f;
$t2 = n;
$t1 = $t2 > 0;
WHILE $t1:
$t1 = a;
PushParam $t1;
$t2 = LCall f;
$t3 = $t2;
a = $t3;
$t5 = n;
$t4 = $t5 - 1;
$t6 = $t4;
n = $t6;
END WHILE
Return  a;
.end doNTimes
m:
PopArg a;
$t2 = a;
$t1 = $t2 * 5;
Return  $t1;
.end m
```

```
main:
$t1 = 5;
PushParam $t1;
$t1 = 5;
PushParam $t1;
$t1 = m;
PushParam $t1;
$t2 = LCall doNTimes;
Return  $t2;
.end main



.data
params: .word 0, 0, 0
.text
.globl  main
_main:
jal main
li $v0,10
syscall
.end _main
doNTimes:
li $v0, 9
li $a0,32
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
lw $t0, 4($a1)
sw $t0,12($fp)
lw $t0, 8($a1)
sw $t0,16($fp)
While_1:
lw $t2 8($fp)
sgt $t1 $t2 0
beq $t1 0 endWhile_1
sw $fp, 20($fp)
add $fp, $fp, 20
move $v1, $fp
lw $v1, 0($v1)
lw $t1 12($v1)
la $a1, params
sw $t1, 0($a1)
```

```
move $v1, $fp
lw $v1, 0($v1)
lw $v1, 16($v1)
sw $s2,4($fp)
move $s2, $fp
lw $fp, 4($v1)
lw $t0, 0($v1)
jalr $t0
move $fp, $s2
lw $s2, 4($fp)
move $t2, $a0
move $t3 $t2
move $v1, $fp
lw $v1, 0($v1)
sw $t3 12($v1)
move $v1, $fp
lw $v1, 0($v1)
lw $t5 8($v1)
sub $t4 $t5 1
move $t6 $t4
move $v1, $fp
lw $v1, 0($v1)
sw $t6 8($v1)
lw $fp, 0($fp)
j While_1
endWhile_1:
lw $a0 12($fp)
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end doNTimes
m:
li $v0, 9
li $a0,12
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
lw $t0, 0($a1)
sw $t0,8($fp)
lw $t2 8($fp)
mul $t1 $t2 5
move $a0, $t1
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
```

```
.end m
main:
li $v0, 9
li $a0,16
syscall
sw $fp, ($v0)
sw $ra, 4($v0)
move $fp, $v0
li $t1 5
la $a1, params
sw $t1, 0($a1)
li $t1 5
sw $t1, 4($a1)
li $v0, 9
li $a0, 8
syscall
la $t0, m
sw $t0, ($v0)
sw $fp, 4($v0)
la $t1, ($v0)
sw $t1, 8($a1)
jal doNTimes
move $t2, $a0
move $a0, $t2
lw $ra 4($fp)
lw $fp 0($fp)
jr $ra
.end main

Run: $a0 = 15625
```

# References


[1] Dibyendu Das. Optimizing subroutines with optional parameters in f90 via function cloning. *SIGPLAN Not.*, 41(8):21–28, August 2006.