CM30171: Compilers

Tyler Christie

December 13, 2021

1 Overview

The aim of this project was to extend the provided lexer and parser for the language –C to provide an interpreter, and two translation phases; from parse tree to TAC (Three Address Code) and from TAC to MIPS assembler. The language itself is a C-like language (as the name implies) with the following features:

- The only datatypes are string literals, integers and functions. The inclusion of functions as a data type means that –C has functionality that standard C does not actually include; C has no ability to handle closures (unless you use non standard libaries such as FFCALL or simulate closures using structs).
- Functional arguments and results; although this comes under the above, it is worth noting that in –C, functions can take functions as arguments and return functions as results.
- Inner functions; functions can be declared within functions. This is quite an unusual trait as not many languages support lexically nested functions.
- Lexical scoping; this aligns with C/C++.
- Variable declarations must come at the top of a block. This is the same as C89 but different to C99, where variable declarations can appear anywhere (other than immediately after a label).

The compiler was approached with an agile philosophy, progressively incrementing functionality and iterating on an MVP, although there are some caveats and regrets about the development approach which will be discussed later.

2 Build and Project Structure

The project structure is kept mainly flat, and all compiles into one binary using the provided Makefile. A unit file was created for each of the main 3 components of the compiler/interpreter, along with additional helper files for each, linked with header files. In addition to the main project, the Makefile also builds and runs the test suite, to ensure all tests succeed after every addition to the code base and subsequent build. Due to time constraints, the resulting binary does not provide any command line parameter parsing for ease of use, although this could easily be added.

3 Design and Implementation

3.1 Interpreter

The code for the interpreter is the cleanest and aspect I'm most proud of in this project. It acheives the full functionality as described by the coursework specification. It accepts an **AST** of type **NODE*** as input and outputs a direct result of the computation, performing a tree walk over this AST.

3.1.1 Environment

```
typedef struct binding {
  TOKEN* name;
  VALUE* value;
  struct binding* next;
} BINDING;

typedef struct frame {
  BINDING* bindings;
  struct frame* next;
}FRAME;
```

Figure 1: The **FRAME** struct

Scope in the interpreter is handled by the **FRAME** struct. A FRAME contains all the bindings for the current scope, and links to any encapsulating scopes. A binding contains the name of a variable and its value, which can be any of string literal, integer, boolean, or closure. Thus the scope for any given function is a linked list where the first element is a frame containing its local scope, and the next element is the frame of the parent within its lexical scope, and so on. As a result, in the case of multiple declarations of a variable with the same name within a program, a function always accesses the variable which is most local to it.

3.1.2 Control Flow

First, the AST parsed by **yyparse()** (from the provided parser) enters the function **interpret**, which does an initial scan of the entire tree and declares any global variables and creates closures for any functions in the global **FRAME**. It then indexes through the bindings of the frame, and finds the closure for 'main'. If there is no main function declared, there is nothing to interpret, and so the compiler throws an error.

Once 'main' is found, it calls **call**, which as the name suggests is the function for calling functions. **call** will find the closure within the current frame

for the given name, check if it is a built-in (i.e. **print_int**, **print_string** or **read_int**) and otherwise extend the current frame (see next section) with any parameters passed to the function (these are evaluated at the point of calling, although in the case of 'main' there are none), and finally call **interpret_tree** with the extended frame. This works recursively for inner functions, and is the basic control sequence of the interpreter.

3.1.3 Extending the Frame

Extending the current frame is handled by the function **extend_frame**. This function takes a list of names (the parameters in the function definition) and a list of values (evaluated from the point of calling), and the current frame. It creates a new frame, And binds each identifier to its evaluation in order. We then set the parent of this new frame as the scope in which the calling function is defined (not the calling functions scope as this would be dynamic scoping). See Figure 2 for an illustration of how the scoping works, and Figure 3 for the corresponding code snippet.

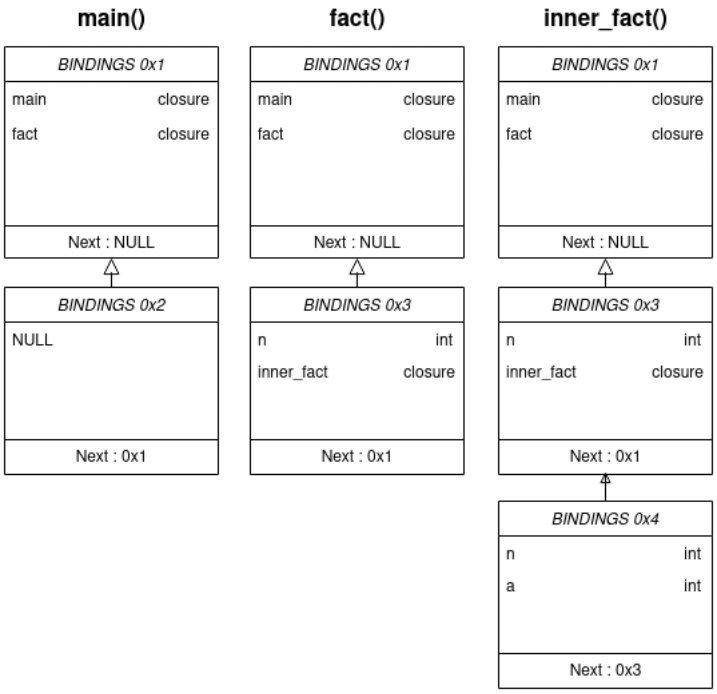

Figure 2: Frame Diagram - lower is more local to the function

```
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(10);}
```

Figure 3: A simple factorial function

3.1.4 interpret_tree

This is the main recursive function of the interpreter, which is called with the entry point 'main', and is where most of the processing happens. The entire function is effectively a switch statement, with a case for each **NODE** type:

LEAF If the node is a leaf, it contains either a constant, string literal, or identifier. For constants and string literals, a new **VALUE** struct is constructed with the value required. If it is an identifier, it will lookup the name in the list of frames described in section 3.1.1, and if it exists return it, otherwise through an 'undefined variable' error (since all variable declarations come at the top of a block).

TILDE (∼) A tilde indicates a declaration. We call the **interpret_tilde** function (simply refactored out of **interpret_tree** for code readability), which checks if the declaration is of a simple type (integer, string literal), and creates an empty binding (initialized to 0 for integers) if the declaration is of type **int x;**, else performs the assignment if the declaration is of type **int x = 1** (or any other assignment operation). If it is not a simple type (i.e. a declaration of an inner function), then we simply call **interpret_tree** again.

D A 'D' indicates a function definition. For this we simply declare a new closure within the current scope.

; A semicolon indicates a sequence of statements, so we simply recurse on the left and then the right. However we encounter an interesting problem here; that of the early return statement. Take the following code sample:

```
int main() {
    if (1==1) {
        return 1;
    }
    return 0;
}
```

(a) Code Snippet

```
D
d
  int
  F
    main
;
  if
    ==
      1
      1
    return
      1
  return
    0
```

(b) Corresponding AST

Figure 4: The Return problem

Looking at Figure 4a, it is clear that the function should return 1; however, looking at the AST, if we process a sequence of statements in the correct order, how are we to know that we should not process the next statement in the sequence, in the case of an early return?

Fortunately, the solution is quite simple. We simply have a pair of global variables: one to indicate that we are processing a sequence, and one to indicate that we have returned. So, once we reach a ';' node, we mark that we are processing a sequence. Then we process the left node. If we reach a return statement, we set our return flag. Once we come back up the tree to the initial ';' node, we check to see if we have returned early. If we have, we

just return the result on the left, and do not process the right.

Assignment (=) On this node, we know we are below the declaration block as **parse_tilde** handles this. We recurse on the left, to hit the **LEAF** case described above (to check if the variable is declared), and then cast the name on the left to a **TOKEN**, and then simply perform the assignment of whatever is on the right to the name in the current scope.

Arithmetic (+,-,*,/,%) and Boolean (¡,¿,¡=,¿=,==,!=) We know that whatever is on the left and right of this node must resolve to an integer, so we simply recurse on both sides, then return a new value with the arithmetic operation performed on the results of the recursion. The same is done for boolean operations, but we simply return a boolean **VALUE**.

RETURN Returns the result of recursing on the left child.

APPLY We first resolve the parameters of the call to a **VALUELIST** (a linked list of values), and pass the current frame, the resolved parameters, and the name of the function to **call**, which does as is described in section 3.1.2.

IF We call the **parse_if** method, which first declares a new scope for the block. Then we evaluate the condition of the branch by calling **interpret_tree**. Depending on the result, we interpret either the consequent (IF block) or the alternative(ELSE block), if there is one (and with its own scope).

There are additional helper functions in the **interpreter.c** file but what has been descibed covers the main functionality of the interpreter. The final result of interpretation is whatever is contained in the return statement in 'main'.

3.2 Intermediate Representation

TAC is generated in a similar way to the interpreter, with a walk over the AST generated by the parser using a similar recursive function. The TAC aspect of the compiler is most likely the weakest aspect of my implementation, as it

has a few problems that were not addressed in the most correct way possible due to time constraints. These will be discussed in the limitations section.

3.2.1 Environment

Scope is handled in a similar way to the interpreter, with an analogue of the interpreter **FRAME**. This FRAME is the same as is used in the MCG section and will be described in the corresponding section of the report. TAC has the addition of an **ENV** struct, which manages the allocation of labels. In retrospect this was overengineered and could've simply been handled with an integer counter. The **ENV** struct simply holds a counter, which is incremented every time a label is generated.

3.2.2 TAC Instructions

The TAC struct has 3 members; an integer 'op', to indicate the type of operation, a **TAC*** next, which indicates the next operation in the sequence, and a union whose member can be any of a STAC, PROC, LOAD, LABEL, IFTEST, GOTO, CALL or RTN.

STAC This struct is the basic type of TAC (Simple TAC - STAC), which is used for arithmetic operations. It takes 3 **TOKEN**s; two sources and a destination.

PROC This struct is for TAC instructions that indicate the start of a procedure. It takes a **TOKEN** 'name' and the arity of the procedure.

LOAD A load instruction. It takes a source and a destination. It is also used for **STORE** operations, as they take the same number of parameters.

LABEL A label for control transfers within a procedure. It takes a label name.

IFTEST An 'if' condition, indicating a conditional branch. It takes two operands to make a comparison between, an integer 'op' indicating the kind of comparison to make (these integers are the same as defined in **C.tab.h** for '<' '>', 'LE_OP', 'GE_OP' 'NE_OP' and 'EQ_OP'), and a label to jump to if the condition evaluates to false.

GOTO An unconditional branch, taking a label to jump to.

CALL Indicating a call to a function. It takes a **TOKEN** 'name' (the name of the function), the arity and a **TOKENLIST** 'args', which are the arguments to the function. These may be any of a constant, another function, a temporary (in the case of some operation e.g. '1+1' being passed as a parameter, eagerly evaluated and assigned to a temporary), or a variable name.

RTN Indicating a return from a function. It contains a union of **CALL** (in the case of a function being returned) or a **TOKEN** in the case of a variable or constant, or temporary (again, in the case of an operation, eagerly evaluated and assigned to a temporary).

There is also **ENDPROC**, which indicates the end of a procedure and takes no parameters. These are all the forms of operation that a TAC can take. These choices will be justified and evaluated in the limitations section.

3.2.3 Control Flow

The AST enters the **gen_tac** function, which initializes the environment and passes the tree to the main recursive function **gen_tac0**, which walks the tree. It then takes the generated TAC and performs another pass on it to generate the graph of basic blocks, although this was later taken out as it was not utilized, which is justified later on.

3.2.4 gen_tac0

gen_tac0 takes as input an AST, and returns a sequence of TAC instructions.

LEAF We simply **LOAD** whatever is in the leaf into a new temporary.

TILDE (∼) We call **parse_tilde**, which checks if the declaration is of type '**int x;**', '**int x = 1**' or a function declaration. If it is the first, then we perform a **LOAD** as in the case of a LEAF, and subsequently a **STORE**. We also assign the temporary to the variable in the FRAME, so that we know that temporary contains that variable. If it is the second, We call **gen_tac0** on the right to load the result of the assignment into a temporary

(if the assignment is of type **int x = 1** then the resulting TAC would be a **LOAD** instruction for 1, if the assignment is of type **int x = 1+1** the TAC would be a **STAC**, and then we assign whatever the destination temporary is to the variable in the FRAME and generate the corresponding **STORE** instruction. If it is a function declaration, we simply call **gen_tac0** again.

D We call **gen_tac0** on the left and right, and append an **ENDPROC**.

d Indicates a function definition on the right and return type on the left. We recurse on the right.

F Indicates a function definition. We create a new **PROC** instruction.

; A sequence; we recurse on the left and right. Since we are not directly computing an answer as in the interpreter, the early return problem is not relevant here.

Assignment (=) Similar to the procedure in **parse_tilde** we recurse on the right, and store whatever is the final destination of the TAC returned in the **TOKEN** on the left.

Arithmetic (+,-,*,/,%) and Boolean (¡,¿,¡=,¿=,==,!=) For arithmetic, we recurse on the left and get the final destination of the TAC returned, then recurse on the right and do the same. We then create a new **STAC** with the operands as these two destinations and the destination address as a new temporary. Booleans do not need to be handled in this main recursive function as we are not computing as in the interpreter, so they are handled by the **parse_if** function which is described later.

RETURN We call **new_return**, which has three cases for each possible return type. If the type is a LEAF (i.e. constant, variable or closure), we create a new **RTN** with the corresponding **TOKEN**. If the type is an operation (e.g. 'return 1+1'), we call **gen_tac0**, and create a new **RTN** with whatever the final destination of the generated TAC is. If the type is APPLY, we create a new **CALL**, with the same procedure as is described in the APPLY section below.

APPLY We call **new_call**, which generates a **CALL** and adds the name, arity and argument **TOKEN**s.

IF We call **parse_if**, which creates a new **IFTEST** with the opcode and two operands in the condition. If the block is of type 'IF... ELSE', We generate the code for the consequent and alternative with a resursive call on **gen_tac0**, and create a **GOTO** instruction for the end of the consequent which points to the next block below the alternative, and a **LABEL** for the alternative which the **IFTEST** points to. If the block is only an IF and no ELSE, we generate the code for the consequent again with a resursive call on **gen_tac0**, and generate a new label below the consequent which the **IFTEST** points to.

This covers all the nodes in the parse tree the TAC generator is able to parse.

3.2.5 Basic Block Generation

A basic block is a straight line code sequence that contains no control transfers[1]. In the case of generating basic blocks from TAC sequences, a basic block:

- Begins with a leader
- Continues until reaching another leader
- Ends with the instruction preceeding a leader

A leader can be:

- The first instruction in a program
- The target of a jump
- The instruction immediately following a jump

Leaders can also be instructions that follow an instruction that can throw an exception and exception handlers[2], however this is not relevant for –C. The basic blocks are generated with a pass over the generated TAC from **gen_tac0**. Unfortunately they were never put into use as due to time constraints, I could not do any machine-independent optimization.

3.2.6 Control Flow Graph (CFG)

The graph of basic blocks forms a Control Flow Graph[3], which represents all the paths of execution through a program. It is the basic structure for optimization. A basic block node in a CFG can technically have unlimited edges (for example in the case of a switch statement), but in the case of –C where the only control transfers we deal with are if. . . else blocks and function calls, the maximum number of edges is two.

```
typedef struct bb {
TOKEN* id;
TAC* leader;
struct bb *nexts[2];
}BB;
```

Figure 5: Basic Block struct

Figure 5 shows the structure of a basic block. It contains the id of the block, the straight line sequence starting at 'leader', and the array of next blocks, which may be any number from 0 to 2.

gen_bbs gen_bbs() takes as input a sequence of TAC and outputs a graph of basic blocks, represented as an adjacency list. We start at the first instruction in the TAC sequence and label this as the first block leader. we step through the sequence until reaching a **IFTEST**, **GOTO**, **LABEL** or the end of the sequence. Once we reach the end of a basic block, we label the block with an id of either the label name if the leader is a label, or an integer otherwise (which is incremented with every new block).

Then we fill in the 'nexts' array. If we have already parsed the basic block to which the current block transfers control to, we fill in its position in the 'nexts' array with a pointer to this block. If we haven't parsed it, we fill it in with a NULL pointer, and update it once this block has been parsed. We do this for every edge of a block. The result looks like this:

```
int main() {
  int x = 1;
  if(x==1){
    return 0;
  }
  else{
    return 1;
  }
}
```

(a) Code Snippet

```
BLOCK #0
PROC main 0
LOAD 1 t0
STORE t0 x
IF (x==1) L1
LINKS TO : 1 L1

BLOCK #1
RETURN 0
GOTO L2
LINKS TO : L2

BLOCK #L1
LABEL L1
RETURN 1
LINKS TO : L2

BLOCK #L2
LABEL L2
ENDPROC
```

(b) Basic Blocks generated

Figure 6: Basic Block Generation

3.3 Intermediate Code - Limitations

There are some considerable limitations in the TAC generation stage, which lead to consequences in the MCG. You will have noticed I make no mention of closures; this is because in my TAC stage I make no effort to generate closures. As a result, I am having to create these in the MCG stage whilst parsing the TAC. Another consequence of this is the inability to 'flatten' the structure of TAC; inner functions cannot be separated out from regular functions because there is no mechanism to do so without closures. As a result, I had to implement a less than ideal workaround. There is a depth counter which increments every time we recurse on a function, so that the depth at any function lexically defined in the global scope is at depth 0 and an inner

function would be at depth 1 and so on. Then any function defined at a depth of 1 or more has the TAC code **INNER_PROC**, so that the MCG parser can recognize this as a function and build a closure. Had I properly implemented the TAC stage, MCG would have been much simpler and effectively template-based, whereas now I am having to 'pick up the pieces' in MCG and build a new environment as I parse the TAC.

Furthermore, my TAC most likely could have been higher-level; instructions such as **STORE** and **LOAD** make it dangerously close to assembly, and reduce the portability and legibility of the TAC; a higher-level TAC design would better hide implementation details, and a smaller instruction set (the current system has 15 TAC instructions) would make translation to MCG simpler.

Details in regard to possible optimizations will be discussed in the Further Improvements section.

3.4 MIPS Assembly Generation

MIPS is generated with a recursive function acting on the TAC sequence generated as per section 3.2 (basic blocks are not passed in since no optimization was done).

3.4.1 Architecture

MIPS is a RISC instruction set architecture created in the 1980s[4] designed for the R2000 microprocessor. It has 32 (32-bit) general purpose registers and a few special registers:

$zero	The value 0
$v0-v1	expression evaluation and function results
$a0-a3	first four parameters for subroutine
$t0-t9	temporary variables
$s0-s7	saved values representing final computed results
$ra	return address

Figure 7: MIPS registers

Figure 7 illustrates the register usage in MIPS architecture. In this implementation, $v0 is always used for syscalls and $v1 is used for function returns. The *s* registers are not used, as they are not required in my implementation, however I will talk about how they could be used to speed up execution later on in this section.

3.4.2 Environment

An activation record (or frame as it is sometimes referred to) is simply a collection of information that can be useful for the calling and returning from procedures. It can contain[5]:

- Parameters
- Returned values
- Control link (or static link)
- Access link; a pointer to data the procedure used that is located in another frame
- Saved machine status
- Local data
- Temporaries

I will not go into detail on all of these as they are not all relevant to my implementation, but a few of these items are stored in the current activation record:

Local Data The contents of all registers in use by a procedure are saved into the activation record.

Static Link The static link of the current frame; that is, the depth in the lexical scope (although as it turns out, due to my choice of implementation this is not used). In addition to these, the activation record (AR) also contains the arity of locals.

This is the contents of the AR as is generated in the machine code, however due to a change of strategy, a lot of this information is now redundant.

The machine code generator uses the same **FRME** (an analogue of the interpreters **FRAME**) as the TAC environment, and it is used to track where variables are (i.e. in register or in memory):

```
typedef struct clsure {
  FRME* env;
  TAC* code;
  int processed;
} CLSURE;

typedef struct bnding {
  TOKEN* name;
  int type;
  union {TOKEN* loc; CLSURE* clos;};
  struct bnding* next;
} BNDING;

typedef struct frme {
  BNDING* bindings;
  int size;
  int stack_pos;
  struct frme* next;
}FRME;
```

Figure 8: MCG FRME

The **FRME** contains a size (this is the same as the size of the AR, it's just included here for convenience), the stack position (which we will come to later) and the next frame. The binding (**BNDING**) is almost identical to the binding in the interpreter, except that in place of a **VALUE** struct it is simply a union of either closure or location (register). A closure (**CLSURE**) is again almost identical to a closure in the interpreter, with the addition of a 'processed' flag, which is a workaround due to the lack of closures generated in TAC, which will be explained later in this section.

Instead of the recommended heap allocation for storing ARs, I opted for heap allocation. This was for multiple reasons; on the suggestion of a peer

that the stack would be faster - in general it is, as stack pointer relative locations can be determined at compile time causing runtime access speeds to be very fast, versus heap allocation which has to be done during run time via **sbrk**, and the other reason being for me at least the stack was simpler to understand rather than having to keep track of pointers in the heap. However, I believe now that I overestimated the speed savings as access speeds are fairly insignificant.

3.4.3 Control Transfer

Although basic blocks are not passed to the MCG for reasons previously mentioned, a basic block is a useful term to describe when we transfer control. In this section assume that when we say 'frame' we are referring to an activation record. We know that four basic things must happen when a transfer of control within execution occurs:

- Frame creation
- Frame deletion
- Frame saving
- Frame restoration

These four actions occur when we enter a basic block, exit a function, exit a basic block, and return to a function after some control transfer respectively.

Frame Creation At the start of a basic block, we must calculate the size of the frame we need to create. This is done by counting the number of locals, and adding to this a fixed number that is the basic information any frame must contain, and finally adding the arity in the case of a procedure. A typical frame would look something like this:

```
# Creating new frame
addiu $sp, $sp -12
li $t1, 12
sw $t1, 0($sp)
li $t2, 1
sw $t2, 4($sp)
li $t3, 0
sw $t3, 8($sp)
# End of creating frame
```

Figure 9: Typical MC for a frame

In Figure 9 we can see that we first decrement the stack pointer by the size of the frame, store the size of the frame in the first location, the static link in the second and finally the arity plus the number of locals. Space is allocated for these in case the frame needs to be saved but they are not actually stored on the stack until necessary.

Frame Deletion When we exit a function, we no longer require this frame, and so to save stack space we simply increment the stack pointer by the size of the frame to point to the next available location.

Frame Saving When we transfer control from within a function, we must save the contents of the registers as they are not guaranteed to be unchanged by whatever function we call. We must save all variables local to a function before we transfer control. Saving a frame will typically look like this:

```
int f() {
   return 1;
}
int main() {
  int x = 3;
  f();
  return x;
}
```

(a) Code Snippet

```
main:
# Creating new frame
addiu $sp, $sp -16
li $t1, 16
sw $t1, 0($sp)
li $t2, 1
sw $t2, 8($sp)
li $t3, 1
sw $t3, 4($sp)
# End of creating frame

li $t0,3

# Saving frame
sw $t0 12($sp)
# End of saving frame

jal f
```

(b) MCG generated - segment

Figure 10: Saving a frame

Figure 10 shows the saving of the frame of 'main' before transferring control. we store the local 'x' in its position on the stack before jumping to 'f()'.

Frame Restoration After we return from a jump in a function, we must restore the state of the registers prior to the jump. Using the same example as above:

```
main:
# Creating new frame
addiu $sp, $sp -16
li $t1, 16
sw $t1, 0($sp)
li $t2, 1
sw $t2, 8($sp)
li $t3, 1
sw $t3, 4($sp)
# End of creating frame

li $t0,3

# Saving frame
sw $t0 12($sp)
# End of saving frame

jal f
# Restoring frame
lw $t0  12($sp)
# End of restoring frame
lw $v1 t0
addiu $sp, $sp 16
jr $ra
```

Figure 11: Restoring a frame

Figure 11 Shows the restoration of the current frame after a jump. We load the local 'x' from its position on the stack back to the register it was previously in after returning from 'f()'.

These are the main operations for maintaining proper state through control transfer in a program.

3.4.4 Control flow

The control flow for the MCG is similar to the interpreter. We enter via **gen_mc** which generates some initial instructions that go at the top of every program (.global premain, .text etc). Then we enter **gen_globframe** which generates the global frame; it simply steps through the TAC, and adds any variable delcarations to the global **FRME**, and generates closures for any procedures. The global frame (AR) creation is done in a function 'premain', which does the global initialization and then jumps to main. In **gen_mc**, we step through the global **FRME** until we find 'main', and then we call the main recursive function **gen_mc0**. **gen_mc0** is again a large switch statement with a case for each TAC instruction type:

3.4.5 gen_mc0

gen_mc0 takes as input a sequence of TAC instructions and outputs a sequence of MIPS assembly instructions, mainly via template-based generation on the fly.