CM30171
Advanced Compilers

An interpreter for --C and compiler to MIPS assembler

Henry Thacker

January 11, 2010

Contents

Chapter 1

Overview

The main aim of this project was to build on top of an existing lexer and parser for the --C language to provide: an interpreter, a form of intermediate representation (three address code) and finally, a compiler to MIPS assembler. --C is a fictitious language that was developed for the purposes of this coursework. The language has several interesting features that had profound effects on the implementation of the project. A few of the pertinent points are mentioned below:

- The language allows the definition of inner functions that capture the scope of their definition environment
- Function typed variables are permitted and can be returned as function results or passed as a parameter (higher-order functions).
- No strings, character types, structures, floating point numbers or pointers. Integer and function typed variables are the only ones supported.

The coursework was completed in several separate stages and developed using C, to simplify the development. At early stages of the project, both C++ and Ruby were seriously considered as implementation languages due to the benefits of using object orientation and the sheer scale of their respective standard libraries. It was decided, however, that the amount of time-overhead involved in interfacing with the existing code would negate much of the benefit and potentially introduce bugs or unknown complexity.

Each of the different parts of the project are split into separate implementation files and folders, but are built as one large binary. Communication between different sections of the interpreter and compiler is achieved through use of shared header files. This makes it easy to use of the output of one part of the compiler (or interpreter) as the input in the next part of the process.

The implementation is fairly complete, but for a few exceptional cases within in the MIPS code generator. These are only very minor issues and indeed all of the given coursework examples (and many of my own creation) work flawlessly throughout the entire pipeline. Throughout the implementation, various text books, web resources and lecture notes were referred to (mainly for the code generation stage). Where required, in this document, the findings from each resource will be attributed.

Chapter 2

Access to Source and Binaries

2.1 Directory

/u/s/ht221/cm30171 (correct UNIX permissions have been set to allow read / execute access).
Alternatively: `http://people.bath.ac.uk/~ht221/cm30171.tar`

2.2 Host Details

Host tested on: limol.bath.ac.uk (part of the LCPU cluster)

2.3 Folder Structure

There are three subfolders in the main source directory, as follows:

debug A copy of the source code and binaries, with debug information and more verbose output for both MIPS and during interpretation

release A copy of the source code and binaries, with all debugging information stripped and the minimum verbosity output

tests A series of tests are available within the examples subdirectory. This "examples" folder is symlinked as examples from both the debug and release folders as a convenient way to execute test cases. The directory also contains a test harness that I developed, although unfortunately the BUCS machines do not have the correct Ruby gems installed to see this in action.

2.4 Instructions

The interpreter, TAC generator and compiler are all encompassed within the `mycc` binary. Invoking `mycc` without any parameters shows the usage switches (please see listing 2.1). **Please note:** some examples will not work in MIPS mode due to some shortcomings that are mentioned later on. Such examples may induce undefined behaviour / results.

To interpret the `fibonacci.c` example, one could run:

```
./mycc -i < examples/fibonacci.c
```

Or to generate the three address code representation of `twice.c`:

```
./mycc -t < examples/twice.c
```

```
henry-thackers-macbook-pro:Compilers henry$ ./mycc
--C Compiler - Input syntax

./mycc [-p|-i|-t|-m] < input_source

-p = parse mode - print out the parse tree and terminate
-i = interpret - interpret the parse tree and print the result
-t = TAC generator - print out TAC representation of the programme
-m = MIPS generator - generate MIPS machine code
```

Listing 2.1: mycc options

2.5 Source Control

The source code can be looked at online with syntax highlighting or downloaded from GitHub: `http://github.com/henrythacker/--c`.

Chapter 3

Detailed Description

3.1 Interpreter

3.1.1 Introduction

The interpreter is a complete implementation of an interpreter for the --C language. It accepts input from the parser in the form of an AST (*Abstract Syntax Tree*) and obeys the instructions contained within the nodes of the tree. We call the tree traversal "a tree walk" and we use this walk to interpret `NODE` terms on the fly. We will see this kind of recursive pattern several times throughout the whole project.

3.1.2 Initial Scan

Firstly, the interpreter scans the AST, looking for global variables and function definitions. This initial scan is done, as we have no idea where to start interpreting without the presence of some kind of entry point into the user's code. It was decided that the most sensible approach would be to require the user-supplied definition of an `int main(void)` entry-point function, in a similar vain to C. As this initial sweep of the AST takes place, any global variables or function declarations are entered into a structure known as the "environment".

Another benefit of performing this initial scan, is that all outermost functions (and global variables) will now be entered into the environment. This means that functions may call other outer-level functions regardless of the order of definition (i.e if one function calls one which is declared lower down in the file). The same, however, does not apply to inner functions / variables.

3.1.3 Environment

Quite simply, the environment houses everything that the interpreter considers to be in local scope at a given execution point. As such, after our initial scan, we can detect the presence of a valid entry-point function by examining the environment. This initial environment is known as the "global environment", as everything contained within, is visible to the whole user program. This initial scan follows exactly the same code-path as the full interpreter does, but passes a flag, in order to ensure that function bodies are never stepped into.

The environment (depicted in figure 3.1) is a hash-table of `values`. These values are references to inner functions or local variables. By following the static link between one environment and another

(please see figure 3.1), it is possible to access and refer to values residing in outer scopes. The idea of a static link was given to us in relation to their use in Activation Records (please see section 3.3) during lectures. Different information is stored for functions as opposed to variables. The `value`'s type is determined with the following constants, the main types are given here, although there are some used for other purposes that are not mentioned here for the sake of clarity.

VT_INTEGR Values of this type are integer variables, integer constants or integer temporaries.

VT_STRING Strings cannot be stored in the environment (this language does not have a string type), this value type is used to construct temporary strings that can be passed around within the interpreter / compiler. Such string values are created when identifiers are walked over, for instance.

VT_FUNCTN These values represent pointers to functions. This can be either a function variable or an actual function.

3.1.4 The Entry Point

If no entry point is found, a fatal error is raised as interpretation cannot continue without a valid entry point. If the entry point was found in our global environment, we lookup where the function definition was found in the AST and execute from this point. The `environment.c` file gives us lots of useful search functionality in order to do things such as this. The particular search we do to find the main entry point is to look for a function in the global environment called `main` with an integer return type and no formal parameters. If we get a match, we are returned a `value` structure (of type `VT_FUNCTN`) which meets our requirements within the global environment. These "function values" each hold a reference to what `NODE` in the AST is considered the "start of the function". Once we have found the corresponding "start of function" `NODE` for the `main()` function, we are able to continue the interpretation process by calling `evaluate` on this entry `NODE`.

3.1.5 Recursive function - evaluate()

The `evaluate` function is now called recursively starting from the entry point. This scan is different from the initial scan, because we now look inside function bodies and any nested functions contained therein. Every time we enter a new function or block (such as an *if statement* or *while loop*), a new environment is created with a static link to the environment of the parent (enclosing) function (or the global environment, as appropriate). As we step over the tree in this recursive function, we encounter `NODES` that are present in the AST. The following list mentions the steps that are taken for the several different (more interesting) types of node. The terms LHS and RHS refer to the *left-hand side* and *right-hand side* of an expression respectively. As the AST is a binary tree, the LHS and RHS can be thought of as the two child nodes under the current node. Please see figure 3.2 for an example AST, that will use some of the elements mentioned below.

Arithmetic Operator We look at the LHS and RHS of the expression call `evaluate` recursively on each. We do this, to try and simplify both sides (operands) into integers. From each recursive call we get back a `value` typed struct for each, which should now contain our simplified integer. If one side happened to be a function application, for instance, the recursive call has already made the necessary calls to coerce this into the appropriate integer value. We calculate the result of the arithmetic expression and return the calculated value.

= Operator This is the assignment operator. We call `evaluate` on the LHS to get the variable into which the result will be stored. For the assignment to be valid, we must have already defined the assignment variable in the environment. Otherwise, we would be using a variable that has not been declared yet. We call `evaluate` on the RHS to find out the value that we're trying to assign to the variable on the LHS. Once we have the value back, we **type check** the assignment to ensure that the assignment is valid. This prevents the user making mistakes like trying to assign an integer to a function typed variable. The type checking operation is a simple check on the `value` type on the LHS and RHS to check that they match. If the types do not match, then a fatal error describing the problem is raised. If all checks pass, then the existing LHS variable is updated with the new RHS value. This assignment will always affect the variable in the environment of its definition. The next time this LHS variable is referenced, it will have the newly assigned value.

APPLY The `APPLY` node type is encountered when a function call will occur. On the LHS of an `APPLY` node is the function name identifier and the RHS contains the list of parameters that are being passed to the function. To invoke the function, the function name is searched for within the current environment (traversing static links where necessary). If a function typed variable is provided instead of a function application, this function variable holds enough information for us to invoke the function without having to traverse multiple levels of environment. If the function is *still* not found, a fatal interpreter error is raised. If the function is found, we switch to the function's definition environment. In this environment, we store new variables with the formal parameter names, but with the specified parameter values used when calling the function. This assignment is also type checked with precisely the same method as before (see point "**= Operator**" above). Once the checks are complete, we now pass the AST node representing the start of the function body to the `evaluate` function and wait for control to return with the associated return value.

WHILE While loops are also possible in --C . The implementation for a `WHILE` statement within --C relies on the `while` construct in the host development language (C). Inside the condition for the while loop, we recursively call `evaluate` on the user specified condition. The condition is on the LHS of the `WHILE` statement. In the body of our while loop, we call `evaluate` on the RHS, which is the body of the user supplied while loop. The return value of the RHS evaluation is checked for several special cases. If we are returned the special `$BREAK` or `$CONTINUE` values then we `break`, or `continue` by using the corresponding C keywords. If any other value is returned, we immediately exit the loop and return this value.

IF and ELSE The `IF` statement is fairly similar to the `WHILE` statement in that the LHS holds the branch condition and we use C's `IF` statement to implement it. Obviously the condition is only considered once in an `IF` statement. One added complication with an if statement, is that an `IF` can be followed by an `ELSE` node. An `ELSE` node needs access to the result of the `IF` condition in order to decide whether to branch to the true or false part of the `ELSE` (please see figure 3.3 for a graphical example). To get around this problem, when the condition is evaluated in the `IF`, the result of that condition is placed into the environment as a temporary integer called `$IF`. When the `ELSE` statement executes, it checks the value of `$IF` to see whether it should branch left or right.

Functions Function definitions span multiple nodes in a tree, because of the amount of information that a function signature contains. Starting from higher up in the AST, we see nodes `D, d, F`. The way each of these nodes is handled is mentioned below.

D On the LHS is the `d` node (explained below). On the RHS is the function body, i.e. the first instruction that constitutes the start of the used-defined function. Once we have evaluated `d`, we store the returned value in the current environment along with a **reference** to the start of the function body. `evaluate` is not called on the RHS, because this is only done when the function is called. The value returned by `d` is a complete function definition that provides enough information for the declared function to be executed.

d On the LHS is the return type of the function, this can be `VOID, INT,` or `FUNCTION`. We add the return type of the function into the temporary function variable returned by the `F` node on the RHS. This is passed back up to the `D` node ready to be stored in the environment.

F On the LHS is the function name (identifier). On the RHS is the list of formal parameters. The interpreter calls `evaluate` on the RHS with a special flag to denote any variables found as parameters, rather than normal variables. This call will return a linked list of `value` structures which contains the parameter list. Once we have the function name and parameters, we build a temporary `VT_FUNCTN value` structure, which will be gradually augmented with information as it is passed up to the `D` node.

RETURN The value to return is found on the LHS of the `RETURN` statement. `evaluate` is called on the node to simplify arithmetic operations or the result of another function call. If the value to return is an identifier (i.e. a variable or a function ptr) then this is looked up in the environment and this `value` (if found) is returned instead of the identifier. Additionally, we also type check the return value by comparing the current function's return type to the value type of the evaluated return value. If the types match then the appropriate value is returned, otherwise a fatal error is raised. The same process prevents us returning `NULL`, when a return value is expected (based on the function definition having a non-`VOID` return type). **Please note:** there is no check to ensure that we actually use a return statement at all, where one is expected, only that - if we do use a return statement, it is type-checked.

3.1.6 Returning from main()

After traversing the AST, the `main` function should have a valid integer return value (unless no `RETURN` statement was present). This value is simply printed to standard out and the interpreter finishes.

3.1.7 State of the Interpreter

Within the limited time that was allocated for the project and the testing completed as part of this, the interpreter is seemingly complete. It is capable of executing all of the example scripts that were created and there are no significant known faults. There are two small minor issues that can be mentioned briefly.

- One thing that does not work in the interpreter, but parses correctly: Functions without parameters that use the keyword `void` to denote no parameters: `int my_parameterless_fn(void)`.
- Return values are type-checked, but no effort is made to detect functions that do not return a value, when they should.

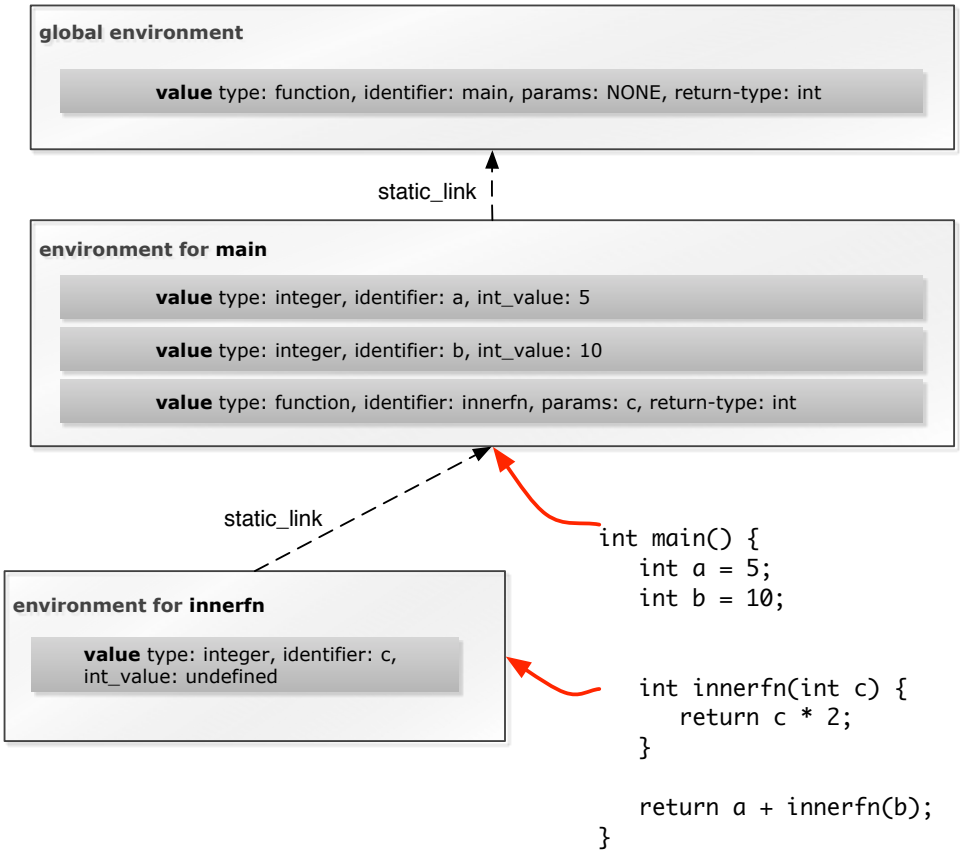

Figure 3.1: Environment Layout - Environment → Code relationship

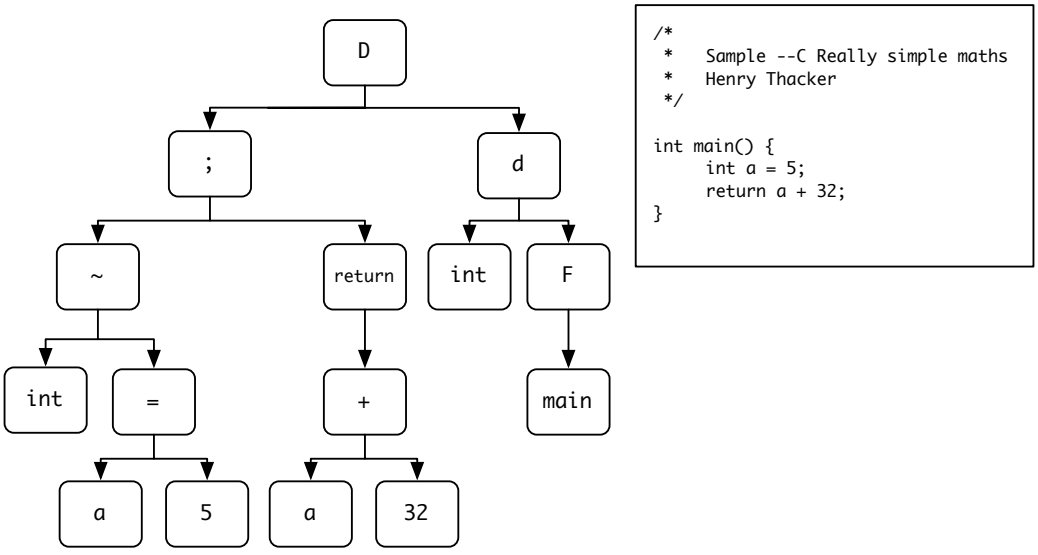

Figure 3.2: Abstract syntax tree example

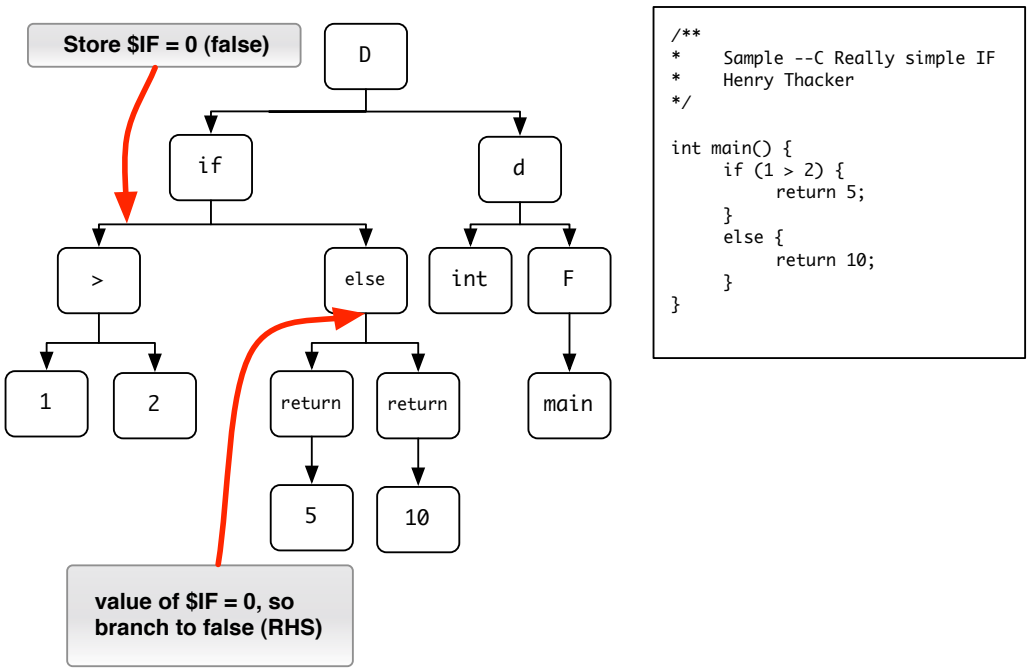

Figure 3.3: IF, ELSE example

```
typedef struct tac_quad {
    char *op; /* Operation, i.e. + for addition */
    value *operand1; /* A ptr to the first operand, could be an integer A */
    value *operand2  /* A ptr to the first operand, could be an integer B */
    value *result; /* The ptr to where the result will be stored, C */
    int type; /* TT_OP - The type is an (arithmetic) operation */
    int subtype; /* Reserved for categorisation of more granular operations */
    int level; /* Used for MIPS code generation */
    struct tac_quad *next; /* Pointer to the next tac_quad */
}tac_quad;
```

Figure 3.4: tac_quad structure

3.2 Intermediate Representation

3.2.1 Introduction

In the coursework specification we were asked to develop a three address code representation of our own design. The input to the Three Address Code generator is the AST (*Abstract Syntax tree*). The coursework specification indicates that it should be possible to read TAC (*Three Address Code*) from a flat file. This was not implemented due to time pressures and it was felt that it was more important to move onto code generation, rather than write a TAC parser. The next section 4.1 presents further background information behind the rationale of using a form of intermediate representation and the associated design choices that were made.

3.2.2 Relation to the Interpreter

The TAC is generated in a very similar way to how the interpreter works, using a large recursive function which handles all the different types of `NODE`. Instead of evaluating the result of operations, we are interested in building TAC "quadruples", internally referred to as a `tac_quad`. Each `tac_quad` is one TAC instruction and contains a **maximum** of three addresses (operand1, operand2, result) and several other pieces of metadata (please see figure 3.4). A `tac_quad` does not contain text or lexemes, each address points to a `value` within our environment. Thus, we are able to perform type checking and environment traversal, just as the interpreter does. The TAC generator generates a linked list of these `tac_quad`s that represent the user-supplied --C source.

As with the interpreter, a pre-scan is done to find all globally accessible functions. All of the global functions which are found are entered into the global environment. We have to perform this pre-scan, because when we look up function names, we must be able to point our `tac_quad`s to the correct references within the relevant environment. No notion of an entry-point is required during TAC generation, because the AST is traversed in its entirety from top to bottom, generating `tac_quad`s on the fly. Unlike the interpreter, global variables are not found in the pre-scan, as this can be done as part of the main scan mentioned below.

3.2.3 Temporaries

In Three Address Code, we are limited to a **maximum** of three addresses. Obviously, we cannot expect to complete any given operation in such a limited space, i.e. : $a = 1 + 3 + 2 + 10$ (4 operations, 5 addresses). In lectures, we were introduced to the concept of temporaries. We can assume that we have an infinite number of temporaries, but they must only appear on the LHS of an expression once. This allows us to split up complex expressions into "**simple**" subexpressions stored in temporaries, that can be combined. This "making simple" procedure is important in the implementation. Thus, our previous example becomes written as a series of simple instructions, seen in listing 3.1.

```
_t1 = 1 + 3
_t2 = 2 + 10
a = _t1 + _t2
```

Listing 3.1: Using Temporaries

In the TAC Generator implementation, temporaries are embraced as full members of the environment. Temporaries are not treated any differently from variables and are accessed using the same functions.

3.2.4 Building tac_quads

After we have completed the pre-scan to populate our global environment, a full scan is made starting from the top of the AST. The TAC instructions are formed by examining each `NODE` and evaluating the LHS and RHS until we have enough information to form a single, complete TAC instruction. The concept of "make simple" was introduced in section 3.2.3 and it refers to the process of reducing complex expressions to a temporary (i.e. the simplest form possible). Our recursive function in the TAC generator is actually called `make_simple` and at each stage, it tries to reduce a subexpression to the simplest form it can (often a temporary). The steps that are taken for a few different `NODE` types are given below.

Arithmetic Operator An integer temporary is created in preparation to hold to results of the arithmetic expression. The LHS and RHS of the operator are passed recursively to `make_simple` to coerce them into integers. If the LHS or RHS are not simple integers or variables, they may themselves be made into integer temporaries. A call is made to `make_quad_value` to build a `tac_quad` of type `TT_OP` (TAC type operation). Operands 1 and 2 are set to reference the LHS and RHS `value`s that were returned by the calls to `make_simple`. The result member of `tac_quad` is set to reference the temporary that we created earlier to hold the result. The resultant `tac_quad` is written into the linked list of TAC statements generated so far.

= Operator Both the LHS (assignment variable) and RHS (assignment value) are passed to `make_simple`. The current environment is searched to ensure that the assignment variable has already been defined, if not, an error is raised. During simplification, the RHS will be reduced to a simple constant, variable or temporary. The simplified value can then be type checked against the declaration for the assignment variable, to ensure that the assignment is valid. If so, the assignment is made in the environment. A `tac_quad` is generated of type `TT_ASSIGN`, an operation of "=", operand 1 is set to reference the assignment value and result is set to reference the assignment variable. The `tac_quad` is added to our list of existing `tac_quad`s.

APPLY Again, the LHS and RHS of the AST are passed to `make_simple`. The LHS contains the function name, while the RHS contains the list of actual parameters to be passed to the function. The environment is searched to check that a function by this name exists in local scope. If the function can not be found an error is raised. Prior to making the actual function call, a few different TAC quads are created to assist during the code generation stage. Firstly a "prepare to call" (`TT_PREPARE`) quad is created with a reference to the function (please see the next chapter for the syntax), to instruct that a function call is about to be made. Next, the actual parameters list (RHS) is passed to another recursive function which generates push statements (`TT_PUSH_PARAM`) for each parameter in the **reverse order** of their definition. This was decided after reading a technique mentioned in "Modern Compiler Design"[Grune, 2000]: "When parameters are passed on the stack, the last parameter is pushed first".

From the function definition found in the environment, we see if a return type was defined. A return type may not be available if the function is a function typed variable, as this will only be bound to another function at runtime. If a return type was found, we create an appropriately-typed temporary in which to store the result of the function application. If no return type was available, a typeless temporary is created. A `TT_FN_CALL tac_quad` is generated with a reference to the function definition stored in the operand1 member and a reference to our return value temporary in the result member. The created "result temporary" is returned as the result of the `make_simple` call, so that the function's result can be used in other expressions.

IF and ELSE On the LHS of an `IF` statement is the condition. The condition is passed to `make_simple` to simplify it to a single term. A TAC goto statement is generated that is jumped to in the event of the condition being true. The label name is automatically generated as `__ifXtrue` (where X is an integer which increments for each `IF` statement that is created). The actual label itself will be defined later on.

The node on the RHS is examined to see if it is an `ELSE` statement. If it is, we call `make_simple` on the RHS (the false portion) and append the TAC that was created. After the false branch we create a jump to the end of the `IF` statement with a similar label that will be defined later on: `__ifXend`.

In both cases (with and without an `ELSE`) the true portion of the `IF` statement is now translated. We generate a `TT_LABEL` denoting the start of the true section (as we discussed earlier: `__ifXtrue`). The true branch is now passed to `make_simple`. Finally, to complete the `IF` statement, the aforementioned end label is appended.

WHILE While statements are rewritten as `IF` statements, but with a jump back to the condition at the end of the true branch.

Functions Each function declaration generates a minimum of 5 TAC instructions, these are detailed in the relevant sections below. It may help to look at figures 3.5 and 3.2 in addition to the textual description.

D The LHS is passed to `make_simple`. This yields a basic `VT_FUNCTN` value which, by this stage, contains much of the required information regarding the function's definition. This function definition is stored in the environment. Several TAC statements are now generated, the semantics of these statements should be referred to in the next section (section 4.1).

Firstly a `TT_BEGIN_FN` is created, with operand1 set to the function reference. Immediately after this, a `TT_FN_DEF tac_quad` is created, again with a reference to the function kept in its operand1 member. `TT_FN_DEF` is used to print out the start of function label in the code generator. Next, a `TT_INIT_FRAME` quad is created, with zero entered as operand1. `TT_INIT_FRAME` gives important information about the required frame size (see section 3.3) to the code generation stage. However, at this point in the TAC generator, we do not know how large the frame needs to be, until we call `make_simple` on the function body so that the environment becomes filled. By adding a placeholder in the correct place, but with a dummy value, it is possible to return to this statement and

fill in the correct frame size, when it is eventually known.

Next, the **formal** parameters are each popped off in `TT_POP_PARAM` operations in the **order that they were defined**. Because the parameters are pushed to **function calls** in reverse order (see the description of `APPLY` above), we now assume we can read the actual parameters off some type of virtual argument stack, which are now conveniently accessible in the correct order. Each parameter is given a dummy value (0, null, etc.) depending on the parameter type and stored as a regular variable in the environment. Thus, each `TT_POP_PARAM` references the relevant uninitialised variable within the function's environment. To mark the end of parameters and the start of user-supplied code, we append a `TT_FN_BODY tac_quad`. Now `make_simple` is called on the RHS (function body).

After all TAC has been generated for the function body, we can query the size of the final environment, using a utility function `env_size`. The previously created `TT_INIT_FRAME` quad is updated with the correct value. Finally, a `TT_END_FN` instruction is created to denote the end of the function.

d, F No TAC instructions are generated here, but a `VT_FUNCTN value` is built to pass up to the `D` node. The procedure here is exactly the same as with the interpreter.

RETURN For a return statement, if the LHS is not null, it is passed to `make_simple`. If the LHS is an identifier, then this identifier is searched for in the environment. Identifiers may appear on the LHS if the function returns a function or integer-typed variable, for instance. A type check is performed on the return value to ensure it is consistent with the function definition. We now append a `TT_RETURN tac_quad` and set the operand1 member to the returned value (or null for a void function).

3.2.5 Output of tac_quads

By the time the `make_simple` call returns, a linked list of complete `tac_quad` structures has been created that constitutes the translation of the user --C program into TAC representation. As mentioned previously, these structures do not contain strings or lexemes, but contain pointers in to an environment structure. Originally the `tac_quad` **did** simply store the lexemes instead of pointers, but the code was refactored after reading several texts and consulting the lecturers. One major source of reference was the course text which summed up the correct path to take.

> "In an implementation [of Three Address Code quads], a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept." [Aho et al., 2006].

When thinking *purely* about Three Address Code, it is hard to see why pointers into the environment are required at all. In fact, this information does not appear to even be mentioned in any of the relevant material that was read. This is why the `tac_quad`s were originally designed (incorrectly) to use lexemes instead. However, it quickly became clear that because the TAC will be passed into the code-generator, it is useful to have access to scope information (provided by the environment structure already) and the type checking routines that are also available. As well as

reducing the code outlay, the environment code was rigorously tested as part of the interpreter.

Up until now, the description of the recursive function `make_simple` has mentioned `NODE` types such as `TT_OP`, `TT_LABEL` etc. This view onto TAC instructions is the way that the TAC generator internally builds and examines the TAC, additionally an optimiser or code-generator would also choose to look at the TAC quads in this fashion. However, as humans, looking at a list of opcodes and pointers is not intuitive, thus a printable TAC representation is created from the instructions with the supplied `print_tac` function (an example is given in figure 3.5). Had a TAC parser been implemented (to load TAC from a file, populating the environment), this would also have been the required input language (discussed further in 4.1). The `print_tac` function understands the syntax and semantics of the TAC language and is able to print out any given TAC instruction.

`print_tac` is another recursive function that walks over the list of generated `tac_node`s and writes out the "human readable" form of each quad. The rewrite rules for each TAC operation are given below.

TT_LABEL is written as:
`[to_string(quad->operand1)]:`, e.g: `__if1true:`

TT_FN_DEF is written as:
`_[to_string(quad->operand1)]:`, e.g: `_main:`

TT_FN_CALL is written as:
`[correct_string_rep(quad->result)] = CallFn _[correct_string_rep(quad->operand1)]:`,
e.g: `_t1 = CallFn _cplusa`

TT_INIT_FRAME is written as:
`InitFrame [correct_string_rep(quad->operand1)]`, e.g: `InitFrame 5`

TT_POP_PARAM is written as:
`PopParam [quad->operand1->identifier]`, e.g: `PopParam a`

TT_PUSH_PARAM is written as:
`PushParam [correct_string_rep(quad->operand1)]`, e.g: `PushParam 5`

TT_IF is written as:
`If [correct_string_rep(quad->operand1)] Goto [correct_string_rep(quad->result)]`,
e.g: `If _t1 Goto __if1true`

TT_ASSIGN is written as:
`[correct_string_rep(quad->result)] [correct_string_rep(quad->op)]`
`[correct_string_rep(quad->operand1)]`, e.g: `_t2 = 5`

TT_GOTO is written as:
`Goto [correct_string_rep(quad->operand1)]`, e.g: `Goto __if4end`

TT_OP is written as:
`[correct_string_rep(quad->result)] = [correct_string_rep(quad->operand1)]`
`[correct_string_rep(quad->op)] [correct_string_rep(quad->operand2)]`,
e.g: `_t4 = 5 + _t1`

TT_RETURN is written as:
`Return [correct_string_rep(quad->operand1)]`, e.g: `Return _t5`

OR `Return`, if no return value is supplied

TT_PREPARE is written as:
`PrepareToCall [param_count(quad->operand1)]`, e.g: `PrepareToCall 5`

TT_BEGIN_FN is written as:
`BeginFn [correct_string_rep(quad->operand1)]`, e.g: `BeginFn main`

TT_FN_BODY is written as:
`FnBody`

TT_END_FN is written as:
`EndFn`

Three functions mentioned above are: `to_string`, `correct_string_rep` and `param_count`. The `to_string` function takes any `value` structure of type `VT_STRING` and writes out the underlying string representation. `correct_string_rep` is a more intelligent version of `to_string` and can handle printing `value`s of any given value type. If the value is an integer, `correct_string_rep` will convert this to a string for us, it can print `VT_STRING`s directly, temporary names are written out rather than their values, function pointers are also resolved to function names and returned as strings. The final function `param_count` takes a look at a function definition (given a reference to one in an environment) and returns the size of the formal parameter list stored within the function definition.

3.2.6 Machine Independent Optimisation (MIO)

Intermediate representation is an ideal place to do various forms of optimisation. Once the translation into TAC is complete, it is easy to spot redundant instructions. Unfortunately no time was available to start implementing any form of optimisation of TAC instructions. Consideration to possible optimisation schemes will be given in the Further Improvements section. The TAC has been written with optimisation in mind and it would be fairly trivial to add at a later stage.

3.2.7 State of the TAC Generator

The TAC generator is perhaps a slightly smaller part of this project than was intended by the coursework specification. It **is** a module in its own right and can be run independently of the code generator. Unfortunately there was no time to implement a separate TAC parser to load TAC instructions directly into the code generator, although this will be discussed in the Critical Evaluation section in further detail.

Functionally, the TAC generator is not *as* complete as the interpreter, but it is still able to execute almost all of the examples as intended. One area that is lacking is the creation of new environments within `IF` and `WHILE` statements. This means that the behaviour when redefining an existing variable has not been rigorously tested and is likely to break. Another problem in the TAC generation process is that type checking of parameter values is not done. This may lead to undefined results when MIPS code is generated off a TAC translation that hasn't been correctly type-checked.

```
/**
 * Please note the indentation/spacing/comments below are NOT present in the
 * generated TAC, but are introduced here to improve legibility
 **/

/* cplus function */

BeginFn cplus
_cplus:
InitFrame 2
PopParam a
FnBody

   /* Inner function cplusa */
   BeginFn cplusa
   _cplusa:
   InitFrame 2
   PopParam b
   FnBody
      _t1 = a + b
      Return _t1
   EndFn

Return cplusa
EndFn

/* Main function */

BeginFn main
_main:
InitFrame 4
FnBody
   PrepareToCall 1
   PushParam 5
   _t2 = CallFn _cplus
   z = _t2
   PrepareToCall 1
   PushParam 9
   _t3 = CallFn _z
   Return _t3
EndFn
```

Figure 3.5: Printed TAC output

Additionally as mentioned in the section on Machine Independent Optimisation (section 3.2.6), there was simply no time to investigate various optimisations that could have been put to use. This, again, is covered in further detail in the Further Improvements section.

Some of the TAC instructions that were developed were also not used in the MIPS generation stage. This forms part of the Critical Evaluation section later in the document.

3.3 MIPS Assembler Compiler

3.3.1 Introduction

Before embarking upon this project, very little was known about the MIPS architecture and code generation, in general. These were the main areas that were researched. The code generator that was produced takes in a list of Three Address Code instructions (as described fully in section 4.1) and outputs MIPS code that is designed to be run in the SPIM emulator[1]. MIPS code can be generated for most input programs, but for a few exceptions that will be mentioned at the end of this section. The main body of the code generator is another recursive procedure which walks the list of TAC instructions, generating code on the fly.

3.3.2 The MIPS Architecture

The MIPS architecture and associated series of RISC (*Reduced Instruction Set Computer*) CPUs were pioneered by Professor John Hennessy of Stanford University in the 1980s. The RISC design strategy concentrates on providing CPU instructions that do less, but execute quickly. RISC is often referred to as load/store, as generally these are the only operations that can access main memory. All other operations require the operands to exist within "Registers". Thus, decisions surrounding register allocation form a large part of the code generation procedure. It is the efficient allocation and use of these registers that lead to a more optimised program. The registers of a typical MIPS machine are given in table 3.1.

3.3.3 Register Descriptors

While compiling for a real machine, it is necessary to have a virtual view onto the internal state of the machine as the program is compiled. This is achieved through use of register descriptors which describe the contents of the registers. In order to do this within the --C compiler, for each register in table 3.1 we create a register descriptor. In practice, we only address a few of these registers. The structure used for the register descriptors is presented in figure 3.7.

Our global view of registers is stored in an array called `regs`, thus we can go directly to `regs[reg_id]` to look into the contents. In the code, an enum (`sys_register` in `codegen_utils.h`) exists so that we can conveniently use the register names to reference registers within the C code itself (i.e. `regs[$sp]`).

3.3.4 Register Allocation

One of the restrictions of the code generator, is that it will only utilise registers $t0-$t9 for user data. `registers.c` provides several useful functions which, when combined, form the basis of the compiler's register allocation procedures.

already_in_reg - Given a `value` (and various other non-important parameters), if the value already exists in a register, we return the register it is in, otherwise a "not found" constant is returned.

first_free_reg - Return the first free register (if available). If all registers are full a "No registers free" constant is returned.

[1] `http://pages.cs.wisc.edu/~larus/spim.html`

choose_best_reg - This function is the main register allocation function. It encodes the logic for the register allocation strategy, which is detailed below. Given a `value`, it is guaranteed to return a register where the value has been loaded into.

save_t_reg - Given a register identifier, the contents will be saved to the correct place in main memory, if the value has been modified

save_t_regs - Any register with a modified value will be saved out to the correct places in main memory

The register allocation strategy is a simple one, based on material from our lectures. The process is described in steps below. The steps outlined are part of the `cg_find_variable` function in `mips.c`

- When asked for a register for variable X, we check whether the variable is already in a register using `already_in_reg`. If so, we return the identifier for this register.
- Next, we see if there is a free register into which X can be loaded. If so, we update the descriptor and return the register identifier. (This and the next step are combined within `choose_best_register`)
- If the value is still not placed then all of our registers have been exhausted. The variable that has been in the registers for the longest time will be removed (writing the modified value back to memory, if necessary).

The removal of the longest residing variable in registers is not necessarily a wise choice. The variable allocation procedure is one part of the MIPS stage that is not particularly optimal, this will be discussed in greater depth later on.

3.3.5 Runtime Support

At runtime we do not have full-access to the type of the information that was available during compilation time. We only have access to the information that we specifically placed in registers or in memory. For each function that is invoked, a structure is (generally) created called an "Activation Record". The purpose of an activation record is to have a standardised place where locals and other function specific runtime information can be stored. When the local variables are in registers and a function call happens, the modified locals can be persisted into the Activation Record and easily restored when the function returns. In most languages, the Activation Record can simply be placed on the stack and popped off when the function terminates.

As discussed in the introduction to this project, the more interesting features of --C (inner functions and function variables) have implications on the implementation. In other languages which do not support inner functions, the scope of a function can be trashed after the function exits as there is no further use for the local state. In --C and languages such as Pascal, because state is captured when functions are defined, there are cases where we must keep the Activation Records in memory, such that variables / functions can be correctly resolved.

The first implication, is that our Activation Record must contain a static link to the containing function's stack frame. This information is presented well with examples in Muchnick's book.

Having read this chapter, the Activation Record was redesigned and the final version can be seen in figure 3.8.

> "... if the source language supports statically nested scopes, the frame contains a static link to the nearest invocation of the statically containing procedure, which is the stack frame in which to look up the value of a variable declared in that procedure ..." [Muchnick, 1997]

Another implication has already been touched upon. We mentioned the fact that the nested functions and function typed variables require Activation Records to stay around longer than they would on a stack. To solve this problem, Activation Records are created in the heap instead of the stack. On entry to each function, a heap allocated activation record is created and populated. The stack is *still* used for the return address, as the push / pop nature of a stack is convenient. Parameters are also passed in the stack as there was not enough time to convert the first four parameters to use the $a0 - $a3 registers in preference to the stack. The "stack pointer" equivalent for these heap allocated activation records is stored in $s0 and the frame pointer in $fp.

A further runtime implication is covered in section 3.3.8, regarding the use and declaration of function variables.

3.3.6 Code Representation

Like TAC is stored internally in `tac_quad` structures, MIPS instructions are held in a `mips_instruction` structure. A few other structures are used for different types of instruction operands. The relevant structures can be examined in figure 3.9. Like the `tac_quad` structures, the code is built up as a linked list, before being printed by a special `print_mips` routine.

3.3.7 Code Generation Procedure

At the start of the code generation procedure, the register descriptor array `regs` is initialised. The parse tree is then passed to the TAC generator. After the TAC has been produced, the linked list of TAC is traversed once and nesting level are written into the `level` member of each `tac_quad`. Having numbered the nesting levels, `linked_sort` performs a linked-list bubble sort on the generated TAC using the level member as the sort criterion. This sort ensures that all nested functions are moved to the outermost level as there is no concept of function nesting in MIPS code.

After the TAC is ready to iterate over, a special `mips_instruction` is output from `codegen_utils.c` that writes the `.text` and `.data` segments into the output. A special constant `EOL` is defined as an end of line character and written into the `.data` segment. `EOL` will be used for printing out the program's final result. Now we loop over the various `tac_quad`s recursively generating and appending the relevant code for each type of quad. A few examples of this will be given below.

TT_BEGIN_FN A new function label `mips_instruction` is created for this function. Function labels take the user-defined name of the function, prefixed with an underscore. The global variable `current_fn` is set as a reference to this function.

TT_INIT_FRAME The frame size that was present in the `tac_quad` is the size of locals in the environment. This frame size is passed to the `activation_record_size` function which adds

on 4 (one for each of the special fields in the activation record: Previous $fp, Static Link, Dynamic Link and Frame Size). This function then multiplies this by 4 to get the number of bytes that are required to store the activation record. The Return Address ($ra) is backed up in $s7, as $ra is overwritten shortly. A function call is now made to a compiler generated MIPS function which will be added to the outputted code at the end. This function is called `mk_ar` (Make Activation Record). We pass the required frame size (in bytes) in register $a0. After the call, the activation record pointer that was returned in $v0 is moved into $s0. The activation record has been pre-populated with all of the necessary details (previous $fp, static link, etc).

TT_FN_BODY The return address that was stored in $s7 as part of the code for `TT_INIT_FRAME` is now pushed into the stack.

TT_POP_PARAM When popping a parameter, the "variable number" of the parameter is retrieved. The "variable number" is assigned to each variable when it is stored in the environment. The variable number is used to work out at what offset from the $fp, the variable will be located in. Parameters are passed in the stack so the value at the top of the stack is popped temporarily into $a0. The contents of $a0 are then written to the correct offset within the activation record using the formula: $-4 * (varnumber + 1)$$fp (the first parameter is variable_number 0).

TT_PUSH_PARAM To push a parameter, `already_in_reg` is called to check whether it is already accessible in a register. If not, we load the variable into register $a0 temporarily. The loaded parameter is then pushed into the stack and the stack pointer adjusted accordingly.

TT_ASSIGN Firstly, a register is allocated to the result variable using `get_register`. The value to store is searched for in registers, if it is found, then a simple `move` operation between registers takes place. Otherwise, a the value is moved into a register using the `choose_best_reg` function. At this point, if the variable is located outside of the local scope, the static links are followed (please see section 3.3.9 for the method). Finally, the result variable is marked as having a modified value.

TT_IF Due to the way conditions are handled in the TAC instruction set, the handling of the `IF` statement during the code generation stage is very simple, as the condition will have already been reduced to an integer. The condition value is loaded into a register using the `get_register` function. As a result, a `bne` (*Branch not equal*) statement is sufficient to perform the branch.

TT_RETURN With return value If the operand1 member of the `tac_quad` is set then a value is being returned. If the value is **NOT** a function typed variable, then the value is simply moved to $v0 as a return value. Because function typed variables capture scope, they are a more complex case and are covered in section 3.3.8. After the return value has been stored in $v0, the steps are the same as those below.

Without a return value If the operand1 member of the `tac_quad` is `NULL` then no return value is returned. Before returning, any $t registers that have been used are saved. The `clear_regs` function is used to clear out our register descriptors. The previously saved return address is popped from the stack and restored into $ra. The previous frame pointer is also restored so that the procedure that we return to can find its variables using the expected offsets. Finally, we restore the previously saved static link ($s0) and jump back to the address in $ra.

TT_FN_CALL When a function is called, any modified $t* registers are saved into the current activation record. The register descriptors are then cleared, so that the callee can allocate these registers as it wishes. A register is allocated to hold the return value of the called function. If we are calling a normal function (i.e. not following a function typed variable), the correct static link is deduced by the rules in section 3.3.9. Before calling the function, we have to set several registers that the callee is expecting. The static link is stored in $v0 and $a1 is set to the dynamic link (current value of $s0). The function is then called. After the function returns, its return value (if applicable) is moved from $v0 into the allocated result register.

If a function is being called through the use of a function typed variable, this procedure is slightly more complicated (please read section 3.3.8 for some background information). The function descriptor address has to be loaded to have enough information in order to execute the function. The function descriptor address is stored as the value for a function typed variable, when the variable is first declared. In order to lookup the descriptor, we load the value for the function typed variable into the first available register. The stored value is the address of the function descriptor. Once we have the function descriptor, it is trivial to find the entry point and static link. The entry point is stored in the first word from the base descriptor address and the static link is stored in the second. The function entry point is loaded into another register and the static link is loaded into $v0. As with normal function calls, the dynamic link is saved into $a1. The function is then called using the address that was found within the function descriptor. After the function returns, its return value (if applicable) is moved from $v0 into the allocated result register.

After the code generation has taken place, a few extra pieces of code are appended:

mk_ar The definition of `mk_ar` is appended into the linked list of code. This code is required so that activation records can be created upon function entry.

rfunc A global variable `has_used_fn_variable` is set to 1 whenever a function typed variable is used. If this is flag is 1 after code generation is complete, then the definition for the `rfunc` function is added to the end of the assembly code. This code is necessary to create function descriptors.

main In order to execute an assembly program, like in C, we must define a `main` function. This `main` function is provided by our compiler. The `main` function stores zero in both $a1 and $v0. These registers are zeroed out because there is no static or dynamic link at the start of execution because no activation records have been created yet. The user defined `main` function, now residing in `_main`, is called. Once the return value is accessible, three MIPS `syscall`s occur. The first of these copies the return value into $a0 and performs `syscall` type 1. This operation causes the return value to be printed out. The second `syscall` prints the `EOL` character that we defined right at the start of the assembly program. Finally, the last `syscall` instructs the processor that the program has terminated.

After all the code has been generated, some basic optimisations on the code are performed - these are detailed in section 3.3.10

Finally, the code is printed to standard out using `print_mips`. The `print_mips` function simply walks down the linked list and prints the operation name, for example `add`. Then each of the operands for the instruction are examined. If the operand is set to a sensible value then it is printed. If the operand is an offset operand, then it is printed: `offset_amount($register)`, register names can simply be printed as are constants. Function labels are printed as labels but prefixed with an underscore.

Most operations that occur in the code generation stage are given internal comments that describe what the operation does. If a debug version of the compiler is run then these comments are outputted by `print_mips`. These comments are not present in the release binaries. The behaviour can be changed by modifying the value of `DEBUG_ON` in `environment.h`.

3.3.8 Function Typed Variables

In order to facilitate the use of function typed variables, at runtime we must be able look up not just the start address of the function (as we can for labels), but also the correct static link. By the time we come to invoke the function, we have lost any information about the context in which the function was originally defined. To remedy this, each time a function typed variable is **created**, a call is made to another helper routine that is appended into the generated assembly: `rfunc`. `rfunc` takes a reference to the function entry point in register $v0 and the correct static link in $v1. 8 bytes of memory is allocated in the heap to store this pair within `rfunc`. The address to this function descriptor is returned in $v0. As a small optimisation, the `rfunc` function is not appended to our assembly file if no function typed variables are used within the user's program.

When the function is invoked, the function descriptor is consulted in order to deduce which static link should be used.

3.3.9 Traversing Static Links

The logic behind traversing static links was something else that had to be researched extensively. Whenever a variable is asked for in a function, if it is stored outside our current environment then we must follow the static link (stored at 0($fp)) to find it. We may need to traverse several levels of static link before we are guaranteed to find it. The `cg_find_variable` calculates the difference in nesting between our current environment and the variable's environment. With this knowledge it is possible to traverse the correct number of static links such that the variable can be accessed from its usual offset.

The logic behind this is fairly straightforward, however it is slightly more difficult to work out what static link to pass in the first place. As the functions are no longer nested in the program's text, the static link between functions must be passed almost as another parameter to the function when it is invoked. Luckily, there are well-known rules that make this easy to ascertain. Muchnick details these rules as follows:

1. If the procedure being called is nested directly within its caller, its static link points to its caller's frame
2. If the procedure is at the same level of nesting as its caller, then its static link is a copy of its caller's static link
3. If the procedure being called is n levels higher than the caller in the nesting structure then its static link can be determined by following n static links back from the caller's static link and copying the static link found there

Figure 3.6: What static link should be passed?[Muchnick, 1997]

These rules are encoded in the `cg_load_static_link` function within `mips.c`. The nesting of a function is determined by taking the nesting of its environment. The environmental nesting level is defined as how many static links we have to follow to get back to the global environment (whose static link is 0). Once the nesting difference between the caller and callee have been calculated, the rules above can be applied.

3.3.10 Optimisation

During the code generation a few **basic** optimisations are carried out. The first very simple optimisation is wherever a 0 zero constant is used, instead of writing it into a register, we substitute the provided $zero register instead. Additionally, to reduce the size of the assembly program, `rfunc` is not added to the program unless function typed variables are used.

In `optimisation.c`, three additional basic optimisations are included as follows:

remove_after_return This optimisation scans over all the generated `mips_instruction`s looking for a return (`jr`) or unconditional jump statement (`j`). All unreachable lines of code after this, up until the next: label, function label or comment are removed. To remove the code, the pointer within the linked list are altered to bypass the redundant instructions.

remove_redundant_move Due to an inefficiency in the code generation phase, the MIPS instruction `move $X, $v0` is often followed by `move $v0, $X`. The second `move` is completely redundant. This optimisation removes the second `move` instruction by altering the pointers to circumvent it.

remove_redundant_sp_move When values are pushed / popped from the stack, the stack pointer is adjusted accordingly. If a function call occurs right at the top of the function body, after all of the parameters have just been popped, quite often a `add $sp, $sp, 4` will be followed directly with a `sub $sp, $sp 4`. This optimisation removes both of these instructions where they occur in pairs.

These optimisations, while simple, do work well to reduce the size of the generated assembly program. Additionally, these examples are there to show that optimisation has been considered and is possible. With a bit more thought, these ideas could be extended to complicated optimisation strategies. The code that is produced is still fairly sub-optimal in places, but it is functional.

3.3.11 State of the MIPS Assembler Compiler

The code generator works with almost all of the examples, but there are a few things that it cannot do. Global variable declarations and operations do not work. If two outer functions contain inner functions with the same name, when these are expanded inline to MIPS assembler, the function labels will collide. Apart from these minor issues, the MIPS compiler is fairly functional. In the Critical Evaluation section a further critique of the produced code will be given, as there is much scope for improvement in this area.

```
typedef struct register_contents {
    value *contents; /* Value stored in the register */
    int accesses; /* How many times the value has been referenced */
    int assignment_id; /* What order this assignment was made */
    int modified; /* Have the contents been modified since load? */
}register_contents;
```

Figure 3.7: Register descriptor record

Table 3.1: MIPS Registers[Dandamudi, 2005]

Register	Internal Name	Description
$zero	$0	This register conveniently always contains zero
$at	$1	Assembler temporary, reserved for use by the assembler
$v0-$v1	$2-$3	Used for function return values
$a0-$a3	$4-$7	Reserved for the first 4 arguments to a function
$t0-$t7	$8-$15	Temporaries (Caller save)
$s0-$s7	$16-$23	Temporaries (Callee save)
$t8-$t9	$24-$25	Temporaries (Caller save)
$k0-$k1	$26-$27	Reserved for operating system
$gp	$28	Global Pointer
$sp	$29	Stack Pointer
$fp	$30	Frame Pointer
$ra	$31	Return Address

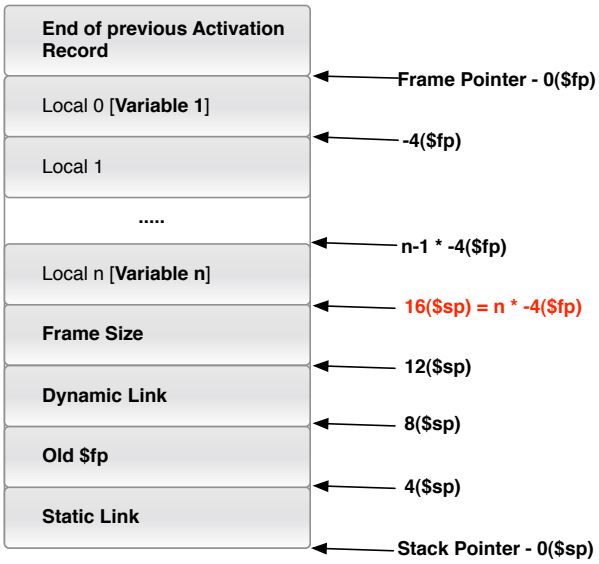

Figure 3.8: Activation Record - Design

```
enum sys_register {
    $zero = 0,
    $at = 1,
    $v0 = 2, $v1 = 3,
    $a0 = 4, $a1 = 5, $a2 = 6, $a3 = 7,
    $t0 = 8, $t1 = 9, $t2 = 10, $t3 = 11, $t4 = 12, $t5 = 13, $t6 = 14,$t7 = 15,
    $s0 = 16, $s1 = 17, $s2 = 18, $s3 = 19, $s4 = 20, $s5 = 21, $s6 = 22, $s7 = 23,
    $t8 = 24, $t9 = 25,
    $k0 = 26, $k1 = 27,
    $gp = 28,
    $sp = 29,
    $fp = 30,
    $ra = 31
}sys_register;

typedef struct register_offset {
    enum sys_register reg;
    int offset;
}register_offset;

typedef union operand {
    enum sys_register reg;
    struct register_offset* reg_offset;
    int constant;
    char *label;
}operand;

typedef struct mips_instruction {
    char *operation;
    int operand1_type;
    int operand2_type;
    int operand3_type;
    union operand *operand1;
    union operand *operand2;
    union operand *operand3;
    char *comment;
    int indent_count;
    struct mips_instruction *next;
}mips_instruction;
```

Figure 3.9: MIPS Instruction structure

Chapter 4

Three Address Code

4.1 Introduction

Throughout the design of the three address code, it was important to keep the purpose of intermediate representations in mind:

> The *intermediate code generation module* translates language-specific constructs in the AST into more general constructs ... One criterion for the level of the intermediate code is that it should be reasonably straightforward to generate machine code from it for various machines ... [Grune, 2000].

In order to genericise the instructions as much as possible, the instruction set provides no special instructions for `WHILE` loops, inner functions or variable type specific operations. The instruction set is described in table 4.1. The **overall** syntax of the TAC instructions will also be discussed briefly afterwards.

As mentioned in section 3.2.1, it is not currently possible to input TAC directly to the code generator. Table 4.1 only describes the syntax and semantics of the TAC instructions as generated from the AST provided by the parser. If TAC input *was* entered directly by the user, the scheme would need to be consistent with the details provided below.

4.2 Instruction Set

Table 4.1: Instruction Set

Instruction	**TAC Type**	**Description**
`Label:`	TT_LABEL	Labels can be made up of any alphanumeric characters. The AST → TAC translation utilises double underscore to denote jumps within `IF` statements, e.g. `__if1end`

Table 4.1: (Continued from previous page)

Instruction	TAC Type	Description
`_FunctionLabel:`	TT_FN_DEF	Function labels are prefixed with a single underscore
`[result = ] CallFn <FnName>`	TT_FN_CALL	`<FnName>` is the function that should be invoked. The `[result = ]` portion is optional if no return value is expected
`InitFrame <FrameSize>`	TT_INIT_FRAME	`<FrameSize>` is the **number** of locals in the local environment of this function. This instruction is important as it gives the code generator information about how much space to allocate for local variables
`PopParam <Var>`	TT_POP_PARAM	Each formal parameter must be popped as part of the function definition
`PushParam <Value>`	TT_PUSH_PARAM	Each actual parameter must be pushed as part of a function call. Parameters must be pushed in **reverse** order. If a function has parameters `int a, int b`, the caller must push the value for `b` before the value for parameter `a`
`If <Cond.> Goto <TrueJmp>`	TT_IF	`<Cond.>` is the branch condition, this should be an integer, temporary or variable `<TrueJmp>` is a label that will be jumped to when `<Cond.>` is true. Enforcing a simple IF statement of this format (with zero denoting FALSE) means we do not have to have a multitude of different comparison functions. The comparison complexity is moved into the `TT_OP` operation
`<Result> = <Value>`	TT_ASSIGN	`<Result>` is the name of a variable into which the value of `<Value>` will be stored
`Goto <Label>`	TT_GOTO	`<Label>` is a label that will immediately be jumped to
`<Result> = <Op1> <Op> <Op2>`	TT_OP	`<Result>` is where the value of of the operation will be stored (temporary or variable), `<Op1>` is the first operand, `<Op2>` is the second operand and `<Op>` is an operator: `+, -, *, /, %, ==, >, <, <=, >=, !=`
`Return <Value>`	TT_RETURN	`<Value>` is the return value. A value may be omitted if the function has a `VOID` return type
`PrepareToCall <ParamCount>`	TT_PREPARE	`<ParamCount>` is the number of parameters that are about to be pushed as part of a function application

Table 4.1: (Continued from previous page)

Instruction	TAC Type	Description
`BeginFn <FnName>`	TT_BEGIN_FN	`<FnName>` is the name of a function that is about to be defined
`FnBody`	TT_FN_BODY	This statement is used after all of the formal parameters have been popped and denotes the start of the function body
`EndFn`	TT_END_FN	This instruction denotes the end of a function

4.3 Overall Syntax

It is not enough to only consider the semantics and syntax of each individual instruction, but also to examine how these instructions can be combined. Each syntax rule is given in terms of a pseudo grammar-like language and a textual description. Although the grammar definitions may seem unwieldy on first glance, they help formally define my textual explanation and could easily be adapted to actually implement the TAC parser that is currently missing. The grammar rules presented here are **not** used as part of the project and simply serve to supplement the textual description.

4.3.1 Function Declaration

A function declaration consists of a `BeginFn FnName` instruction, followed by the function label prefixed with an underscore: `_FnName`. An `InitFrame` instruction is then required (with the number of the function locals): `InitFrame 10`. All parameters must now be "popped" in separate `PopParam` instructions. If the function takes no parameters then no `PopParam` instructions are required. To indicate the start of the function body, the `FnBody` instruction must follow. At this point, the TAC for the function body is appended. At the end of the body, a `EndFn` instruction must be used to indicate the end of the function.

```
EOL :=
    : "\n"
    | "\r" "\n"
    | EOL "\r"
    | EOL "\r" "\n"

Pop_Param_List :=
    :  TT_POP_PARAM EOL
    |  Pop_Param_List TT_POP_PARAM EOL

Function_Declaration :=
    : TT_BEGIN_FN EOL TT_FN_DEF EOL TT_INIT_FRAME EOL Pop_Param_List
      TT_FN_BODY EOL TAC_Instructions TT_END_FN
    | TT_BEGIN_FN EOL TT_FN_DEF EOL TT_INIT_FRAME EOL TT_FN_BODY EOL
      TAC_Instructions TT_END_FN
```

4.3.2 Function Application

A valid function application must first start with a `PrepareToCall [ArgNumber]` instruction. Each of the actual parameter values are now "pushed", in **reverse** order. Argument pushing is achieved through use of the `PushParam` instruction. The number of required push operations is indicated by the value of `[ArgNumber]` in the `PrepareToCall` instruction. If `[ArgNumber]` = 0, then no `PushParam` instructions are required. Finally, `CallFn [FnName]` is used to invoke the function with the correct parameters.

```
EOL :=
    : "\n"
    | "\r" "\n"
    | EOL "\r"
    | EOL "\r" "\n"

Push_Param_List :=
    :  TT_PUSH_PARAM EOL
    |  Push_Param_List TT_PUSH_PARAM EOL

Function_Application :=
    : TT_PREPARE EOL Push_Param_List TT_FN_CALL EOL
    | TT_PREPARE EOL TT_FN_CALL EOL
```

4.3.3 TAC Syntax

With the two definitions above, it is possible to write rules which describe a valid TAC input. The grammar below defines a goal `Valid_TAC` which we are trying to aim for when writing TAC Instructions that are to be considered valid. To meet the criteria for `Valid_TAC`, there must be **at least** one function declaration. On either side of a function declaration, variable assignments and operations are permitted (i.e. the use of global variables). To examine what TAC is permissible within a function declaration, we must backtrack slightly. With the **Function Declaration** rule above (see section 4.3.1), we make a reference to `TAC_Instructions` which is defined below. `TAC_Instruction` is any instruction that is permitted within the body of a function. This rule allows us to nest function declarations, use `IF` statements, jumps, label declarations and call other functions.

Although this language has now been completely specified, there are still some constructs which may not be backwards-compatible with the --C parser (please see figure 4.1). An equivalent version that will parse is presented in figure 4.2. Such statements (`TT_OP`) **are** permissible in the TAC specification to allow for assigning a calculation to a global variable during variable initialisation (see the TAC equivalent in figure 4.3). This is an interesting aside, but not a problem as such, because we have met the target that all valid --C programs can be converted to a TAC representation with the instruction set that has been presented.

```
EOL :=
    : "\n"
    | "\r" "\n"
    | EOL "\r"
    | EOL "\r" "\n"
```

```
Variable_Declaration :=
    : TT_ASSIGN EOL
    | Variable_Declaration TT_ASSIGN EOL

Variable_Operations :=
    : TT_OP EOL
    | Variable_Operations TT_OP EOL

Variable_Use :=
    : Variable_Declaration
    | Variable_Operations
    | Variable_Use Variable_Declaration
    | Variable_Use Variable_Operations

TAC_Instructions :=
    : Variable_Use
    | Function_Declaration
    | Function_Application
    | TT_IF EOL
    | TT_GOTO EOL
    | TT_RETURN EOL
    | TT_LABEL EOL
    | TAC_Instructions Variable_Use
    | TAC_Instructions Function_Declaration
    | TAC_Instructions Function_Application
    | TAC_Instructions TT_IF EOL
    | TAC_Instructions TT_GOTO EOL
    | TAC_Instructions TT_RETURN EOL
    | TAC_Instructions TT_LABEL EOL

Valid_TAC :=
    : Function_Declaration
    | Variable_Use Function_Declaration
    | Variable_Use Function_Declaration Variable_Use
    | Valid_TAC Function_Declaration
    | Valid_TAC Variable_Use

Goal := Valid_TAC
```

```
int a = 15;
a = a - 12; /* << --C parser will not allow this outside of a function body */
            /* TAC syntax allows such statements outside + inside functions */
int main() {
return a;
}
```

Figure 4.1: Modifying global variables - --C

```
int a = 15 - 12;

int main() {
return a;
}
```

Figure 4.2: Fixed: Modifying global variables - --C

```
_t1 = 15 - 12 /* TT_OPs are allowed on global vars */
a = _t1
BeginFn main
_main:
InitFrame 0
FnBody
Return a
EndFn
```

Figure 4.3: Modifying global variables - TAC

Chapter 5

Testing

5.1 Summary

The interpreter and TAC generator were able to cope with all of the test cases that were given (except the expected failure cases). The code generator was not able to work with the `innertest.c` and `voidfn.c` examples. The `voidfn.c` example does not work because it uses global variables. The `innertest.c` example does not work because the inner functions both have the same name.

The `typecheck1.c` and `typecheck2.c` files are both designed to fail, as they both contain intentional typing problems. Unfortunately the TAC generator is not able to type check parameters, so invalid TAC and MIPS code is generated for the `typecheck1.c` example.

5.2 Test Cases

5.2.1 Test 1 - bigmath.c

Input File

```
/*Result: 2034*/


/*
*       Sample --C Test large number of variable assignments
*       Henry Thacker
*/

int main() {
        int a = 5, b = 10;
        int c = 929, d = 32;
        int g;
        int e = 90;
        int f = 72;
        int h = 442;
        g = 12;
        return a + b + c + d + e + f + g + (2 * h);
```

```
}
```

Purpose of Test

This test was created to see what happens when registers need to be spilled into the activation record.

Expected Result

The expected output for this test = Result: 2034

Interpreter Result

The result from the interpreter is: 2034

Generated TAC

```
BeginFn main
_main:
InitFrame 16
FnBody
a = 5
b = 10
c = 929
d = 32
e = 90
f = 72
h = 442
g = 12
_t7 = a + b
_t6 = _t7 + c
_t5 = _t6 + d
_t4 = _t5 + e
_t3 = _t4 + f
_t2 = _t3 + g
_t8 = 2 * h
_t1 = _t2 + _t8
Return _t1
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:55 2010

.data
        EOL:    .asciiz "\n"
.text
```

```
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 80      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 5
        lw $t1, -8($fp) # Load local variable
        li $t1, 10
        lw $t2, -12($fp)        # Load local variable
        li $t2, 929
        lw $t3, -16($fp)        # Load local variable
        li $t3, 32
        lw $t4, -24($fp)        # Load local variable
        li $t4, 90
        lw $t5, -28($fp)        # Load local variable
        li $t5, 72
        lw $t6, -32($fp)        # Load local variable
        li $t6, 442
        lw $t7, -20($fp)        # Load local variable
        li $t7, 12
        add $t8, $t0, $t1
        add $t9, $t8, $t2
        sw $t3, -16($fp)        # Write out used local variable
        sw $t4, -24($fp)        # Write out used local variable
        lw $t4, -16($fp)        # Load local variable
        add $t3, $t9, $t4
        sw $t5, -28($fp)        # Write out used local variable
        sw $t6, -32($fp)        # Write out used local variable
        lw $t6, -24($fp)        # Load local variable
        add $t5, $t3, $t6
        sw $t7, -20($fp)        # Write out used local variable
        sw $t0, -4($fp) # Write out used local variable
        lw $t0, -28($fp)        # Load local variable
        add $t7, $t5, $t0
        sw $t1, -8($fp) # Write out used local variable
        sw $t8, -60($fp)        # Write out used local variable
        lw $t8, -20($fp)        # Load local variable
        add $t1, $t7, $t8
        sw $t2, -12($fp)        # Write out used local variable
        sw $t9, -56($fp)        # Write out used local variable
        li $t9, 2
        lw $t4, -32($fp)        # Load local variable
        mult $t9, $t4
        mflo $t2
```

```
        sw $t3, -52($fp)        # Write out used local variable
        add $t3, $t1, $t2
        move $v0, $t3   # Assign values
        sw $t1, -40($fp)        # Write out used local variable
        sw $t2, -64($fp)        # Write out used local variable
        sw $t3, -36($fp)        # Write out used local variable
        sw $t5, -48($fp)        # Write out used local variable
        sw $t7, -44($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
```

```
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
2034
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The MIPS code writes variables out to memory quite a bit. The MIPS code just follows the simple rule where any spilled register should be written out to memory if it contains a modified value. Without any kind of liveness checking, it is difficult for the compiler to take more intelligent decisions. The code produces the correct answer.

5.2.2 Test 2 - cplusa.c

Input File

```
/*Result: 14*/

/*
*       Sample --C static +5 function Programme
*       Henry Thacker
*/

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

int main() {
        function z = cplus(5);
        return z(9);
}
```

Purpose of Test

This test was given to us as part of the coursework specification. It is designed to test scope capture, inner functions and function typed variables.

Expected Result

The expected output for this test = Result: 14

Interpreter Result

The result from the interpreter is: 14

Generated TAC

```
BeginFn cplus
_cplus:
InitFrame 2
PopParam a
FnBody
BeginFn cplusa
_cplusa:
InitFrame 2
PopParam b
FnBody
_t1 = a + b
```

```
Return _t1
EndFn
Return cplusa
EndFn
BeginFn main
_main:
InitFrame 4
FnBody
PrepareToCall 1
PushParam 5
_t2 = CallFn _cplus
z = _t2
PrepareToCall 1
PushParam 9
_t3 = CallFn _z
Return _t3
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:55 2010

.data
        EOL:    .asciiz "\n"
.text

_cplus:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        la $v0, _cplusa # Store address of function
        move $v1, $s0   # Store static link to call with
        jal rfunc       # Register fn variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 32      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
```

```
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $a0, 5
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _cplus
        move $t0, $v0
        lw $t1, -4($fp) # Load local variable
        move $t1, $t0   # Assign values
        li $a0, 9
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        sw $t0, -8($fp) # Write out used local variable
        sw $t1, -4($fp) # Write out used local variable
        lw $t0, -4($fp) # Load local variable
        lw $t2, ($t0)   # Get Fn address
        lw $v0, 4($t0)  # Get static link
        move $a1, $s0   # Pass dynamic link
        jalr $t2
        move $t1, $v0
        sw $t1, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_cplusa:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, ($s0)   # Move up a static link
        lw $t2, 12($t1) # Load framesize for static link
        add $t2, $t2, $t1       # Seek to $fp [end of AR]
        lw $t1, -4($t2) # a
        lw $t2, -4($fp) # Load local variable
        add $t0, $t1, $t2
        move $v0, $t0   # Assign values
        sw $t0, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
```

```
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
# Make a new function variable entry
# Precondition: $v0 contains function address, $v1 contains static link
# Returns: address of allocated fn entry descriptor in $v0
rfunc:
        move $s1, $v0   # Backup fn address in $s1
        li $a0, 8       # Space required for descriptor
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        sw $s1, ($v0)   # Store fn address
        sw $v1, 4($v0)  # Store static link
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
```

```
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
14
```

Test Analysis

Interpreter The interpreter gave the expected result

TAC The TAC seems fairly optimised except that the return value for a function call could be written straight into the answer variable rather than into a temporary first.

MIPS Except for the overhead introduced by using function variables (i.e the calls to and definition for `rfunc`), the code doesn't have many trivial optimisations. The code produces the correct answer.

5.2.3 Test 3 - factorial.c

Input File

```
/*Result: 720*/

/*
*       Sample --C Factorial Programme
*       Henry Thacker
*/

int fact(int n) {
        int inner_fact(int n, int a) {
                /* Mention how IF stmts need to be in curlies */
                if (n==0) return a;
                return inner_fact(n-1, a * n);
        }
        return inner_fact(n, 1);
}

/* Main entry point */
int main() {
        return fact(6);
}
```

Purpose of Test

This test was given to us as part of the coursework specification. It is mainly used to check recursion.

Expected Result

The expected output for this test = Result: 720

Interpreter Result

The result from the interpreter is: 720

Generated TAC

```
BeginFn fact
_fact:
InitFrame 3
PopParam n
FnBody
BeginFn inner_fact
_inner_fact:
InitFrame 7
PopParam n
PopParam a
```

```
FnBody
_t1 = n == 0
If _t1 Goto __if1true
Goto __if1end
__if1true:
Return a
__if1end:
_t2 = n - 1
_t3 = a * n
PrepareToCall 2
PushParam _t3
PushParam _t2
_t4 = CallFn _inner_fact
Return _t4
EndFn
PrepareToCall 2
PushParam 1
PushParam n
_t5 = CallFn _inner_fact
Return _t5
EndFn
BeginFn main
_main:
InitFrame 1
FnBody
PrepareToCall 1
PushParam 6
_t6 = CallFn _fact
Return _t6
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:55 2010

.data
        EOL:    .asciiz "\n"
.text

_fact:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
```

```
        li $a0, 1
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        lw $t0, -4($fp) # Load local variable
        move $a0, $t0   # Assign values
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        move $v0, $s0   # Set this current activation record as the
           static link
        move $a1, $s0   # Pass dynamic link
        jal _inner_fact
        move $t0, $v0
        sw $t0, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 20      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $a0, 6
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _fact
        move $t0, $v0
        sw $t0, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_inner_fact:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 44      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        add $sp, $sp, 4 # Move stack pointer
```

```
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -8($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, -4($fp) # Load local variable
        seq $t0, $t1, $zero     # $c = $a == $b
        bne $t0, $zero, __if1true
        j __if1end
__if1true:
        lw $t2, -8($fp) # Load local variable
        move $v0, $t2   # Assign values
        sw $t0, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if1end:
        lw $t1, -4($fp) # Load local variable
        sub $t0, $t1, 1
        lw $t3, -8($fp) # Load local variable
        mult $t3, $t1
        mflo $t2
        sub $sp, $sp, 4 # Move stack pointer
        sw $t2, ($sp)   # Write param into stack
        sub $sp, $sp, 4 # Move stack pointer
        sw $t0, ($sp)   # Write param into stack
        sw $t0, -16($fp)        # Write out used local variable
        sw $t2, -20($fp)        # Write out used local variable
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _inner_fact
        move $t0, $v0
        sw $t0, -24($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
```

```
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
720
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer.

5.2.4 Test 4 - fibonacci.c

Input File

```
/*Result: 34*/

/*
*       Sample ––C Fibonacci Programme
*       Henry Thacker
*/

int fib(int n) {
        if (n < 2)
                return n;
        else
                return fib(n - 1) + fib(n - 2);
}

/* Main entry point */
int main() {
        return fib(9);
}
```

Purpose of Test

This is another test which is designed to check that recursion works correctly.

Expected Result

The expected output for this test = Result: 34

Interpreter Result

The result from the interpreter is: 34

Generated TAC

```
BeginFn fib
_fib:
InitFrame 7
PopParam n
FnBody
_t1 = n < 2
If _t1 Goto __if1true
_t3 = n - 1
PrepareToCall 1
PushParam _t3
_t4 = CallFn _fib
_t5 = n - 2
```

```
PrepareToCall 1
PushParam _t5
_t6 = CallFn _fib
_t2 = _t4 + _t6
Return _t2
Goto __if1end
__if1true:
Return n
__if1end:
EndFn
BeginFn main
_main:
InitFrame 1
FnBody
PrepareToCall 1
PushParam 9
_t7 = CallFn _fib
Return _t7
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:55 2010

.data
        EOL:    .asciiz "\n"
.text

_fib:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 44      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, -4($fp) # Load local variable
        slti $t0, $t1, 2        # $c = $a < b
        bne $t0, $zero, __if1true
        sub $t2, $t1, 1
        sub $sp, $sp, 4 # Move stack pointer
        sw $t2, ($sp)   # Write param into stack
        sw $t0, -8($fp) # Write out used local variable
        sw $t2, -16($fp)        # Write out used local variable
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
```

```
        jal _fib
        move $t0, $v0
        lw $t2, -4($fp) # Load local variable
        sub $t1, $t2, 2
        sub $sp, $sp, 4 # Move stack pointer
        sw $t1, ($sp)   # Write param into stack
        sw $t0, -20($fp)        # Write out used local variable
        sw $t1, -24($fp)        # Write out used local variable
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _fib
        move $t0, $v0
        lw $t2, -20($fp)        # Load local variable
        add $t1, $t2, $t0
        move $v0, $t1   # Assign values
        sw $t0, -28($fp)        # Write out used local variable
        sw $t1, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
        j __if1end
__if1true:
        lw $t0, -4($fp) # Load local variable
        move $v0, $t0   # Assign values
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if1end:
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        move $v0, $zero # Null return value
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 20      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $a0, 9
```

```
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _fib
        move $t0, $v0
        sw $t0, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
```

```
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
34
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer.

5.2.5 Test 5 - gt.c

Input File

```
/*Result: 1*/

/*
*       Sample --C Greater Than test
*       Henry Thacker
*/

int main() {
        if (2 > 1) {
                return 1;
        }
        else {
                return 0;
        }
        /* Should never get here */
        return 0;
}
```

Purpose of Test

This is to check that the comparison operators work that are used during the code generation phase

Expected Result

The expected output for this test = Result: 1

Interpreter Result

The result from the interpreter is: 1

Generated TAC

```
BeginFn main
_main:
InitFrame 1
FnBody
_t1 = 2 > 1
If _t1 Goto __if1true
Return 0
Goto __if1end
__if1true:
Return 1
__if1end:
Return 0
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:55 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 20      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $t1, 2
        li $t2, 1
        sgt $t0, $t1, $t2       # $c = $a > $b
        bne $t0, $zero, __if1true
        li $v0, 0
        sw $t0, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
        j __if1end
__if1true:
        li $v0, 1
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if1end:
        li $v0, 0
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
```

```
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
1
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS In this example there is a lot of overhead introduced by creating the complex activation record. For simple "leaf functions" like `_main`, the activation record could be avoided altogether. Despite the complexity, the code produces the correct answer.

5.2.6 Test 6 - gte.c

Input File

```
/*Result: 1*/

/*
*       Sample --C Greater Than / Equal test
*       Henry Thacker
*/

int main() {
        if (5 >= 5) {
                return 1;
        }
        else {
                return 0;
        }
        /* Should never get here */
        return 0;
}
```

Purpose of Test

This is to check that the comparison operators work that are used during the code generation phase

Expected Result

The expected output for this test = Result: 1

Interpreter Result

The result from the interpreter is: 1

Generated TAC

```
BeginFn main
_main:
InitFrame 1
FnBody
_t1 = 5 >= 5
If _t1 Goto __if1true
Return 0
Goto __if1end
__if1true:
Return 1
__if1end:
Return 0
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:55 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 20      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $t1, 5
        li $t2, 5
        sge $t0, $t1, $t2       # $c = $a >= $b
        bne $t0, $zero, __if1true
        li $v0, 0
        sw $t0, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
        j __if1end
__if1true:
        li $v0, 1
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if1end:
        li $v0, 0
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
```

```
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
1
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The constant 5 is loaded into two registers. This is wasteful as the comparison could be done more simply with: `sge $t0, $t1, $t1`, freeing up $t2. The code produces the correct answer.

5.2.7 Test 7 - gte2.c

Input File

```
/*Result: 1*/

/*
*       Sample --C Greater Than / Equal test 2
*       Henry Thacker
*/

int main() {
        if (6 >= 5) {
                return 1;
        }
        else {
                return 0;
        }
        /* Should never get here */
        return 0;
}
```

Purpose of Test

This is to check that the comparison operators work that are used during the code generation phase

Expected Result

The expected output for this test = Result: 1

Interpreter Result

The result from the interpreter is: 1

Generated TAC

```
BeginFn main
_main:
InitFrame 1
FnBody
_t1 = 6 >= 5
If _t1 Goto __if1true
Return 0
Goto __if1end
__if1true:
Return 1
__if1end:
Return 0
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:55 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 20      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $t1, 6
        li $t2, 5
        sge $t0, $t1, $t2       # $c = $a >= $b
        bne $t0, $zero, __if1true
        li $v0, 0
        sw $t0, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
        j __if1end
__if1true:
        li $v0, 1
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if1end:
        li $v0, 0
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
```

```
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
1
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer.

5.2.8 Test 8 - iftest.c

Input File

```
/*Result: 9*/

/*
*       Sample ––C Nested IF Programme
*       Henry Thacker
*/

int main() {
        int a = 15;
        if (1) {
                if (0) {
                        a = 5;
                }
                else {
                        a = 12;
                }
        }
        else {
                a = 9;
        }
        if (1) {
                if (0) {
                        a = a - 1;
                }
                else {
                        a = a - 3;
                }
        }
        else {
                a = a - 3;
        }
        return a;
}
```

Purpose of Test

This code is designed to test that branching (including nested IF / ELSE statements) operations work as intended.

Expected Result

The expected output for this test = Result: 9

Interpreter Result

The result from the interpreter is: 9

Generated TAC

```
BeginFn main
_main:
InitFrame 4
FnBody
a = 15
If 1 Goto __if1true
a = 9
Goto __if1end
__if1true:
If 0 Goto __if2true
a = 12
Goto __if2end
__if2true:
a = 5
__if2end:
__if1end:
If 1 Goto __if3true
_t1 = a - 3
a = _t1
Goto __if3end
__if3true:
If 0 Goto __if4true
_t2 = a - 3
a = _t2
Goto __if4end
__if4true:
_t3 = a - 1
a = _t3
__if4end:
__if3end:
Return a
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:55 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 32      # Store the frame size required for this AR
        jal mk_ar
```

```
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 15
        li $t1, 1
        bne $t1, $zero, __if1true
        li $t0, 9
        j __if1end
__if1true:
        bne $zero, $zero, __if2true
        li $t0, 12
        j __if2end
__if2true:
        li $t0, 5
__if2end:
__if1end:
        li $t2, 1
        bne $t2, $zero, __if3true
        sub $t3, $t0, 3
        move $t0, $t3   # Assign values
        j __if3end
__if3true:
        bne $zero, $zero, __if4true
        sub $t4, $t0, 3
        move $t0, $t4   # Assign values
        j __if4end
__if4true:
        sub $t5, $t0, 1
        move $t0, $t5   # Assign values
__if4end:
__if3end:
        move $v0, $t0   # Assign values
        sw $t0, -4($fp) # Write out used local variable
        sw $t3, -8($fp) # Write out used local variable
        sw $t4, -12($fp)        # Write out used local variable
        sw $t5, -16($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
```

```
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
9
```

Test Analysis

Interpreter The interpreter gave the expected result

TAC There are cases where a couple of different labels are defined next to each other (i.e. `__if2end` and `__if1end`). The reference to one of these labels could be removed (and all references to it refactored) to save extra lines in the output. This change would benefit the code generation stage too.

MIPS The code produces the correct answer.

5.2.9 Test 9 - indirection.c

Input File

```
/*Result: 2*/

/*
*       Sample --C Function indirection test Programme
*       Henry Thacker
*/

int add1(int i) {
        return i + 1;
}

function test() {
        return add1;
}

int main() {
        function f = test();
        return f(1);
}
```

Purpose of Test

This test is designed to ensure that function-typed return values can be resolved and called properly.

Expected Result

The expected output for this test = Result: 2

Interpreter Result

The result from the interpreter is: 2

Generated TAC

```
BeginFn add1
_add1:
InitFrame 2
PopParam i
FnBody
_t1 = i + 1
Return _t1
EndFn
BeginFn test
_test:
InitFrame 0
```

```
FnBody
Return add1
EndFn
BeginFn main
_main:
InitFrame 4
FnBody
PrepareToCall 0
_t2 = CallFn _test
f = _t2
PrepareToCall 1
PushParam 1
_t3 = CallFn _f
Return _t3
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:55 2010

.data
        EOL:    .asciiz "\n"
.text

_add1:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, -4($fp) # Load local variable
        addi $t0, $t1, 1
        move $v0, $t0   # Assign values
        sw $t0, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_test:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 16      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
```

```
        sw $s7, ($sp)   # Save return address in stack
        la $v0, _add1   # Store address of function
        move $v1, $s0   # Store static link to call with
        jal rfunc       # Register fn variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 32      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _test
        move $t0, $v0
        lw $t1, -4($fp) # Load local variable
        move $t1, $t0   # Assign values
        li $a0, 1
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        sw $t0, -8($fp) # Write out used local variable
        sw $t1, -4($fp) # Write out used local variable
        lw $t0, -4($fp) # Load local variable
        lw $t2, ($t0)   # Get Fn address
        lw $v0, 4($t0)  # Get static link
        move $a1, $s0   # Pass dynamic link
        jalr $t2
        move $t1, $v0
        sw $t1, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
```

```
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
# Make a new function variable entry
# Precondition: $v0 contains function address, $v1 contains static link
# Returns: address of allocated fn entry descriptor in $v0
rfunc:
        move $s1, $v0   # Backup fn address in $s1
        li $a0, 8       # Space required for descriptor
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        sw $s1, ($v0)   # Store fn address
        sw $v1, 4($v0)  # Store static link
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
2
```

Test Analysis

Interpreter The interpreter gave the expected result

TAC The TAC seems fairly optimised except that the return value for a function call could be

written straight into the answer variable rather than into a temporary first.

MIPS The code produces the correct answer.

5.2.10 Test 10 - innertest.c

Input File

```
/*Result: 33*/

/*
*       Sample --C Inner Test Programme
*       Henry Thacker
*/

/* Main entry point */

int fn1() {
        int inner1() {
                return 11;
        }
        return inner1();
}

int fn2() {
        int inner1() {
                return 22;
        }
        return inner1();
}

int main() {
        return fn1() + fn2();
}
```

Purpose of Test

This tests are designed to check calling inner functions works. It is expected that the code generation phase will fail because the two inner functions have the same name.

Expected Result

The expected output for this test = Result: 33

Interpreter Result

The result from the interpreter is: 33

Generated TAC

```
BeginFn fn1
_fn1:
InitFrame 2
```

```
FnBody
BeginFn inner1
_inner1:
InitFrame 0
FnBody
Return 11
EndFn
PrepareToCall 0
_t1 = CallFn _inner1
Return _t1
EndFn
BeginFn fn2
_fn2:
InitFrame 2
FnBody
BeginFn inner1
_inner1:
InitFrame 0
FnBody
Return 22
EndFn
PrepareToCall 0
_t2 = CallFn _inner1
Return _t2
EndFn
BeginFn main
_main:
InitFrame 3
FnBody
PrepareToCall 0
_t4 = CallFn _fn1
PrepareToCall 0
_t5 = CallFn _fn2
_t3 = _t4 + _t5
Return _t3
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_fn1:
        move $s7, $ra   # Store Return address in $s7
```

```
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        move $v0, $s0   # Set this current activation record as the
           static link
        move $a1, $s0   # Pass dynamic link
        jal _inner1
        move $t0, $v0
        sw $t0, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_fn2:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        move $v0, $s0   # Set this current activation record as the
           static link
        move $a1, $s0   # Pass dynamic link
        jal _inner1
        move $t0, $v0
        sw $t0, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _fn1
        move $t0, $v0
        sw $t0, -8($fp) # Write out used local variable
```

```
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _fn2
        move $t0, $v0
        lw $t2, -8($fp) # Load local variable
        add $t1, $t2, $t0
        move $v0, $t1   # Assign values
        sw $t0, -12($fp)        # Write out used local variable
        sw $t1, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_inner1:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 16      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $v0, 11
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_inner1:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 16      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $v0, 22
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
```

```
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s

spim: (parser) Label is defined for the second time on line 80 of file temp
  _inner1:
        ^
spim: (parser) Label is defined for the second time on line 81 of file /Users/henry/test.s
  _inner1:
        ^


22
```

Test Analysis

Interpreter The interpreter gave the expected result

TAC The TAC seems fairly optimised except that the return value for a function call could be written straight into the answer variable rather than into a temporary first.

MIPS The code is invalid because there are function labels with duplicate names (see section 3.3.11). As a result, this test will not run properly. SPIM does try to execute the code anyway and gives 22 as the result. This value comes from the wrong `inner1` function being selected, so the result is calculated as $11 + 11 = 22$ instead of $11 + 22 = 33$.

5.2.11 Test 11 - keytest.c

Input File

```
/*Result: 83*/

/*
*       Sample --C Test scope traversal
*       Henry Thacker
*/

int main() {
        int a = 5;
        int b = 9;
        int my_func() {
                int c = 20;
                int test() {
                        int d = 11;
                        int test2() {
                                d = 18;
                                return c + d + b + a + 13;
                        }
                        return test2() + d;
                }
                return test();
        }
        return my_func();
}
```

Purpose of Test

This test is devised to test scope traversal within multiple nested functions.

Expected Result

The expected output for this test = Result: 83

Interpreter Result

The result from the interpreter is: 83

Generated TAC

```
BeginFn main
_main:
InitFrame 4
FnBody
a = 5
b = 9
```

```
BeginFn my_func
_my_func:
InitFrame 3
FnBody
c = 20
BeginFn test
_test:
InitFrame 4
FnBody
d = 11
BeginFn test2
_test2:
InitFrame 4
FnBody
d = 18
_t4 = c + d
_t3 = _t4 + b
_t2 = _t3 + a
_t1 = _t2 + 13
Return _t1
EndFn
PrepareToCall 0
_t6 = CallFn _test2
_t5 = _t6 + d
Return _t5
EndFn
PrepareToCall 0
_t7 = CallFn _test
Return _t7
EndFn
PrepareToCall 0
_t8 = CallFn _my_func
Return _t8
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 32      # Store the frame size required for this AR
        jal mk_ar
```

```
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 5
        lw $t1, -8($fp) # Load local variable
        li $t1, 9
        sw $t0, -4($fp) # Write out used local variable
        sw $t1, -8($fp) # Write out used local variable
        move $v0, $s0   # Set this current activation record as the
           static link
        move $a1, $s0   # Pass dynamic link
        jal _my_func
        move $t0, $v0
        sw $t0, -16($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_my_func:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 20
        sw $t0, -4($fp) # Write out used local variable
        move $v0, $s0   # Set this current activation record as the
           static link
        move $a1, $s0   # Pass dynamic link
        jal _test
        move $t0, $v0
        sw $t0, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_test:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 32      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
```

```
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 11
        sw $t0, -4($fp) # Write out used local variable
        move $v0, $s0   # Set this current activation record as the
           static link
        move $a1, $s0   # Pass dynamic link
        jal _test2
        move $t0, $v0
        lw $t2, -4($fp) # Load local variable
        add $t1, $t0, $t2
        move $v0, $t1   # Assign values
        sw $t0, -16($fp)        # Write out used local variable
        sw $t1, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_test2:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 32      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, ($s0)   # Move up a static link
        lw $t1, 12($t0) # Load framesize for static link
        add $t1, $t1, $t0       # Seek to $fp [end of AR]
        lw $t0, -4($t1) # d
        li $t0, 18
        lw $t2, ($s0)   # Move up a static link
        lw $t2, ($t2)   # Move up a static link
        lw $t3, 12($t2) # Load framesize for static link
        add $t3, $t3, $t2       # Seek to $fp [end of AR]
        lw $t2, -4($t3) # c
        add $t1, $t2, $t0
        lw $t4, ($s0)   # Move up a static link
        lw $t4, ($t4)   # Move up a static link
        lw $t4, ($t4)   # Move up a static link
        lw $t5, 12($t4) # Load framesize for static link
        add $t5, $t5, $t4       # Seek to $fp [end of AR]
        lw $t4, -8($t5) # b
        add $t3, $t1, $t4
        lw $t6, ($s0)   # Move up a static link
        lw $t6, ($t6)   # Move up a static link
        lw $t6, ($t6)   # Move up a static link
```

```
        lw $t7, 12($t6) # Load framesize for static link
        add $t7, $t7, $t6       # Seek to $fp [end of AR]
        lw $t6, -4($t7) # a
        add $t5, $t3, $t6
        addi $t7, $t5, 13
        move $v0, $t7   # Assign values
        lw $t8, ($s0)   # Move up a static link
        lw $s6, 12($t8) # Load framesize for static link
        add $s6, $s6, $t8       # Seek to $fp [end of AR]
        sw $t0, -4($s6) # Save distant modified variable
        sw $t1, -16($fp)        # Write out used local variable
        sw $t3, -12($fp)        # Write out used local variable
        sw $t5, -8($fp) # Write out used local variable
        sw $t7, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
```

```
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
83
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer, but there is a lot of function calling overhead and static link traversal.

5.2.12 Test 12 - lt.c

Input File

```
/*Result: 1*/

/*
*       Sample --C Less Than test
*       Henry Thacker
*/

int main() {
        if (1 < 2) {
                return 1;
        }
        else {
                return 0;
        }
        /* Should never get here */
        return 0;
}
```

Purpose of Test

This is to check that the comparison operators work that are used during the code generation phase

Expected Result

The expected output for this test = Result: 1

Interpreter Result

The result from the interpreter is: 1

Generated TAC

```
BeginFn main
_main:
InitFrame 1
FnBody
_t1 = 1 < 2
If _t1 Goto __if1true
Return 0
Goto __if1end
__if1true:
Return 1
__if1end:
Return 0
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 20      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $t1, 1
        slti $t0, $t1, 2        # $c = $a < b
        bne $t0, $zero, __if1true
        li $v0, 0
        sw $t0, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
        j __if1end
__if1true:
        li $v0, 1
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if1end:
        li $v0, 0
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
```

```
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
1
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer.

5.2.13 Test 13 - lte.c

Input File

```
/*Result: 1*/

/*
*       Sample --C Less Than / Equal test
*       Henry Thacker
*/

int main() {
        if (1 <= 1) {
                return 1;
        }
        else {
                return 0;
        }
        /* Should never get here */
        return 0;
}
```

Purpose of Test

This is to check that the comparison operators work that are used during the code generation phase

Expected Result

The expected output for this test = Result: 1

Interpreter Result

The result from the interpreter is: 1

Generated TAC

```
BeginFn main
_main:
InitFrame 1
FnBody
_t1 = 1 <= 1
If _t1 Goto __if1true
Return 0
Goto __if1end
__if1true:
Return 1
__if1end:
Return 0
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 20      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $t1, 1
        li $t2, 1
        sle $t0, $t1, $t2       # $c = $a <= $b
        bne $t0, $zero, __if1true
        li $v0, 0
        sw $t0, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
        j __if1end
__if1true:
        li $v0, 1
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if1end:
        li $v0, 0
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
```

```
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
1
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer.

5.2.14 Test 14 - lte2.c

Input File

```
/*Result: 1*/

/*
*       Sample --C Less Than / Equal test 2
*       Henry Thacker
*/

int main() {
        if (2 <= 3) {
                return 1;
        }
        else {
                return 0;
        }
        /* Should never get here */
        return 0;
}
```

Purpose of Test

This is to check that the comparison operators work that are used during the code generation phase

Expected Result

The expected output for this test = Result: 1

Interpreter Result

The result from the interpreter is: 1

Generated TAC

```
BeginFn main
_main:
InitFrame 1
FnBody
_t1 = 2 <= 3
If _t1 Goto __if1true
Return 0
Goto __if1end
__if1true:
Return 1
__if1end:
Return 0
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 20      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $t1, 2
        li $t2, 3
        sle $t0, $t1, $t2       # $c = $a <= $b
        bne $t0, $zero, __if1true
        li $v0, 0
        sw $t0, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
        j __if1end
__if1true:
        li $v0, 1
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if1end:
        li $v0, 0
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
```

```
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
1
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer.

5.2.15 Test 15 - maxmin.c

Input File

```
/*Result: 5*/

/*
*       Sample --C max/min integer Programme
*       Henry Thacker
*/

int max(int a, int b) {
        if (a >= b) {
                return a;
        }
        return b;
}

int min(int a, int b) {
        if (a <= b) {
                return a;
        }
        return b;
}


int main() {
        int a = 5, b = 9;
        return min(a, b);
}
```

Purpose of Test

This test was written to ensure that a return statement actually returns control to the calling function.

Expected Result

The expected output for this test = Result: 5

Interpreter Result

The result from the interpreter is: 5

Generated TAC

```
BeginFn max
_max:
InitFrame 3
```

```
PopParam a
PopParam b
FnBody
_t1 = a >= b
If _t1 Goto __if1true
Goto __if1end
__if1true:
Return a
__if1end:
Return b
EndFn
BeginFn min
_min:
InitFrame 3
PopParam a
PopParam b
FnBody
_t2 = a <= b
If _t2 Goto __if2true
Goto __if2end
__if2true:
Return a
__if2end:
Return b
EndFn
BeginFn main
_main:
InitFrame 3
FnBody
a = 5
b = 9
PrepareToCall 2
PushParam b
PushParam a
_t3 = CallFn _min
Return _t3
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_max:
```

```
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        add $sp, $sp, 4 # Move stack pointer
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -8($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, -4($fp) # Load local variable
        lw $t2, -8($fp) # Load local variable
        sge $t0, $t1, $t2       # $c = $a >= $b
        bne $t0, $zero, __if1true
        j __if1end
__if1true:
        move $v0, $t1   # Assign values
        sw $t0, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if1end:
        lw $t0, -8($fp) # Load local variable
        move $v0, $t0   # Assign values
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_min:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        add $sp, $sp, 4 # Move stack pointer
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -8($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, -4($fp) # Load local variable
        lw $t2, -8($fp) # Load local variable
        sle $t0, $t1, $t2       # $c = $a <= $b
        bne $t0, $zero, __if2true
        j __if2end
```

```
__if2true:
        move $v0, $t1   # Assign values
        sw $t0, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if2end:
        lw $t0, -8($fp) # Load local variable
        move $v0, $t0   # Assign values
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 5
        lw $t1, -8($fp) # Load local variable
        li $t1, 9
        sub $sp, $sp, 4 # Move stack pointer
        sw $t1, ($sp)   # Write param into stack
        sub $sp, $sp, 4 # Move stack pointer
        sw $t0, ($sp)   # Write param into stack
        sw $t0, -4($fp) # Write out used local variable
        sw $t1, -8($fp) # Write out used local variable
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _min
        move $t0, $v0
        sw $t0, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
```

```
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
5
```

Test Analysis

Interpreter The interpreter gave the expected result

TAC One of the functions `max` is never called. It should be removed from the generated TAC as an optimisation.

MIPS The code produces the correct answer.

5.2.16 Test 16 - reallysimple.c

Input File

```
/*Result: 24*/

/*
*       Sample --C Test simple function call
*       Henry Thacker
*/

int test(int a) {
        return a + 1;
}

int main() {
        int a = 23;
        return test(a);
}
```

Purpose of Test

This was a very early test to check that parameter passing and function calls work as expected.

Expected Result

The expected output for this test = Result: 24

Interpreter Result

The result from the interpreter is: 24

Generated TAC

```
BeginFn test
_test:
InitFrame 2
PopParam a
FnBody
_t1 = a + 1
Return _t1
EndFn
BeginFn main
_main:
InitFrame 2
FnBody
a = 23
PrepareToCall 1
PushParam a
```

```
_t2 = CallFn _test
Return _t2
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_test:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, -4($fp) # Load local variable
        addi $t0, $t1, 1
        move $v0, $t0   # Assign values
        sw $t0, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 23
        sub $sp, $sp, 4 # Move stack pointer
        sw $t0, ($sp)   # Write param into stack
        sw $t0, -4($fp) # Write out used local variable
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _test
        move $t0, $v0
        sw $t0, -8($fp) # Write out used local variable
```

```
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
24
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer.

5.2.17 Test 17 - simple.c

Input File

```
/*Result: 37*/

/*
*       Sample --C Really simple maths
*       Henry Thacker
*/

int main() {
        int a = 5;
        return a + 32;
}
```

Purpose of Test

This was another very early test to check that operators function as expected.

Expected Result

The expected output for this test = Result: 37

Interpreter Result

The result from the interpreter is: 37

Generated TAC

```
BeginFn main
_main:
InitFrame 2
FnBody
a = 5
_t1 = a + 32
Return _t1
EndFn
```

Generated MIPS Assembly

```

# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
```

```
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 5
        addi $t1, $t0, 32
        move $v0, $t1   # Assign values
        sw $t0, -4($fp) # Write out used local variable
        sw $t1, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
37
```

Test Analysis

Interpreter The interpreter gave the expected result

TAC This whole example could be simplified at compile time to generate the TAC: `_t1 = 37`.

MIPS The code produces the correct answer.

5.2.18 Test 18 - simpleinner.c

Input File

```
/*Result: 11*/

/*
*       Sample --C Test simple inner function
*       Henry Thacker
*/

int main() {
        int times2(int a) {
                return a * 2;
        }
        int me(int b) {
                int addOneTimes2(int a) {
                        return times2(a) + 1;
                }
                return addOneTimes2(b);
        }
        return me(5);
}
```

Purpose of Test

This is another scope traversal test.

Expected Result

The expected output for this test = Result: 11

Interpreter Result

The result from the interpreter is: 11

Generated TAC

```
BeginFn main
_main:
InitFrame 3
FnBody
BeginFn times2
_times2:
InitFrame 2
PopParam a
FnBody
_t1 = a * 2
Return _t1
```

```
EndFn
BeginFn me
_me:
InitFrame 3
PopParam b
FnBody
BeginFn addOneTimes2
_addOneTimes2:
InitFrame 3
PopParam a
FnBody
PrepareToCall 1
PushParam a
_t3 = CallFn _times2
_t2 = _t3 + 1
Return _t2
EndFn
PrepareToCall 1
PushParam b
_t4 = CallFn _addOneTimes2
Return _t4
EndFn
PrepareToCall 1
PushParam 5
_t5 = CallFn _me
Return _t5
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        li $a0, 5
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
```

```
        move $v0, $s0   # Set this current activation record as the
           static link
        move $a1, $s0   # Pass dynamic link
        jal _me
        move $t0, $v0
        sw $t0, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_times2:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, -4($fp) # Load local variable
        li $t2, 2
        mult $t1, $t2
        mflo $t0
        move $v0, $t0   # Assign values
        sw $t0, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_me:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        move $a0, $t0   # Assign values
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        move $v0, $s0   # Set this current activation record as the
           static link
        move $a1, $s0   # Pass dynamic link
        jal _addOneTimes2
        move $t0, $v0
```

```
        sw $t0, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_addOneTimes2:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        move $a0, $t0   # Assign values
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        lw $v0, ($s0)   # Move up one static link
        lw $v0, ($v0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _times2
        move $t0, $v0
        addi $t1, $t0, 1
        move $v0, $t1   # Assign values
        sw $t0, -12($fp)        # Write out used local variable
        sw $t1, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
```

```
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
11
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer.

5.2.19 Test 19 - twice.c

Input File

```
/*Result: 7*/

/*
*       Sample --C test double application Programme
*       Henry Thacker
*/

function twice(function f) {
        int g(int x) { return f(f(x)); }
        return g;
}

int add(int a) {
        return a + 1;
}

int main() {
        function twicea = twice(add);
        return twicea(5);
}
```

Purpose of Test

This example was given to us in the coursework specification. It is designed to check that double application, function parameters, scope capture and function return types work.

Expected Result

The expected output for this test = Result: 7

Interpreter Result

The result from the interpreter is: 7

Generated TAC

```
BeginFn twice
_twice:
InitFrame 2
PopParam f
FnBody
BeginFn g
_g:
InitFrame 3
PopParam x
```

```
FnBody
PrepareToCall 1
PushParam x
_t1 = CallFn _f
PrepareToCall 1
PushParam _t1
_t2 = CallFn _f
Return _t2
EndFn
Return g
EndFn
BeginFn add
_add:
InitFrame 2
PopParam a
FnBody
_t3 = a + 1
Return _t3
EndFn
BeginFn main
_main:
InitFrame 4
FnBody
PrepareToCall 1
PushParam add
_t4 = CallFn _twice
twicea = _t4
PrepareToCall 1
PushParam 5
_t5 = CallFn _twicea
Return _t5
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_twice:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
```

```
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        la $v0, _g      # Store address of function
        move $v1, $s0   # Store static link to call with
        jal rfunc       # Register fn variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_add:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, -4($fp) # Load local variable
        addi $t0, $t1, 1
        move $v0, $t0   # Assign values
        sw $t0, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 32      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        la $v0, _add    # Store address of function
        move $v1, $s0   # Store static link to call with
        jal rfunc       # Register fn variable
        move $t0, $v0   # Return fn descriptor address
        sub $sp, $sp, 4 # Move stack pointer
        sw $t0, ($sp)   # Write param into stack
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _twice
        move $t0, $v0
        lw $t1, -4($fp) # Load local variable
        move $t1, $t0   # Assign values
```

```
        li $a0, 5
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        sw $t0, -8($fp) # Write out used local variable
        sw $t1, -4($fp) # Write out used local variable
        lw $t0, -4($fp) # Load local variable
        lw $t2, ($t0)   # Get Fn address
        lw $v0, 4($t0)  # Get static link
        move $a1, $s0   # Pass dynamic link
        jalr $t2
        move $t1, $v0
        sw $t1, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_g:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 28      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        move $a0, $t0   # Assign values
        sub $sp, $sp, 4 # Move stack pointer
        sw $a0, ($sp)   # Write param into stack
        lw $t0, ($s0)   # Move up a static link
        lw $t1, 12($t0) # Load framesize for static link
        add $t1, $t1, $t0       # Seek to $fp [end of AR]
        lw $t0, -4($t1) # f
        lw $t2, ($t0)   # Get Fn address
        lw $v0, 4($t0)  # Get static link
        move $a1, $s0   # Pass dynamic link
        jalr $t2
        move $t1, $v0
        sub $sp, $sp, 4 # Move stack pointer
        sw $t1, ($sp)   # Write param into stack
        sw $t1, -8($fp) # Write out used local variable
        lw $t0, ($s0)   # Move up a static link
        lw $t1, 12($t0) # Load framesize for static link
        add $t1, $t1, $t0       # Seek to $fp [end of AR]
        lw $t0, -4($t1) # f
        lw $t2, ($t0)   # Get Fn address
        lw $v0, 4($t0)  # Get static link
```

```
        move $a1, $s0   # Pass dynamic link
        jalr $t2
        move $t1, $v0
        sw $t1, -12($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
# Make a new function variable entry
# Precondition: $v0 contains function address, $v1 contains static link
# Returns: address of allocated fn entry descriptor in $v0
rfunc:
        move $s1, $v0   # Backup fn address in $s1
        li $a0, 8       # Space required for descriptor
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        sw $s1, ($v0)   # Store fn address
        sw $v1, 4($v0)  # Store static link
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
```

```
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
7
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code produces the correct answer.

5.2.20 Test 20 - typecheck1.c

Input File

```
/*Result: FAIL*/

/*
*       Sample --C Type checking #1
*       This example should FAIL
*
*       Henry Thacker
*/

int myfn(int a) {
        return a + 2;
}


int main() {
        return myfn(myfn);
}
```

Purpose of Test

This example is supposed to cause a failure in the interpreter. Unfortunately the TAC generator is likely to still generate a TAC representation for this input, because it is unable to type check parameters. As a result, the code generator will also go on to generate an invalid MIPS assembly program.

Expected Result

The expected output for this test = Typecheck failure

Interpreter Result

The result from the interpreter is: Could not be interpreted

Generated TAC

```
BeginFn myfn
_myfn:
InitFrame 2
PopParam a
FnBody
_t1 = a + 2
Return _t1
EndFn
BeginFn main
_main:
```

```
InitFrame 1
FnBody
PrepareToCall 1
PushParam myfn
_t2 = CallFn _myfn
Return _t2
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 19:16:24 2010

.data
        EOL:    .asciiz "\n"
.text

_myfn:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 24      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        lw $a0, ($sp)   # Pop the parameter
        sw $a0, -4($fp) # Write param into heap
        sw $s7, ($sp)   # Save return address in stack
        lw $t1, -4($fp) # Load local variable
        addi $t0, $t1, 2
        move $v0, $t0   # Assign values
        sw $t0, -8($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 20      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        la $v0, _myfn   # Store address of function
        move $v1, $s0   # Store static link to call with
        jal rfunc       # Register fn variable
        move $t0, $v0   # Return fn descriptor address
        sub $sp, $sp, 4 # Move stack pointer
        sw $t0, ($sp)   # Write param into stack
```

```
        lw $v0, ($s0)   # Point callee to same static link as mine (
           caller)
        move $a1, $s0   # Pass dynamic link
        jal _myfn
        move $t0, $v0
        sw $t0, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
# Make a new function variable entry
# Precondition: $v0 contains function address, $v1 contains static link
# Returns: address of allocated fn entry descriptor in $v0
rfunc:
        move $s1, $v0   # Backup fn address in $s1
        li $a0, 8       # Space required for descriptor
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        sw $s1, ($v0)   # Store fn address
        sw $v1, 4($v0)  # Store static link
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
```

```
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
268566550
```

Test Analysis

Interpreter The interpreter gave the expected result - An integer parameter was expected to be passed to `myfn`, so the interpreter raised an error.

TAC Because the TAC generator does not type-check pushed parameters, it continues to generate a TAC translation for the invalid program.

MIPS The TAC representation given to the code generator is incorrect, so the generated MIPS program returns an undefined response when run in SPIM.

5.2.21 Test 21 - typecheck2.c

Input File

```
/*Result: FAIL*/

/*
*       Sample --C Type checking #2
*       This example should FAIL
*
*       Henry Thacker
*/

int myfn(int a) {
        return myfn;
}


int main() {
        return myfn(5);
}
```

Purpose of Test

This example is supposed to cause a type check failure in both the interpreter and the TAC generator. As a result, no MIPS assembly code should be generated.

Expected Result

The expected output for this test = Typecheck failure

Interpreter Result

The result from the interpreter is: Could not be interpreted

Generated TAC

```
Fatal exception: Expected integer return value
```

Generated MIPS Assembly

```
Can not be compiled
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
The following symbols are undefined:
```

```
main

[0x00400014] 0x0c000000  jal 0x00000000 [main]           ; 180: jal main
```

Test Analysis

Interpreter The interpreter gave the expected result - `myfn` was expected to return an integer not a function pointer, so the interpreter fails for this input.

TAC Unlike parameters, the TAC generator happily type-checks return values, so the TAC representation is never generated as the typing error is detected.

MIPS As no TAC representation is generated, we can't generate any MIPS code to run in SPIM.

5.2.22 Test 22 - voidfn.c

Input File

```
/*Result: 5*/

/*
*       Sample --C test calling void fn Programme
*       Henry Thacker
*/

int a = 1;

void fn() {
        a = 5;
}

int main() {
        fn();
        return a;
}
```

Purpose of Test

This test is designed to make sure that calling void functions work and that correct scope rules are observed. Because this test uses global variables, it is expected that the code generator will not work for this example.

Expected Result

The expected output for this test = Result: 5

Interpreter Result

The result from the interpreter is: 5

Generated TAC

```
a = 1
BeginFn fn
_fn:
InitFrame 0
FnBody
a = 5
EndFn
BeginFn main
_main:
InitFrame 1
FnBody
```

```
PrepareToCall 0
_t1 = CallFn _fn
Return a
EndFn
```

Generated MIPS Assembly

```
Can not be compiled
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
The following symbols are undefined:
main

[0x00400014] 0x0c000000  jal 0x00000000 [main]           ; 180: jal main
```

Test Analysis

Interpreter The interpreter gave the expected result

MIPS The code generator will not work for this example, because the example uses global variables. This restriction has already been noted in section 3.3.11.

5.2.23 Test 23 - while.c

Input File

```
/*Result: 5*/

/*
*       Sample --C While Loop Programme
*       Henry Thacker
*/

/* Main entry point */
int main() {
        int i = 5;
        while (i > 1)  {
                if (i==4) {
                        break;
                }
                if (i<4) {
                        return 9;
                }
                i = i - 1;
        }
        return 5;
}
```

Purpose of Test

This test is designed to check that the `break` statement within a while loop works.

Expected Result

The expected output for this test = Result: 5

Interpreter Result

The result from the interpreter is: 5

Generated TAC

```
BeginFn main
_main:
InitFrame 5
FnBody
i = 5
__while1:
_t1 = i > 1
If _t1 Goto __if1true
Goto __while1end
```

```
__if1true:
_t2 = i == 4
If _t2 Goto __if2true
Goto __if2end
__if2true:
Goto __while1end
__if2end:
_t3 = i < 4
If _t3 Goto __if3true
Goto __if3end
__if3true:
Return 9
__if3end:
_t4 = i - 1
i = _t4
Goto __while1
__if1end:
__while1end:
Return 5
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:56 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 36      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 5
__while1:
        li $t2, 1
        sgt $t1, $t0, $t2       # $c = $a > $b
        bne $t1, $zero, __if1true
        j __while1end
__if1true:
        li $t4, 4
        seq $t3, $t0, $t4       # $c = $a == $b
        bne $t3, $zero, __if2true
```

```
        j __if2end
__if2true:
        j __while1end
__if2end:
        slti $t5, $t0, 4        # $c = $a < b
        bne $t5, $zero, __if3true
        j __if3end
__if3true:
        li $v0, 9
        sw $t0, -4($fp) # Write out used local variable
        sw $t1, -8($fp) # Write out used local variable
        sw $t3, -12($fp)        # Write out used local variable
        sw $t5, -16($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
__if3end:
        lw $t1, -4($fp) # Load local variable
        sub $t0, $t1, 1
        move $t1, $t0   # Assign values
        j __while1
__if1end:
__while1end:
        li $v0, 5
        sw $t0, -20($fp)        # Write out used local variable
        sw $t1, -4($fp) # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
```

```
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
5
```

Test Analysis

Interpreter The interpreter gave the expected result

TAC There are cases where a couple of different labels are defined next to each other (i.e. `__if1end` and `__while1end`). The reference to one of these labels could be removed (and all references to it refactored) to save extra lines in the output. This change would benefit the code generation stage too.

MIPS The code produces the correct answer.

5.2.24 Test 24 - while2.c

Input File

```
/*Result: 3*/

/*
*       Sample --C While Loop #2 Programme
*       Henry Thacker
*/

/* Main entry point */
int main() {
        int i = 5;
        while (i > 1)  {
                i = i - 1;
                if (i==4) {
                        continue;
                }
                return i;
        }
        return 5;
}
```

Purpose of Test

This test is designed to check that the `continue` statement within a while loop works.

Expected Result

The expected output for this test = Result: 3

Interpreter Result

The result from the interpreter is: 3

Generated TAC

```
BeginFn main
_main:
InitFrame 4
FnBody
i = 5
__while1:
_t1 = i > 1
If _t1 Goto __if1true
Goto __while1end
__if1true:
_t2 = i - 1
```

```
i = _t2
_t3 = i == 4
If _t3 Goto __if2true
Goto __if2end
__if2true:
Goto __while1
__if2end:
Return i
Goto __while1
__if1end:
__while1end:
Return 5
EndFn
```

Generated MIPS Assembly

```
# Sun Jan 10 14:14:57 2010

.data
        EOL:    .asciiz "\n"
.text

_main:
        move $s7, $ra   # Store Return address in $s7
        li $a0, 32      # Store the frame size required for this AR
        jal mk_ar
        move $s0, $v0   # Store heap start address in $s0
        sub $sp, $sp, 4
        sw $s7, ($sp)   # Save return address in stack
        lw $t0, -4($fp) # Load local variable
        li $t0, 5
__while1:
        li $t2, 1
        sgt $t1, $t0, $t2       # $c = $a > $b
        bne $t1, $zero, __if1true
        j __while1end
__if1true:
        sub $t3, $t0, 1
        move $t0, $t3   # Assign values
        li $t5, 4
        seq $t4, $t0, $t5       # $c = $a == $b
        bne $t4, $zero, __if2true
        j __if2end
__if2true:
        j __while1
__if2end:
        move $v0, $t0   # Assign values
```

```
        sw $t0, -4($fp) # Write out used local variable
        sw $t1, -8($fp) # Write out used local variable
        sw $t3, -12($fp)        # Write out used local variable
        sw $t4, -16($fp)        # Write out used local variable
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
        j __while1
__if1end:
__while1end:
        li $v0, 5
        lw $ra, ($sp)   # Get return address
        add $sp, $sp, 4 # Pop return address from stack
        lw $fp, 4($s0)  # Load previous frame ptr
        lw $s0, 8($s0)  # Load dynamic link
        jr $ra  # Jump to $ra
# Make a new activation record
# Precondition: $a0 contains total required heap size, $a1 contains
   dynamic link, $v0 contains static link
# Returns: start of heap address in $v0, heap contains reference to
   static link and old $fp value
mk_ar:
        move $s1, $v0   # Backup static link in $s1
        li $v0, 9       # Allocate space systemcode
        syscall # Allocate space on heap
        move $s2, $fp   # Backup old $fp in $s2
        add $fp, $v0, $a0       # $fp = heap start address + heap size
        sw $s1, ($v0)   # Save static link
        sw $s2, 4($v0)  # Save old $fp
        sw $a1, 8($v0)  # Save dynamic link
        sw $a0, 12($v0) # Save framesize
        jr $ra
        .globl main
main:
        move $a1, $zero # Zero dynamic link
        move $v0, $zero # Zero static link
        jal _main
        move $a0, $v0   # Retrieve the return value of the main
           function
        li $v0, 1       # Print integer
        syscall
        li $v0, 4       # Print string
        la $a0, EOL     # Printing EOL character
        syscall
        li $v0, 10      # System exit
```

```
        syscall
```

SPIM Output

```
SPIM Version 7.4 of January 1, 2009
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /opt/local/share/spim/exceptions.s
3
```

Test Analysis

Interpreter The interpreter gave the expected result

TAC There are cases where a couple of different labels are defined next to each other (i.e. `__if1end` and `__while1end`). The reference to one of these labels could be removed (and all references to it refactored) to save extra lines in the output. This change would benefit the code generation stage too. Additionally some of the arithmetic operations are suboptimal, for instance: `_t2 = i - 1` followed by `i = _t2`, would be written better as: `i = i - 1`.

MIPS The code produces the correct answer.

Chapter 6

Critical Analysis

6.1 General Comments

Given the limited time that we were allocated for this project, the level of progression that has been made through the various stages of this project is really satisfying. The interpreter is essentially fully functional and the other two sections only have a few minor flaws to remedy. Not only has this project been a great aid to further our understanding in the Compilers field, but has also led to a greater appreciation of System Architecture. It is a shame that this project has been our only exposure to assembler during the course. There are some general comments that can be made regarding the state of the various parts of the compiler (in no particular order).

Due to the size of the project and the number of implementation files, there will likely be some very small parts of code which were never used. The codebase has grown fairly large now and would probably benefit from being more rigorously documented to ensure that no functions exist that duplicate functionality of others or any that may never even be called.

Memory management is an area that would have to be looked at for this to ever transition into a serious compiler. The only place where memory is explicitly freed is where optimisations take place. With the limited use of a language without strings and a large standard library, it is difficult to see this becoming a problem in practice, but it still something to consider. Some of the functions, particularly those heavily used such as `store()` in `environment.c` are fairly unwieldy and have ended up with a multitude of different parameters for different contexts. Such large functions would be far clearer and thus less prone to bugs if they were rewritten and tested as smaller functions.

The coding standard is consistent throughout, all externally visible functions are declared in header files to try and give a greater sense of modularity. The code was compiled with the C89 (ANSI) and pedantic flags to try and clear up any non-standard code. While a test harness was developed for the generated programs, it would have also made sense to also look at testing the important functions within the coursework code itself to ensure that the functions all work as expected. In a few places within the project the `assert()` function is used (defined in `utilities.c`) to ensure that the internal state of the program is as expected. It would have been beneficial to introduce this to more places within the code.

Due to the number of different types of data that can be output by the various stages of the compiler, the use of `var_args` was really beneficial in this project. Using `var_args` means that

commonly used functions such as `fatal` and `make_label_operand` are able to make use of variable numbers of arguments of varying types. This saves space and complexity by reducing the necessity of writing multiple "type specific" functions.

6.2 Interpreter

The interpreter is the most functional part of the whole system, there are only two known issues. The first issue is that writing `my_function(void)`, where the `void` denotes no parameters, will parse correctly but is not interpreted. Unfortunately this issue was picked up fairly late, but it would be fairly trivial to fix. The parameter-list in these cases simply contains the integer constant `VOID`. A simple `IF` statement could disregard this when storing the function in the environment.

The second issue is that although return values are type checked, no error is thrown if **no** value is returned and the function definition expects a value to be returned. The current behaviour is actually more like the behaviour in C. In order to implement similar behaviour to Java, it would be necessary to follow every possible branch to check that each branch returns a value.

The environment structure was created as part of the implementation of the interpreter. It uses a hash table implementation to minimise the cost of accessing and storing values within the environment. The environment has a lot of utility functions that are used through every part of the project. Unfortunately this utility is offset by quite a lot of complexity that is introduced through making these functions flexible.

Another good part of the implementation is that the interpreter has full type checking for both parameters and return values. The associated functions were purposely written so that they could be reused in other parts of the project.

6.3 TAC Generator

The TAC Generator works for almost all of the test cases that were chosen. One issue with the TAC generator is that it does not accept input from standard input (in order to load user-specified TAC instructions directly into the code generator). This particular feature was alluded to in the coursework specification, but unfortunately not implemented due to time pressure. The only way to enter data into the TAC generator is via the parser.

One of the main problems with the TAC generator at the moment is that it does not type check parameter values that are passed to functions. This means that some programs that are not accepted by the interpreter will be accepted by the TAC generator. This behaviour is obviously incorrect, but luckily the solution is already implemented in the interpreter. To fix this behaviour in the TAC generator, it would be a case of changing the `push_params` procedure to take a reference to the callee function, so that the formal parameters can be compared to the actual parameters.

Some superfluous TAC instructions are present in the instruction set that are not used in the MIPS code generator (`TT_FN_DEF`, `TT_PREPARE`). The reason for this is that the TAC instruction set was developed prior to writing any of the code generator implementation. These additional instructions are not necessarily problematic as, in fact, they only serve to demonstrate the difference

between what information is needed in each stage of the compiler. If further code generators were created for different architectures, they may benefit from these additional instructions, this is why these instructions have been left as part of the instruction set.

While the interpreter supports separate nested scopes within `IF` and `WHILE` statements, this functionality was not included in the TAC generator. This omission was an oversight and it was noted too late to include in the final code. Logically this feature is no more difficult to support in the TAC generator than in the interpreter.

Considering that the TAC instruction set was designed completely in advance of even researching MIPS, the instruction set has shown itself as ample to support the full code-generation procedure.

6.4 MIPS Assembler Compiler

While the MIPS code generator does have several weaknesses, it is actually very capable as demonstrated by the test results. As probably the most complicated part of the project, it is unsurprising that several problems exist within the implementation.

One issue with the MIPS code generation is that nested functions cannot share the same names, even if they reside in different scopes. The reason for this is that the user defined names are used to generate the function labels. If two functions have the same name, then they will produce the same function label. Obviously at run-time the processor will be unable to determine which one of the functions to invoke and a fatal error will be raised.

MIPS instructions are created within the code generator as a series of `mips_instruction` structures. In order to create these structures, a helper function called `mips` is provided. Because the `mips` function caters for the definition of any valid MIPS instruction, the function contains many parameters. As a result, wherever `mips` is called, a very long line of code needs to be written. Because this function is used in many places within `mips.c`, the whole file looks fairly untidy. To remedy this problem, it would be better to split the MIPS instruction set into different types of instruction, depending on the number and type of parameters that they cater for. This would enable much more compact `mips_instruction` helper functions to be created.

Currently no analysis of basic blocks or functions is undertaken to ascertain what runtime support is needed. A lot more could be done to examine whether functions call other functions, whether they use function-typed variables, etc. This analysis would potentially mean that large amounts of code could removed. At the moment, every function has a complex activation record created upon function entry. If the function does not call another function and we have enough registers free, we could actually avoid creating an activation record at all. Additionally, some analysis could be done to ascertain how many registers are required by a function. If the function only requires one or two registers, then it may be possible to preserve all of the caller's registers, instead of writing them out into memory.

The reserved argument registers $a0-$a3 are also not used to pass the first four arguments. It was a shame that there was not time to implement this feature. We might even be able to save some activation record space if most of our arguments could fit into registers. At the moment, the code generator takes up a lot of space in the heap due to the number of Activation Records that

are allocated. If this were being implemented in a real machine it would be necessary to use some type of garbage collection to reclaim the space that is allocated on the heap. Another waste in the code generator is that user program's are only able to use the $t registers. Initially this was the case because the compiler needed to use some registers for certain operations. The solution will be discussed in the Further Improvements section.

The optimisation that is currently performed within `optimisation.c` is used to remove redundant patterns of code. These fragments of code are introduced by exceptional cases that occur with certain inputs to the code generation procedure. Ideally some form of standard code optimisation would have been implemented, but unfortunately there was not enough time to fully research and implement these techniques. The basic optimisations that **are** provided show that such optimisation is possible and has been considered.

A final problem with the code generation phase is that global variables do not work. The solution to this is also fairly simple and will be covered in the next section.

Chapter 7

Further Improvements

The critical analysis has highlighted some areas that could improved as part of a future project. This section will detail some of these ideas in further depth along with possible implementation approaches.

7.1 Register Use

One of the main criticisms surrounding the code generator are the register allocation choices that it makes. Currently it uses a version of a very simple approach that was discussed in lectures. Although this approach works, it generates code that would be many factors slower than a properly optimised program. Within the test cases, it is possible to see places where suboptimal decisions are made - for instance `bigmath.c`, which was purposely designed to test register spills. Luckily, much research has been done into effective register allocation and there are documented methods such as graph colouring that aim to optimise the choices that are made at compile time. Although the graph colouring algorithm is generally an NP-complete problem, there are algorithms that are better in practice.

The first step to improving the register allocation procedure is to implement a liveness check. This will enable us to make better decisions about which registers can be considered for reuse. A more comprehensive approach would be to implement a full graph colouring algorithm.

Currently, the compiler will only allow the $t registers to be used for application data, because it reserves many of the $s registers for its own use. Ideally, after a better register allocation procedure has been implemented, the compiler shouldn't need to **reserve** any registers. Instead, when the compiler needs to use a register, it can use the same register allocation procedure as followed for user data. This will ensure that more variables can be present in registers, which tends to lead to better optimised programs.

7.2 TAC Parsing

As noted in the document, it is currently not possible to load TAC instructions directly into the code generator. In order to implement this, a basic lexer would need to be written in order to generate the correct `tac_quad` structures from the inputted file. Based on the instruction set that was presented in section 4.1, the input file could be examined line-by-line and split using whitespace characters

as a delimiter. At this point we would be able to examine what type of instruction is being created and initialise the corresponding `tac_quad` structure. In addition to this, we would need to fill the environment "on the fly" as is currently done within the TAC generator. One problem with loading TAC instructions from a file, is that we have lost much of the type information that is given to us by the original environment structure. This would make it impossible to perform type checking without rewriting the instruction set to provide this information.

7.3 Optimisation

As a further extension to the submitted code, it would be good to look at partitioning the code into basic blocks. Once the code is in basic blocks, it is possible to optimise the blocks as if they were individual programs. Although some basic optimisations exist in `optimisation.c`, these operate on the generated MIPS code. Most of the genuinely helpful optimisations occur during the TAC generation stage over a basic block. The "copy propagation" optimisation would be particularly beneficial for some of the code that is generated by the TAC generator. As discussed in tests such as `indirection.c`, return values from function calls are typically stored in a temporary and then assigned straight into another variable. The "copy propagation" optimisation would eliminate redundant code like this.

Another optimisation technique "constant propagation" could be used to reduce simple arithmetic calculations, such as those in `bigmath.c`, at compile time. For a limited language like --C , this type of optimisation is likely to have a great effect, as most example scripts tend to rely on arithmetic.

Upon splitting the program up into basic blocks, there will be situations where whole blocks of code can be removed. The definitions of functions which are not even used are currently translated into MIPS code. Also, `IF` statements which branch on constants / variables, such as in `iftest.c` could also be hugely simplified using a similar procedure.

7.4 Supporting Global Variables

In order to support global variables in the code generation phase, it would be necessary to re-create the idea of a "global environment" in the code generator. To do this, the global environment would be found and the number of local variables would be checked. Within the main method, a space would be allocated within the heap of the required size. When looping over the `tac_quad`s, if we see a variable assignment and we are not within an function, then the variable could be added to the correct place within the global environment.

Chapter 8

Source Code

8.1 C files

8.1.1 frontend folder

main.c

```
#include <stdio.h>
#include <ctype.h>
#include "nodes.h"
#include "C.tab.h"
#include <string.h>
#include "interpreter.h"
#include "tacgenerator.h"
#include "mips.h"

#define MODE_PARSE 0
#define MODE_TAC_GEN 1
#define MODE_INTERPRET 2
#define MODE_MIPS 3

char *named(int t)
{
    static char b[100];
    if (isgraph(t) || t==' ') {
      sprintf(b, "%c", t);
      return b;
    }
    switch (t) {
      default: return "???";
    case IDENTIFIER:
      return "id";
    case CONSTANT:
      return "constant";
    case STRING_LITERAL:
      return "string";
```

```
    case LE_OP:
      return "<=";
    case GE_OP:
      return ">=";
    case EQ_OP:
      return "==";
    case NE_OP:
      return "!=";
    case EXTERN:
      return "extern";
    case AUTO:
      return "auto";
    case INT:
      return "int";
    case VOID:
      return "void";
    case APPLY:
      return "apply";
    case LEAF:
      return "leaf";
    case IF:
      return "if";
    case ELSE:
      return "else";
    case WHILE:
      return "while";
    case CONTINUE:
      return "continue";
    case BREAK:
      return "break";
    case RETURN:
      return "return";
    }
}

void print_leaf(NODE *tree, int level)
{
    TOKEN *t = (TOKEN *)tree;
    int i;
    for (i=0; i<level; i++) putchar(' ');
    if (t->type == CONSTANT) printf("%d\n", t->value);
    else if (t->type == STRING_LITERAL) printf("\"%s\"\n", t->lexeme);
    else if (t) puts(t->lexeme);
}

void print_tree0(NODE *tree, int level)
{
```

```
    int i;
    if (tree==NULL) return;
    if (tree->type==LEAF) {
      print_leaf(tree->left, level);
    }
    else {
      for(i=0; i<level; i++) putchar(' ');
      printf("%s\n", named(tree->type));
/*      if (tree->type=='~') { */
/*        for(i=0; i<level+2; i++) putchar(' '); */
/*        printf("%p\n", tree->left); */
/*      } */
/*      else */
        print_tree0(tree->left, level+2);
      print_tree0(tree->right, level+2);
    }
}

void print_tree(NODE *tree)
{
    print_tree0(tree, 0);
}

extern int yydebug;
extern NODE* yyparse(void);
extern NODE* ans;
extern void init_symbtable(void);

char *mode_to_string(int mode)
{
        switch(mode)
        {
                case(MODE_PARSE):
                        return "parse tree";
                case(MODE_INTERPRET):
                        return "interpret";
                case(MODE_TAC_GEN):
                        return "three address code generator";
                case(MODE_MIPS):
                        return "MIPS code generator";
                default:
                        return "Unknown";
        }
}

/* Start processing the input per the relevant mode */
void process(NODE *tree, int run_mode)
```

```
{
        switch(run_mode)
        {
                case(MODE_PARSE):
                        printf("\nparse finished with %p\n", tree);
                print_tree(tree);
                        break;
                case(MODE_INTERPRET):
                        start_interpret(tree);
                        break;
                case(MODE_TAC_GEN):
                        print_tac(start_tac_gen(tree));
                        break;
                case(MODE_MIPS):
                        code_gen(tree);
                        break;
                default:
                        return;
        }
}

/* Print out usage instructions */
void print_runflags() {
        printf("--C Compiler - Input syntax\n");
        printf("\n");
        printf("./mycc [-p|-i|-t|-m] < input_source\n");
        printf("\n");
        printf("-p = parse mode - print out the parse tree and
           terminate\n");
        printf("-i = interpret - interpret the parse tree and print the
            result\n");
        printf("-t = TAC generator - print out TAC representation of
           the programme\n");
        printf("-m = MIPS generator - generate MIPS machine code\n");
        printf("\n");
}

int main(int argc, char** argv)
{
    NODE* tree;
        int mode = MODE_PARSE;
    if (argc==1)
        {
                /* if no flag specified - print out flags */
                print_runflags();
        }
        else
```

```
        {
                /* Define the allowed parameters */
                if (strcmp(argv[1],"-p")==0) mode=MODE_PARSE;
                if (strcmp(argv[1],"-i")==0) mode=MODE_INTERPRET;
                if (strcmp(argv[1],"-t")==0) mode=MODE_TAC_GEN;
                if (strcmp(argv[1],"-m")==0) mode=MODE_MIPS;
                init_symbtable();
            yyparse();
            tree = ans;
                printf("\n--------\n");
                process(tree, mode);
        }
    return 0;
}
```

8.1.2 interpreter folder

arithmetic.c

```
#include "arithmetic.h"

/**
*       arithmetic.c by Henry Thacker
*       Version 2 - Rewritten 14/11/2009
*
*       Arithmetic operations for the --C language
*
*/

value *arithmetic(environment *env, int operator, value *operand1,
   value *operand2) {
        int i_operand1 = to_int(env, operand1);
        int i_operand2 = to_int(env, operand2);
        switch(operator) {
                case '+':
                        return int_value(i_operand1 + i_operand2);
                case '-':
                        return int_value(i_operand1 - i_operand2);
                case '*':
                        return int_value(i_operand1 * i_operand2);
                case '/':
                        return int_value(i_operand1 / i_operand2);
                case '%':
                        return int_value(i_operand1 % i_operand2);
                case '>':
                        return int_value(i_operand1 > i_operand2);
                case '<':
                        return int_value(i_operand1 < i_operand2);
                case NE_OP:
```

```
                        return int_value(i_operand1 != i_operand2);
                case EQ_OP:
                        return int_value(i_operand1 == i_operand2);
                case LE_OP:
                        return int_value(i_operand1 <= i_operand2);
                case GE_OP:
                        return int_value(i_operand1 >= i_operand2);
                case '!':
                        return int_value(!i_operand1);
                default:
                        return NULL;
        }
}
```

interpreter.c

```
#include "interpreter.h"

/**
*       interpreter.c by Henry Thacker
*       Version 2 - Rewritten 14/11/2009
*
*       Interpreter for the --C language
*
*/

/* Assign variable */
/* Assign data to identifier in env */
value *assign(environment *env, value *identifier, value *data, int
   is_declarator) {
        if (env==NULL || identifier==NULL) return;
        return store(env, data->value_type, to_string(identifier), data
           , 0, is_declarator, 0, 0);
}

/* Count number of parameters */
int param_count(value *val) {
        int count = 0;
        value *current_param;
        if (val == NULL) return 0;
        if (val->value_type == VT_FUNCTN) {
                if (val->data.func->params == NULL) return 0;
                 current_param = val->data.func->params;
                while (current_param) {
                        current_param = current_param->next;
                        count++;
                }
        }
```

```
        else {
                /* We must have a function call with some chain of
                   parameters to count */
                current_param = val;
                while (current_param) {
                        current_param = current_param->next;
                        count++;
                }
        }
        return count;
}

/* Check whether main entry point exists */
value *main_entry_point(environment *env) {
        value *main_fn = search(env, "main", VT_FUNCTN, INT, 0);
        if (!main_fn) return NULL;
        /* There must be no parameters to the main fn */
        if (!main_fn->data.func->params) {
                return main_fn;
        }
        fatal("Entry point - int main() must have no parameters");
        return NULL;
}

/* Execute a function */
value *execute_fn(environment *env, value *fn_reference, value *params,
    int flag) {
        NODE *node;
        environment *definition_env;
        if (!fn_reference) {
                fatal("Function is undefined");
        }
        else {
                environment *new_env;
                /* Check parameter count */
                if (param_count(params) == param_count(fn_reference)) {
                        node = fn_reference->data.func->node_value;
                        definition_env = fn_reference->data.func->
                           definition_env;
                        new_env = create_environment(definition_env);
                        /* Copy parameters into environment */
                        if (param_count(params) > 0) {
                                define_parameters(new_env, fn_reference
                                   , params, env);
                        }
                        return evaluate(new_env, node, flag,
                           fn_reference->data.func->return_type);
```

```
                }
                else {
                        fatal("Formal and actual parameter count
                           differs");
                        return NULL;
                }
        }
}

/* Build a new function definition - don't store in environment yet
   though */
/* This data structure is further embellished within the recursive
   evaluate fn */
value *build_function(environment *env, value *fn_name, value *
   param_list) {
        value *tmp_value = calloc(1, sizeof(value));
        char *identifier = to_string(fn_name);
        function_declaration *fn_value = calloc(1, sizeof(
           function_declaration));
        tmp_value->value_type = VT_FUNCTN;
        tmp_value->identifier = malloc(sizeof(char) * (strlen(
           identifier) + 1));
        strcpy(tmp_value->identifier, identifier);
        fn_value->params = param_list;
        fn_value->definition_env = env;
        tmp_value->data.func = fn_value;
        return tmp_value;
}

/* Return NULL function for pointing at uninitialized function ptrs */
value *build_null_function() {
        return build_function(NULL, string_value("$NULL_FN"), NULL);
}

/* Declare variables underneath a declarator tree */
void declare_variables(environment *env, NODE *node, int variable_type,
    int return_type) {
        value *variable_name = NULL;
        value *variable_value = NULL;
        if (env == NULL || node == NULL) {
                return;
        }
        else if (type_of(node) == ',') {
                declare_variables(env, node->left, variable_type,
                   return_type);
                declare_variables(env, node->right, variable_type,
                   return_type);
```

```
        return;
}
else if (type_of(node) == '=') { /* Specific assignment */
        variable_name = evaluate(env, node->left, 0,
           return_type);
        variable_value = evaluate(env, node->right, 0,
           return_type);
}
else if (type_of(node) == LEAF) { /* Undefined assignment */
        variable_name = evaluate(env, node->left, 0,
           return_type);
}
/* Assign variable */
if (variable_name) {
        if (variable_value) {
                /* If variable_value is a string, we need to do
                    a fn/variable lookup */
                if (variable_value->value_type == VT_STRING) {
                        value *old_name = variable_value;
                        variable_value = get(env, to_string(
                           variable_value));
                        if (!variable_value) {
                                fatal("Could not find
                                   identifier '%s'", to_string(
                                   old_name));
                        }
                }
                /* We have to assign the specified initial
                   value, AFTER typechecking */
                type_check_assignment(variable_name,
                   variable_value, variable_type);
                assign(env, variable_name, variable_value, 1);
        }
        else {
                /* Assign a default initialization value for
                   this type */
                switch(variable_type) {
                        case INT:
                                assign(env, variable_name,
                                   int_value(0), 1);
                                break;
                        case VOID:
                                assign(env, variable_name,
                                   void_value(), 1);
                                break;
                        case FUNCTION:
```

```
                                        assign(env, variable_name,
                                            null_function, 1);
                                        break;
                                }
                        }
                }
        }
        else {
                fatal("Could not ascertain variable name!");
        }
}

/* Go down the declarator tree initialising the variables, at this
    stage */
void register_variable_subtree(environment *env, NODE *node, int
    return_type) {
        NODE *original_node = node;
        /* Ensure we have all required params */
        if (!env || !node || type_of(node) != '~') return;
        /* Skip over LEAF nodes */
        if (node->left != NULL && type_of(node->left) == LEAF) {
                node = node->left;
        }
        if (node->left != NULL && (type_of(node->left) == VOID ||
            type_of(node->left) == FUNCTION || type_of(node->left) ==
            INT)) {
                /* Find variable type */
                int variable_type = to_int(NULL, evaluate(env, node->
                    left, 0, return_type));
                declare_variables(env, original_node->right,
                    variable_type, return_type);
        }
}

/* Recursive evaluation of AST */
value *evaluate(environment *env, NODE *node, int flag, int return_type
    ) {
        value *lhs = NULL, *rhs = NULL, *temp = NULL;
        environment *new_env;
        char *identifier;
        /* Check if we were passed an invalid node */
        if (!node) return NULL;
        switch(type_of(node)) {
                case '+':
                case '-':
                case '*':
                case '/':
                case '%':
```

```
case '>':
case '<':
case NE_OP:
case EQ_OP:
case LE_OP:
case GE_OP:
case '!':
        lhs = evaluate(env, node->left, flag,
            return_type);
        rhs = evaluate(env, node->right, flag,
            return_type);
        return arithmetic(env, type_of(node), lhs, rhs)
            ;
case '=':
        lhs = evaluate(env, node->left, flag,
            return_type);
        rhs = evaluate(env, node->right, flag,
            return_type);
        if (rhs && rhs->value_type!=VT_INTEGR && rhs->
            value_type!=VT_FUNCTN) {
                if (rhs->value_type == VT_STRING) {
                        rhs = get(env, rhs->data.
                            string_value);
                }
                else {
                        rhs = get(env, rhs->identifier)
                            ;
                }
                if (!rhs) fatal("Undeclared identifier"
                    );
        }
        /* Check the LHS variable has already been
            defined */
        temp = get(env, to_string(lhs));
        assert(temp!=NULL, "Variable not defined");
        /* Type check the assignment */
        type_check_assignment(lhs, rhs, vt_type_convert
            (temp->value_type));
        assign(env, lhs, rhs, 0);
        return NULL;
case APPLY:
        /* FN Name */
        lhs = evaluate(env, node->left, flag,
            return_type);
        /* Params */
        rhs = evaluate(env, node->right, flag,
            return_type);
```

```
        /* Lookup function */
        temp = search(env, to_string(lhs), VT_FUNCTN,
           VT_ANY, 1);
        return execute_fn(env, temp, rhs, flag);
case IDENTIFIER:
        return string_value(cast_from_node(node)->
           lexeme);
case CONSTANT:
        return int_value(cast_from_node(node)->value);
case VOID:
case FUNCTION:
case INT:
        return int_value(type_of(node));
case LEAF:
        return evaluate(env, node->left, flag,
           return_type);
case IF:
        new_env = create_environment(env);
        /* LHS is condition */
        lhs = evaluate(env, node->left, flag,
           return_type);
        /* Store temporary value indicating condition
           outcome */
        assign(new_env, string_value(IF_EVAL_SYMBOL),
           lhs, 1);
        if (to_int(env, lhs)) {
                /* Condition is true */
                return evaluate(new_env, node->right,
                   flag, return_type);
        }
        else {
                /* We need to look at whether the RHS
                   is ELSE */
                if (type_of(node->right) == ELSE) {
                        /* we need to traverse into the
                           else */
                        return evaluate(new_env, node->
                           right, flag, return_type);
                }
        }
        return NULL;
case ELSE:
        temp = last_if_evaluation(env);
        if (!temp) fatal("Could not find evaluation of
           last IF statement");
        if (to_int(env, temp)) {
```

```
                return evaluate(env, node->left, flag,
                    return_type);
        }
        else {
                return evaluate(env, node->right, flag,
                    return_type);
        }
        return NULL;
case CONTINUE:
        return string_value(CONTINUE_EVAL_SYMBOL);
case BREAK:
        return string_value(BREAK_EVAL_SYMBOL);
case WHILE:
        new_env = create_environment(env);
        while(to_int(new_env, evaluate(new_env, node->
            left, flag, return_type))) {
                rhs = evaluate(new_env, node->right,
                    flag, return_type);
                if (rhs!=NULL && rhs->value_type==
                    VT_STRING && strcmp(to_string(rhs),
                    BREAK_EVAL_SYMBOL)==0) {
                        break;
                }
                else if (rhs!=NULL && rhs->value_type==
                    VT_STRING && strcmp(to_string(rhs),
                    CONTINUE_EVAL_SYMBOL)==0) {
                        continue;
                }
                else if (rhs!=NULL) {
                        return rhs;
                }
        }
        return NULL;
case 'D':
        /* LHS is FN definition */
        /* LHS is executed in current environment */
        lhs = evaluate(env, node->left, flag,
            return_type);
        if (lhs!=NULL) {
                /* Point function to the correct fn
                    body */
                lhs->data.func->node_value = node->
                    right;
                /* Store function definition in
                    environment */
                store_function(env, lhs, NULL);
        }
```

```
        return NULL;
case 'd':
        /* LHS is the type */
        lhs = evaluate(env, node->left, flag,
           return_type);
        /* RHS is fn name & params */
        rhs = evaluate(env, node->right, flag,
           return_type);
        /* Store return type */
        rhs->data.func->return_type = to_int(env, lhs);
        return rhs;
case 'F':
        /* FN name in LHS */
        lhs = evaluate(env, node->left, flag,
           return_type);
        /* Pull our parameters */
        rhs = evaluate(env, node->right,
           INTERPRET_PARAMS, return_type);
        return build_function(env, lhs, rhs);
case RETURN:
        lhs = evaluate(env, node->left, flag,
           return_type);
        /* Provide lookup for non-constants */
        if (lhs && lhs->value_type!=VT_INTEGR) {
                if (lhs->value_type == VT_STRING) {
                        lhs = get(env, lhs->data.
                           string_value);
                }
                else {
                        lhs = get(env, lhs->identifier)
                           ;
                }
                if (!lhs) fatal("Undeclared identifier"
                   );
        }
        type_check_return(lhs, return_type);
        return lhs;
case ',':
        lhs = evaluate(env, node->left, flag,
           return_type);
        rhs = evaluate(env, node->right, flag,
           return_type);
        if (lhs && rhs) {
                return join(lhs, rhs);
        }
        return NULL;
case '~':
```

```
                        /* Do not pre-register formal parameters */
                        if (flag != INTERPRET_PARAMS) {
                                /* First sweep - initialise variables
                                   with correct type - assignment
                                   typechecking done here */
                                register_variable_subtree(env, node,
                                   return_type);
                        }
                        /* Variable Type */
                        lhs = evaluate(env, node->left, flag,
                           return_type);
                        /* Variable Name */
                        rhs = evaluate(env, node->right, flag,
                           return_type);
                        if (flag == INTERPRET_PARAMS) {
                                return int_param(to_string(rhs), to_int
                                   (env, lhs));
                        }
                        return NULL;
                        /* return rhs; */
                case ';':
                        lhs = NULL;
                        rhs = NULL;
                        if (node->left!=NULL) lhs = evaluate(env, node
                           ->left, flag, return_type);
                        if (lhs==NULL) {
                                if (node->right!=NULL) rhs = evaluate(
                                   env, node->right, flag, return_type)
                                   ;
                                if (rhs==NULL) {
                                        return NULL;
                                }
                                else {
                                        return rhs;
                                }
                        }
                        else {
                                return lhs;
                        }
                default:
                        printf("Unrecognised token type: %d\n", type_of
                           (node));
                        return NULL;
        }
}

/* Start the interpretation process at the top of the AST */
```

```
void start_interpret(NODE *start) {
        /* Store a reference to the NULL function */
        null_function = build_null_function();
        initial_environment = create_environment(NULL);
        evaluate(initial_environment, start, INTERPRET_FN_SCAN, INT);
        debug("Function scan complete.");
        if (main_entry_point(initial_environment)) {
                value *return_value;
                debug("Entry point - int main() found");
                debug("Starting full interpretation...");
                return_value = execute_fn(initial_environment,
                   main_entry_point(initial_environment), NULL,
                   INTERPRET_FULL);
                print_return_value(initial_environment, return_value);
        }
        else {
                fatal("Entry point - int main() NOT found!");
        }
}
```

8.1.3 mips folder

codegen_utils.c

```
#include "codegen_utils.h"

/**
*       codegen_utils.c by Henry Thacker
*
*       Utilised used by the MIPS code generator
*
*/

/* Print out the operand value */
void print_mips_operand(mips_instruction *instruction, int operand) {
        int type;
        union operand *operand_value;
        int more_operands = 0;
        switch(operand) {
                case 1:
                        type = instruction->operand1_type;
                        operand_value = instruction->operand1;
                        more_operands = instruction->operand2 &&
                           instruction->operand2_type != OT_UNSET;
                        break;
                case 2:
                        type = instruction->operand2_type;
                        operand_value = instruction->operand2;
```

```
                more_operands = instruction->operand3 &&
                    instruction->operand3_type != OT_UNSET;
                break;
            case 3:
                type = instruction->operand3_type;
                operand_value = instruction->operand3;
                more_operands = 0;
                break;
        }
        if (!operand_value || type==OT_UNSET) return;
        switch(type) {
            case OT_REGISTER:
                printf("%s%s", register_name(operand_value->reg
                    ), more_operands ? ", " : "");
                break;
            case OT_OFFSET:
                if (operand_value->reg_offset->offset == 0) {
                    printf("(%s)%s", register_name(
                        operand_value->reg_offset->reg),
                        more_operands ? ", " : "");
                }
                else {
                    printf("%d(%s)%s", operand_value->
                        reg_offset->offset, register_name(
                        operand_value->reg_offset->reg),
                        more_operands ? ", " : "");
                }
                break;
            case OT_COMMENT:
                printf("%s%s", operand_value->label,
                    more_operands ? ", " : "");
                break;
            case OT_CONSTANT:
                printf("%d%s", operand_value->constant,
                    more_operands ? ", " : "");
                break;
            case OT_FN_LABEL:
                printf("_%s%s", operand_value->label,
                    more_operands ? ", " : "");
                break;
            case OT_ZERO_ADDRESS:
            case OT_LABEL:
                printf("%s%s", operand_value->label,
                    more_operands ? ", " : "");
                break;
        }
}
```

```
/* Write MIPS code */
void print_mips(mips_instruction *instructions) {
        int i = 0;
        if (!instructions) return;
        for (i = 0; i < instructions->indent_count; i++) {
                /* Indent as requested */
                printf("\t");
        }
        /* Check to see if we have a single operand by itself (with no
           operation specified) - needs printing in a different way*/
        if (instructions->operation && strlen(instructions->operation)
           == 0 && instructions->operand1_type != OT_UNSET &&
           instructions->operand2_type == OT_UNSET && instructions->
           operand3_type == OT_UNSET) {
                print_mips_operand(instructions, 1);
                if (instructions->operand1_type != OT_ZERO_ADDRESS &&
                   instructions->operand1_type != OT_COMMENT) printf(":
                   ");
        }
        else {
                printf("%s ", instructions->operation);
                print_mips_operand(instructions, 1);
                print_mips_operand(instructions, 2);
                print_mips_operand(instructions, 3);
        }
        if (DEBUG_ON && instructions->comment && strlen(instructions->
           comment) > 0) {
                printf("\t# %s\n", instructions->comment);
        }
        else {
                printf("\n");
        }
        print_mips(instructions->next);
}

/* Create a new mips instruction */
mips_instruction *mips(char *operation, int op1type, int op2type, int
   op3type, operand *op1, operand *op2, operand *op3, char *comments,
   int indent_count) {
        mips_instruction *instruction = (mips_instruction *)malloc(
           sizeof(mips_instruction));
        /* Copy operation */
        instruction->operation = malloc(sizeof(char) * (strlen(
           operation) + 1));
        strcpy(instruction->operation, operation);
        /* Copy comments */
```

```
        instruction->comment = malloc(sizeof(char) * (strlen(comments)
            + 1));
        strcpy(instruction->comment, comments);
        /* Assign operands */
        instruction->operand1_type = op1type;
        instruction->operand2_type = op2type;
        instruction->operand3_type = op3type;
        instruction->operand1 = op1;
        instruction->operand2 = op2;
        instruction->operand3 = op3;
        /* Set indentation for printing */
        instruction->indent_count = indent_count;
        instruction->next = NULL;
        return instruction;
}

mips_instruction *syscall(char *comment) {
 return mips("", OT_ZERO_ADDRESS, OT_UNSET, OT_UNSET,
     make_label_operand("syscall"), NULL, NULL, comment, 1);
}

/* Generate a pseudo-instruction to hold a comment */
mips_instruction *mips_comment(operand *comment, int indent_count) {
        mips_instruction *instruction = (mips_instruction *)malloc(
            sizeof(mips_instruction));
        instruction->operation = "";
        /* Comment moved into a special operand */
        instruction->comment = "";
        instruction->operand1_type = OT_COMMENT;
        instruction->operand2_type = OT_UNSET;
        instruction->operand3_type = OT_UNSET;
        instruction->operand1 = comment;
        /* Set indentation for printing */
        instruction->indent_count = indent_count;
        instruction->next = NULL;
        return instruction;
}

operand *new_operand() {
        return (operand *)malloc(sizeof(operand));
}

operand *make_constant_operand(int constant) {
        operand *tmp = new_operand();
        tmp->constant = constant;
        return tmp;
}
```

```
/* Register specific operand creators */

operand *make_register_operand(int reg_identifier) {
        operand *tmp = new_operand();
        tmp->reg = reg_identifier;
        return tmp;
}

operand *make_offset_operand(int reg_identifier, int offset_amount) {
        operand *tmp = new_operand();
        register_offset *offset = (register_offset *)malloc(sizeof(
           register_offset));
        offset->reg = reg_identifier;
        offset->offset = offset_amount;
        tmp->reg_offset = offset;
        return tmp;
}

/* End Register specific operand creators */

operand *make_label_operand(char *label, ...) {
        va_list arglist;
        operand *tmp = new_operand();
        char *bigstring = malloc(sizeof(char) * 300);
        int i;
        va_start(arglist, label);
        vsprintf(bigstring, label, arglist);
        va_end(arglist);
        /* Copy label */
        tmp->label = malloc(sizeof(char) * (strlen(bigstring) + 1));
        strcpy(tmp->label, bigstring);
        free(bigstring);
        return tmp;
}

/* RUNTIME SUPPORT FNs */

/* Write header into source */
void write_preamble() {
        operand *comment;
        mips_instruction *ins;
        struct tm *local_time;
        time_t gen_time;
        gen_time = time(NULL);
        local_time = localtime(&gen_time);
```

```
        comment = make_label_operand("# Compiled from --C\n# %s\n.data\
            n\tEOL:\t.asciiz \"\\n\"\n.text\n", asctime(local_time));
        append_mips(mips_comment(comment, 0));
}

/* Write stub fn to call into the user code and return the value */
void write_epilogue() {
        append_mips(mips("", OT_ZERO_ADDRESS, OT_UNSET, OT_UNSET,
            make_label_operand(".text"), NULL, NULL, "", 1));
        append_mips(mips(".globl", OT_LABEL, OT_UNSET, OT_UNSET,
            make_label_operand("main"), NULL, NULL, "", 1));
        append_mips(mips("", OT_LABEL, OT_UNSET, OT_UNSET,
            make_label_operand("main"), NULL, NULL, "", 0));
        /* Pass zero dynamic and static links */
        append_mips(mips("move", OT_REGISTER, OT_REGISTER, OT_UNSET,
            make_register_operand($a1), make_register_operand($zero),
            NULL, "Zero dynamic link", 1));
        append_mips(mips("move", OT_REGISTER, OT_REGISTER, OT_UNSET,
            make_register_operand($v0), make_register_operand($zero),
            NULL, "Zero static link", 1));
        /* Call the _main fn */
        append_mips(mips("jal", OT_LABEL, OT_UNSET, OT_UNSET,
            make_label_operand("_main"), NULL, NULL, "", 1));
        /* Get hold of the return value */
        append_mips(mips("move", OT_REGISTER, OT_REGISTER, OT_UNSET,
            make_register_operand($a0), make_register_operand($v0), NULL
            , "Retrieve the return value of the main function", 1));
        /* Print the int */
        append_mips(mips("li", OT_REGISTER, OT_CONSTANT, OT_UNSET,
            make_register_operand($v0), make_constant_operand(1), NULL,
            "Print integer", 1));
        append_mips(syscall(""));
        /* Print an EOL character */
        append_mips(mips("li", OT_REGISTER, OT_CONSTANT, OT_UNSET,
            make_register_operand($v0), make_constant_operand(4), NULL,
            "Print string", 1));
        append_mips(mips("la", OT_REGISTER, OT_LABEL, OT_UNSET,
            make_register_operand($a0), make_label_operand("EOL"), NULL,
             "Printing EOL character", 1));
        append_mips(syscall(""));
        /* Sys exit */
        append_mips(mips("li", OT_REGISTER, OT_CONSTANT, OT_UNSET,
            make_register_operand($v0), make_constant_operand(10), NULL,
             "System exit", 1));
        append_mips(syscall(""));
}
```

```
/* Outputs fn that is used to build a new activation record, local_size
    passed in $a0 */
void write_activation_record_fn() {
        /* Make label */
        operand *comment;
        comment = make_label_operand("# Make a new activation record\n#
            Precondition: $a0 contains total required heap size, $a1
           contains dynamic link, $v0 contains static link\n# Returns:
           start of heap address in $v0, heap contains reference to
           static link and old $fp value");
        append_mips(mips_comment(comment, 0));
        append_mips(mips("", OT_LABEL, OT_UNSET, OT_UNSET,
           make_label_operand("mk_ar"), NULL, NULL, "", 0));
        append_mips(mips("move", OT_REGISTER, OT_REGISTER, OT_UNSET,
           make_register_operand($s1), make_constant_operand($v0), NULL
           , "Backup static link in $s1", 1));
        append_mips(mips("li", OT_REGISTER, OT_CONSTANT, OT_UNSET,
           make_register_operand($v0), make_constant_operand(9), NULL,
           "Allocate space systemcode", 1));
        append_mips(syscall("Allocate space on heap"));
        append_mips(mips("move", OT_REGISTER, OT_REGISTER, OT_UNSET,
           make_register_operand($s2), make_constant_operand($fp), NULL
           , "Backup old $fp in $s2", 1));
        /* Point $fp to the correct place */
        append_mips(mips("add", OT_REGISTER, OT_REGISTER, OT_REGISTER,
           make_register_operand($fp), make_register_operand($v0),
           make_register_operand($a0), "$fp = heap start address + heap
            size", 1));
        /* Save static link */
        append_mips(mips("sw", OT_REGISTER, OT_OFFSET, OT_UNSET,
           make_register_operand($s1), make_offset_operand($v0, 0),
           NULL, "Save static link", 1));
        /* Save old $fp */
        append_mips(mips("sw", OT_REGISTER, OT_OFFSET, OT_UNSET,
           make_register_operand($s2), make_offset_operand($v0, 4),
           NULL, "Save old $fp", 1));
        /* Save dynamic link */
        append_mips(mips("sw", OT_REGISTER, OT_OFFSET, OT_UNSET,
           make_register_operand($a1), make_offset_operand($v0, 8),
           NULL, "Save dynamic link", 1));
        /* Save frame size */
        append_mips(mips("sw", OT_REGISTER, OT_OFFSET, OT_UNSET,
           make_register_operand($a0), make_offset_operand($v0, 12),
           NULL, "Save framesize", 1));
        /* Jump back */
        append_mips(mips("jr", OT_REGISTER, OT_UNSET, OT_UNSET,
           make_register_operand($ra), NULL, NULL, "", 1));
```

```
}

/* Outputs fn that is used to register new function variables */
void write_register_fn_variable() {
        operand *comment;
        comment = make_label_operand("# Make a new function variable
           entry\n# Precondition: $v0 contains function address, $v1
           contains static link\n# Returns: address of allocated fn
           entry descriptor in $v0");
        append_mips(mips_comment(comment, 0));
        append_mips(mips("", OT_LABEL, OT_UNSET, OT_UNSET,
           make_label_operand("rfunc"), NULL, NULL, "", 0));
        append_mips(mips("move", OT_REGISTER, OT_REGISTER, OT_UNSET,
           make_register_operand($s1), make_constant_operand($v0), NULL
           , "Backup fn address in $s1", 1));
        append_mips(mips("li", OT_REGISTER, OT_CONSTANT, OT_UNSET,
           make_register_operand($a0), make_constant_operand(8), NULL,
           "Space required for descriptor", 1));
        append_mips(mips("li", OT_REGISTER, OT_CONSTANT, OT_UNSET,
           make_register_operand($v0), make_constant_operand(9), NULL,
           "Allocate space systemcode", 1));
        append_mips(syscall("Allocate space on heap"));
        append_mips(mips("sw", OT_REGISTER, OT_OFFSET, OT_UNSET,
           make_register_operand($s1), make_offset_operand($v0, 0),
           NULL, "Store fn address", 1));
        append_mips(mips("sw", OT_REGISTER, OT_OFFSET, OT_UNSET,
           make_register_operand($v1), make_offset_operand($v0, 4),
           NULL, "Store static link", 1));
        append_mips(mips("jr", OT_REGISTER, OT_UNSET, OT_UNSET,
           make_register_operand($ra), NULL, NULL, "", 1));
}

/* END RUNTIME SUPPORT FNs */

/* Get the friendly register name from the enum value */
char *register_name(enum sys_register reg_id) {
        /* Friendly names for registers */
        static char *rgs[] = {  "$zero",
                                                        "$at",
                                                        "$v0", "$v1",
                                                        "$a0", "$a1", "
                                                           $a2", "$a3",
                                                        "$t0", "$t1", "
                                                           $t2", "$t3",
                                                            "$t4", "$t5
                                                           ", "$t6", "
                                                           $t7",
```

```
                                                        "$s0", "$s1", "
                                                            $s2", "$s3",
                                                             "$s4", "$s5
                                                            ", "$s6", "
                                                            $s7",
                                                        "$t8", "$t9",
                                                        "$k0", "$k1",
                                                        "$gp",
                                                        "$sp",
                                                        "$fp",
                                                        "$ra"
                                                    };
        int register_names = sizeof(rgs) / sizeof(char *);
        if (reg_id < 0 || reg_id > register_names) return "";
        return rgs[reg_id];
}
```

mips.c

```
#include "mips.h"

/**
*       mips.c by Henry Thacker
*
*       Basic code generator to MIPS assembly language from TAC
    representation
*
*/

/* Append MIPS code to generated code stack */
void append_mips(struct mips_instruction *ins) {
        /* Are there any existing instructions? */
        if (!instructions) {
                instructions = ins;
        }
        else {
                struct mips_instruction *tmp_instruction = instructions
                    ;
                while (tmp_instruction->next != NULL) {
                        tmp_instruction = tmp_instruction->next;
                }
                /* Append instruction onto the end */
                tmp_instruction->next = ins;
        }
}

/* Find the depth difference between two different environments */
```

```
int depth_difference(environment *caller_env, environment *callee_env)
   {
        int depth = 0;
        environment *env = caller_env;
        if (callee_env == caller_env) return 0;
        while(env) {
                env = env->static_link;
                depth++;
                if (env == callee_env) {
                        return depth;
                }
        }
        return -1;
}

/* Valid register - can the USER utilise the given register? */
int register_use_allowed(int reg_id) {
        /* can use all $t0-$t9 registers */
        return (reg_id >= $t0 && reg_id <= $t7) || (reg_id == $t8 ||
           reg_id == $t9);
}

int is_argument_register(int reg_id) {
        return reg_id >= $a0 && reg_id <= $a3;
}

/* Is the value a constant? */
int is_constant(value *var) {
        return var && !var->temporary && var->value_type == VT_INTEGR
           && var->identifier[0]=='_';
}

/* Given a number of locals contained with a fn, return the overall
   byte size required in the associated activation record */
int activation_record_size(int local_size) {
        int word_size = 4;
        /* Special fields are: */
        /* - Previous $fp */
        /* - Static link */
        /* - Dynamic link */
        /* - Frame size */
        int special_fields = 4;
        int allocation_size = (word_size * local_size) + (word_size *
           special_fields);
        return allocation_size;
}
```

```
/* ==== VARIABLE FNS ==== */

/* Load a variable in local scope */
void cg_load_local_var(value *var, int destination_register) {
        int num = var->variable_number;
        append_mips(mips("lw", OT_REGISTER, OT_OFFSET, OT_UNSET,
            make_register_operand(destination_register),
            make_offset_operand($fp, -4 * (num + 1)), NULL, "Load local
            variable", 1));
}

/* Find a variable, load it into a register if not already in one,
   return the register ID */
int cg_find_variable(value *variable, environment *current_env, int
   should_attempt_load) {
        int reg_id;
        if (!variable || (!variable->stored_in_env && !is_constant(
            variable))) {
                fatal("Could not find variable %s!", correct_string_rep
                    (variable));
        }
        reg_id = already_in_reg(regs, variable, current_env, &
            has_used_fn_variable);
        if (reg_id == REG_VALUE_NOT_AVAILABLE) {
                reg_id = choose_best_reg(regs, current_env);
                regs[reg_id]->contents = variable;
                if (!should_attempt_load) return reg_id;
                if (is_constant(variable)) {
                        append_mips(mips("li", OT_REGISTER, OT_CONSTANT
                            , OT_UNSET, make_register_operand(reg_id),
                            make_constant_operand(to_int(NULL, variable)
                            ), NULL, "", 1));
                        return reg_id;
                }
                /* Support loading local variables */
                if (variable->stored_in_env->static_link == current_env
                    ) {
                        cg_load_local_var(variable, reg_id);
                }
                else {
                        int depth = (current_env->nested_level -
                            variable->stored_in_env->nested_level) + 1;
                        int i = 0;
                        int num = variable->variable_number;
                        int offset_reg = choose_best_reg(regs,
                            current_env);
```

```
                        append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                            OT_UNSET, make_register_operand(reg_id),
                            make_offset_operand($s0, 0), NULL, "Move up
                            a static link", 1));
                        for (i = 1; i < depth; i++) {
                                append_mips(mips("lw", OT_REGISTER,
                                    OT_OFFSET, OT_UNSET,
                                    make_register_operand(reg_id),
                                    make_offset_operand(reg_id, 0), NULL
                                    , "Move up a static link", 1));
                        }
                        append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                            OT_UNSET, make_register_operand(offset_reg),
                             make_offset_operand(reg_id, 12), NULL, "
                            Load framesize for static link", 1));
                        append_mips(mips("add", OT_REGISTER,
                            OT_REGISTER, OT_REGISTER,
                            make_register_operand(offset_reg),
                            make_register_operand(offset_reg),
                            make_register_operand(reg_id), "Seek to $fp
                            [end of AR]", 1));
                        append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                            OT_UNSET, make_register_operand(reg_id),
                            make_offset_operand(offset_reg, -4 * (num +
                            1)), NULL, correct_string_rep(variable), 1))
                            ;
                }
        }
        return reg_id;
}

/* Store a value in a given register */
void cg_store_in_reg(int reg, value *operand, environment *current_env)
    {
        if (is_constant(operand)) {
                append_mips(mips("li", OT_REGISTER, OT_CONSTANT,
                    OT_UNSET, make_register_operand(reg),
                    make_constant_operand(to_int(NULL, operand)), NULL,
                    "", 1));
        }
        else {
                int value_reg = cg_find_variable(operand, current_env,
                    1);
                /* Make the assignment */
                append_mips(mips("move", OT_REGISTER, OT_REGISTER,
                    OT_UNSET, make_register_operand(reg),
                    make_register_operand(value_reg), NULL, "Assign
```

```
                    values", 1));
        }
}

/* Either return the existing register that has been used to store a
   variable in, or insert into new register */
int get_register(value *variable, environment *current_env, int
   must_already_exist) {
        int reg_id = cg_find_variable(variable, current_env,
           must_already_exist);
        if (reg_id == REG_VALUE_NOT_AVAILABLE) {
                if (must_already_exist) {
                        fatal("Could not find variable %s",
                           correct_string_rep(variable));
                }
                reg_id = choose_best_reg(regs, current_env);
                regs[reg_id]->contents = variable;
        }
        return reg_id;
}

/* ==== CODE GENERATION ==== */


/* Generate code for an operation */
void cg_operation(int operation, value *op1, value *op2, value *result,
    environment *current_env) {
        int result_reg = get_register(result, current_env, 0);
        int op1_reg = -1;
        int op2_reg = -1;
        switch(operation) {
                case '+':
                        op1_reg = get_register(op1, current_env, 1);
                        if (is_constant(op2)) {
                                append_mips(mips("addi", OT_REGISTER,
                                   OT_REGISTER, OT_CONSTANT,
                                   make_register_operand(result_reg),
                                   make_register_operand(op1_reg),
                                   make_constant_operand(to_int(NULL,
                                   op2)), "", 1));
                        }
                        else {
                                op2_reg = get_register(op2, current_env
                                   , 1);
                                append_mips(mips("add", OT_REGISTER,
                                   OT_REGISTER, OT_REGISTER,
                                   make_register_operand(result_reg),
```

```
                        make_register_operand(op1_reg),
                        make_register_operand(op2_reg), "",
                        1));
                }
                break;
        case '-':
                op1_reg = get_register(op1, current_env, 1);
                if (is_constant(op2)) {
                        append_mips(mips("sub", OT_REGISTER,
                            OT_REGISTER, OT_CONSTANT,
                            make_register_operand(result_reg),
                            make_register_operand(op1_reg),
                            make_constant_operand(to_int(NULL,
                            op2)), "", 1));
                }
                else {
                        op2_reg = get_register(op2, current_env
                            , 1);
                        append_mips(mips("sub", OT_REGISTER,
                            OT_REGISTER, OT_REGISTER,
                            make_register_operand(result_reg),
                            make_register_operand(op1_reg),
                            make_register_operand(op2_reg), "",
                            1));
                }
                break;
        case '*':
                op1_reg = get_register(op1, current_env, 1);
                op2_reg = get_register(op2, current_env, 1);
                append_mips(mips("mult", OT_REGISTER,
                    OT_REGISTER, OT_UNSET, make_register_operand
                    (op1_reg), make_register_operand(op2_reg),
                    NULL, "", 1));
                append_mips(mips("mflo", OT_REGISTER, OT_UNSET,
                     OT_UNSET, make_register_operand(result_reg)
                    , NULL, NULL, "", 1));
                break;
        case '/':
                op1_reg = get_register(op1, current_env, 1);
                op2_reg = get_register(op2, current_env, 1);
                append_mips(mips("div", OT_REGISTER,
                    OT_REGISTER, OT_UNSET, make_register_operand
                    (op1_reg), make_register_operand(op2_reg),
                    NULL, "", 1));
                append_mips(mips("mflo", OT_REGISTER, OT_UNSET,
                     OT_UNSET, make_register_operand(result_reg)
                    , NULL, NULL, "", 1));
```

```
            break;
        case '%':
            op1_reg = get_register(op1, current_env, 1);
            op2_reg = get_register(op2, current_env, 1);
            append_mips(mips("div", OT_REGISTER,
               OT_REGISTER, OT_UNSET, make_register_operand
               (op1_reg), make_register_operand(op2_reg),
               NULL, "", 1));
            append_mips(mips("mfhi", OT_REGISTER, OT_UNSET,
                OT_UNSET, make_register_operand(result_reg)
                , NULL, NULL, "", 1));
            break;
        case '<':
            op1_reg = get_register(op1, current_env, 1);
            if (is_constant(op2)) {
                    append_mips(mips("slti", OT_REGISTER,
                       OT_REGISTER, OT_CONSTANT,
                        make_register_operand(result_reg),
                        make_register_operand(op1_reg),
                        make_constant_operand(to_int(NULL,
                        op2)), "$c = $a < b", 1));
            }
            else {
                    op2_reg = get_register(op2, current_env
                        , 1);
                    append_mips(mips("slti", OT_REGISTER,
                       OT_REGISTER, OT_REGISTER,
                        make_register_operand(result_reg),
                        make_register_operand(op1_reg),
                        make_register_operand(op2_reg), "$c
                        = $a < $b", 1));
            }
            break;
        case '>':
            op1_reg = get_register(op1, current_env, 1);
            op2_reg = get_register(op2, current_env, 1);
            append_mips(mips("sgt", OT_REGISTER,
               OT_REGISTER, OT_REGISTER,
               make_register_operand(result_reg),
               make_register_operand(op1_reg),
               make_register_operand(op2_reg), "$c = $a >
               $b", 1));
            break;
        case LE_OP:
            op1_reg = get_register(op1, current_env, 1);
            op2_reg = get_register(op2, current_env, 1);
```

```
                        append_mips(mips("sle", OT_REGISTER,
                            OT_REGISTER, OT_REGISTER,
                            make_register_operand(result_reg),
                            make_register_operand(op1_reg),
                            make_register_operand(op2_reg), "$c = $a <=
                            $b", 1));
                        break;
                case GE_OP:
                        op1_reg = get_register(op1, current_env, 1);
                        op2_reg = get_register(op2, current_env, 1);
                        append_mips(mips("sge", OT_REGISTER,
                            OT_REGISTER, OT_REGISTER,
                            make_register_operand(result_reg),
                            make_register_operand(op1_reg),
                            make_register_operand(op2_reg), "$c = $a >=
                            $b", 1));
                        break;
                case EQ_OP:
                        op1_reg = get_register(op1, current_env, 1);
                        op2_reg = get_register(op2, current_env, 1);
                        append_mips(mips("seq", OT_REGISTER,
                            OT_REGISTER, OT_REGISTER,
                            make_register_operand(result_reg),
                            make_register_operand(op1_reg),
                            make_register_operand(op2_reg), "$c = $a ==
                            $b", 1));
                        break;
                case NE_OP:
                        op1_reg = get_register(op1, current_env, 1);
                        op2_reg = get_register(op2, current_env, 1);
                        append_mips(mips("sne", OT_REGISTER,
                            OT_REGISTER, OT_REGISTER,
                            make_register_operand(result_reg),
                            make_register_operand(op1_reg),
                            make_register_operand(op2_reg), "$c = $a !=
                            $b", 1));
                        break;
                default:
                        fatal("Unrecognised operator!");
                        break;
        }
        /* Set modified flag */
        regs[result_reg]->modified = 1;
}

/* Code generate PUSHING a parameter */
void cg_push_param(value *operand, environment *current_env) {
```

```
        int reg_id = already_in_reg(regs, operand, current_env, &
            has_used_fn_variable);
        if (reg_id == REG_VALUE_NOT_AVAILABLE) {
                reg_id = $a0;
                cg_store_in_reg($a0, operand, current_env);
        }
        append_mips(mips("sub", OT_REGISTER, OT_REGISTER, OT_CONSTANT,
            make_register_operand($sp), make_register_operand($sp),
            make_constant_operand(4), "Move stack pointer", 1));
        append_mips(mips("sw", OT_REGISTER, OT_OFFSET, OT_UNSET,
            make_register_operand(reg_id), make_offset_operand($sp, 0),
            NULL, "Write param into stack", 1));
}

/* Code generate POPPING a parameter */
void cg_pop_param(value *operand) {
        int num = operand->variable_number;
        append_mips(mips("lw", OT_REGISTER, OT_OFFSET, OT_UNSET,
            make_register_operand($a0), make_offset_operand($sp, 0),
            NULL, "Pop the parameter", 1));
        append_mips(mips("sw", OT_REGISTER, OT_OFFSET, OT_UNSET,
            make_register_operand($a0), make_offset_operand($fp, -4 * (
            num + 1)), NULL, "Write param into heap", 1));
        append_mips(mips("add", OT_REGISTER, OT_REGISTER, OT_CONSTANT,
            make_register_operand($sp), make_register_operand($sp),
            make_constant_operand(4), "Move stack pointer", 1));
}

/* Code generate an assignment */
void cg_assign(value *result, value *operand1, environment *current_env
    ) {
        int result_reg = get_register(result, current_env, 1);
        cg_store_in_reg(result_reg, operand1, current_env);
        regs[result_reg]->modified = 1;
}

/* Code generate an IF statement */
void cg_if(value *condition, value *true_label, environment *
    current_env) {
        int condition_register = get_register(condition, current_env,
            1);
        append_mips(mips("bne", OT_REGISTER, OT_REGISTER, OT_LABEL,
            make_register_operand(condition_register),
            make_register_operand($zero), make_label_operand(
            correct_string_rep(true_label)), "", 1));
}
```

```
/* Code generate a fn call */
void cg_fn_call(value *result, value *fn_def, environment *current_env)
    {
        int result_reg = get_register(result, current_env, 0);
        regs[result_reg]->contents = result;
        regs[result_reg]->modified = 1;
        /* Pass dynamic link in $a1 */
        append_mips(mips("move", OT_REGISTER, OT_REGISTER, OT_UNSET,
            make_register_operand($a1), make_register_operand($s0), NULL
            , "Pass dynamic link", 1));
        append_mips(mips("jal", OT_LABEL, OT_UNSET, OT_UNSET,
            make_label_operand("_%s", correct_string_rep(fn_def)), NULL,
             NULL, "", 1));
        append_mips(mips("move", OT_REGISTER, OT_REGISTER, OT_UNSET,
            make_register_operand(result_reg), make_register_operand($v0
            ), NULL, "", 1));
}

/* Generate code to traverse a static link - used when calling a
   function */
void cg_load_static_link(value *caller, value *callee) {
        int i = 0;
        if (!caller || !callee) return;
        if (caller->stored_in_env == callee->stored_in_env) {
                /* Same level */
                append_mips(mips("lw", OT_REGISTER, OT_OFFSET, OT_UNSET
                    , make_register_operand($v0), make_offset_operand(
                    $s0, 0), NULL, "Point callee to same static link as
                    mine (caller)", 1));
        }
        else if (caller->stored_in_env == callee->stored_in_env->
            static_link) {
                /* Directly nested */
                append_mips(mips("move", OT_REGISTER, OT_REGISTER,
                    OT_UNSET, make_register_operand($v0),
                    make_register_operand($s0), NULL, "Set this current
                    activation record as the static link", 1));
        }
        else {
                /* Arbitrarily nested */
                int diff = depth_difference(caller->stored_in_env,
                    callee->stored_in_env);
                append_mips(mips("lw", OT_REGISTER, OT_OFFSET, OT_UNSET
                    , make_register_operand($v0), make_offset_operand(
                    $s0, 0), NULL, "Move up one static link", 1));
                for (i=0; i < diff; i++) {
```

```
                        append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                            OT_UNSET, make_register_operand($v0),
                            make_offset_operand($v0, 0), NULL, "Point
                            callee to same static link as mine (caller)"
                            , 1));
                }
        }
}

/* ==== TAC OPTIMISATION ==== */

/* Number TAC nesting levels */
void number_tac_levels(tac_quad *top) {
        int current_level = 0;
        while (top) {
                switch(top->type) {
                        case TT_BEGIN_FN:
                                current_level++;
                                top->level = current_level;
                                break;
                        case TT_END_FN:
                                top->level = current_level;
                                current_level--;
                                break;
                        default:
                                top->level = current_level;
                                break;
                }
                top = top->next;
        }
}

/* Bubblesort the linked list */
tac_quad *linked_sort(tac_quad *input) {
        tac_quad *current, *next, *previous, *tmp;
        int swaps;
        swaps = 1;
        while (swaps) {
                swaps = 0;
                current = input;
                next = current->next;
                previous = NULL;
                while (current && next && current!=next) {
                        if (current->level > next->level) {
                                swaps = 1;
                                /* Is this the head element? */
                                if (!previous) {
```

```
                                        tmp = next->next;
                                        next->next = current;
                                        current->next = tmp;
                                        /* Repoint head */
                                        input = next;
                                        previous = next;
                                }
                                else {
                                        tmp = next->next;
                                        next->next = current;
                                        current->next = tmp;
                                        previous->next = next;
                                        previous = next;
                                }
                        }
                        else {
                                /* Skip over elements - correct sorting
                                    already */
                                previous = current;
                                current = current->next;
                        }
                        next = current->next;
                }
        }
        /* Return the sorted version */
        return input;
}

/* ==== MAIN RECURSIVE LOOP ==== */


/* Main recursive code generation function */
void write_code(tac_quad *quad) {
        int depth_difference = 0;
        int size = 0;
        int temporary;
        int frame_size;
        if (quad==NULL) return;
        switch(quad->type) {
                case TT_FN_DEF:
                        break;
                case TT_INIT_FRAME:
                        size = to_int(NULL, quad->operand1);
                        size = activation_record_size(size);
                        frame_size = size;
                        /* Save return address in $s7 */
```

```
            append_mips(mips("move", OT_REGISTER,
                OT_REGISTER, OT_UNSET, make_register_operand
                ($s7), make_register_operand($ra), NULL, "
                Store Return address in $s7", 1));
            /* Make activation record of required size */
            append_mips(mips("li", OT_REGISTER, OT_CONSTANT
                , OT_UNSET, make_register_operand($a0),
                make_constant_operand(frame_size), NULL, "
                Store the frame size required for this AR",
                1));
            append_mips(mips("jal", OT_LABEL, OT_UNSET,
                OT_UNSET, make_label_operand("mk_ar"), NULL,
                 NULL, "", 1));
            /* Store a reference to activation record
                address in $s0 */
            append_mips(mips("move", OT_REGISTER,
                OT_REGISTER, OT_UNSET, make_register_operand
                ($s0), make_register_operand($v0), NULL, "
                Store heap start address in $s0", 1));
            break;
    case TT_FN_BODY:
            /* Save return address in stack */
            append_mips(mips("sub", OT_REGISTER,
                OT_REGISTER, OT_CONSTANT,
                make_register_operand($sp),
                make_register_operand($sp),
                make_constant_operand(4), "", 1));
            append_mips(mips("sw", OT_REGISTER, OT_OFFSET,
                OT_UNSET, make_register_operand($s7),
                make_offset_operand($sp, 0), NULL, "Save
                return address in stack", 1));
            break;
    case TT_BEGIN_FN:
            if (strcmp(correct_string_rep(quad->operand1),
                "main")==0) entry_point = quad;
            current_fn = quad->operand1;
            append_mips(mips("", OT_LABEL, OT_UNSET,
                OT_UNSET, make_label_operand("_%s",
                correct_string_rep(quad->operand1)), NULL,
                NULL, "", 0));
            break;
    case TT_GOTO:
            append_mips(mips("j", OT_LABEL, OT_UNSET,
                OT_UNSET, make_label_operand(
                correct_string_rep(quad->operand1)), NULL,
                NULL, "", 1));
            break;
```

```
case TT_POP_PARAM:
        cg_pop_param(quad->operand1);
        break;
case TT_LABEL:
        append_mips(mips("", OT_LABEL, OT_UNSET,
           OT_UNSET, make_label_operand(
           correct_string_rep(quad->operand1)), NULL,
           NULL, "", 0));
        break;
case TT_ASSIGN:
        cg_assign(quad->result, quad->operand1,
           current_fn->stored_in_env);
        break;
case TT_PUSH_PARAM:
        cg_push_param(quad->operand1, current_fn->
           stored_in_env);
        break;
case TT_PREPARE:
        break;
case TT_IF:
        cg_if(quad->operand1, quad->result, current_fn
           ->stored_in_env);
        break;
case TT_OP:
        cg_operation(quad->subtype, quad->operand1,
           quad->operand2, quad->result, current_fn->
           stored_in_env);
        break;
case TT_FN_CALL:
        /* Wire out live registers into memory, in-case
            they're overwritten */
        save_t_regs(regs, current_fn->stored_in_env);
        clear_regs(regs);
        if (!quad->operand1->data.func || !quad->
           operand1->data.func->node_value) {
              /* Dealing with fn variable - we can
                 deduce its entry point & static link
                  from */
              /* runtime stored information */
              int fn_variable = get_register(quad->
                 operand1, current_fn->stored_in_env,
                  1);
              int result_reg = get_register(quad->
                 result, current_fn->stored_in_env,
                 0);
              int address_reg = choose_best_reg(regs,
                  current_fn->stored_in_env);
```

```
                append_mips(mips("lw", OT_REGISTER,
                    OT_OFFSET, OT_UNSET,
                    make_register_operand(address_reg),
                    make_offset_operand(fn_variable, 0),
                     NULL, "Get Fn address", 1));
                append_mips(mips("lw", OT_REGISTER,
                    OT_OFFSET, OT_UNSET,
                    make_register_operand($v0),
                    make_offset_operand(fn_variable, 4),
                     NULL, "Get static link", 1));
                regs[result_reg]->contents = quad->
                    result;
                regs[result_reg]->modified = 1;
                /* Pass dynamic link in $a1 */
                append_mips(mips("move", OT_REGISTER,
                    OT_REGISTER, OT_UNSET,
                    make_register_operand($a1),
                    make_register_operand($s0), NULL, "
                    Pass dynamic link", 1));
                append_mips(mips("jalr", OT_REGISTER,
                    OT_UNSET, OT_UNSET,
                    make_register_operand(address_reg),
                    NULL, NULL, "", 1));
                append_mips(mips("move", OT_REGISTER,
                    OT_REGISTER, OT_UNSET,
                    make_register_operand(result_reg),
                    make_register_operand($v0), NULL, ""
                    , 1));
            }
            else {
                /* Work out what static link to pass */
                cg_load_static_link(current_fn, quad->
                    operand1);
                cg_fn_call(quad->result, quad->operand1
                    , current_fn->stored_in_env);
            }
            break;
        case TT_END_FN:
            /* Save regs */
            save_t_regs(regs, current_fn->stored_in_env);
            clear_regs(regs);
            /* Load return address from stack */
            append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                OT_UNSET, make_register_operand($ra),
                make_offset_operand($sp, 0), NULL, "Get
                return address", 1));
```

```
            append_mips(mips("add", OT_REGISTER,
               OT_REGISTER, OT_CONSTANT,
               make_register_operand($sp),
               make_register_operand($sp),
               make_constant_operand(4), "Pop return
               address from stack", 1));
            append_mips(mips("move", OT_REGISTER,
               OT_REGISTER, OT_UNSET, make_register_operand
               ($v0), make_register_operand($zero), NULL, "
               Null return value", 1));
            /* Load previous frame pointer */
            append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
               OT_UNSET, make_register_operand($fp),
               make_offset_operand($s0, 4), NULL, "Load
               previous frame ptr", 1));
            /* Load previous static link */
            append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
               OT_UNSET, make_register_operand($s0),
               make_offset_operand($s0, 8), NULL, "Load
               dynamic link", 1));
            append_mips(mips("jr", OT_REGISTER, OT_UNSET,
               OT_UNSET, make_register_operand($ra), NULL,
               NULL, "Jump to $ra", 1));
            break;
      case TT_RETURN:
            /* Save the return value */
            /* Restore the activation record */
            if (quad->operand1) {
                  if (quad->operand1->value_type==
                     VT_FUNCTN) {
                        /* Return address to function
                           */
                        append_mips(mips("la",
                           OT_REGISTER, OT_LABEL,
                           OT_UNSET,
                           make_register_operand($v0),
                           make_label_operand("_%s",
                           correct_string_rep(quad->
                           operand1)), NULL, "Store
                           address of function", 1));
                        append_mips(mips("move",
                           OT_REGISTER, OT_REGISTER,
                           OT_UNSET,
                           make_register_operand($v1),
                           make_register_operand($s0),
                           NULL, "Store static link to
                           call with", 1));
```

```
                        append_mips(mips("jal",
                            OT_LABEL, OT_UNSET, OT_UNSET
                            , make_label_operand("rfunc"
                            ), NULL, NULL, "Register fn
                            variable", 1));
                        /* $v0 now contains fn
                            descriptor, can be used to
                            execute the function in the
                            right way */
                        has_used_fn_variable = 1;
                    }
                    else {
                        cg_store_in_reg($v0, quad->
                            operand1, current_fn->
                            stored_in_env);
                    }
                }
                /* Save regs */
                save_t_regs(regs, current_fn->stored_in_env);
                clear_regs(regs);
                /* Load return address from stack */
                append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                    OT_UNSET, make_register_operand($ra),
                    make_offset_operand($sp, 0), NULL, "Get
                    return address", 1));
                append_mips(mips("add", OT_REGISTER,
                    OT_REGISTER, OT_CONSTANT,
                    make_register_operand($sp),
                    make_register_operand($sp),
                    make_constant_operand(4), "Pop return
                    address from stack", 1));
                /* Load previous frame pointer */
                append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                    OT_UNSET, make_register_operand($fp),
                    make_offset_operand($s0, 4), NULL, "Load
                    previous frame ptr", 1));
                /* Load previous static link */
                append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                    OT_UNSET, make_register_operand($s0),
                    make_offset_operand($s0, 8), NULL, "Load
                    dynamic link", 1));
                append_mips(mips("jr", OT_REGISTER, OT_UNSET,
                    OT_UNSET, make_register_operand($ra), NULL,
                    NULL, "Jump to $ra", 1));
                break;
            default:
                fatal("Unrecognised TAC quad");
```

```
                        break;
        }
        write_code(quad->next);
}

/* ==== START POINT FOR TRANSLATION ==== */

/* Generate MIPS code for given tree */
void code_gen(NODE *tree) {
        tac_quad *quad = NULL;
        if (!tree) {
                fatal("Invalid input");
        }
        has_used_fn_variable = 0;
        regs = (register_contents **) malloc(sizeof(register_contents
           *) * REG_COUNT);
        init_register_view(regs);
        quad = start_tac_gen(tree);
        /* Identify nesting within TAC elements */
        number_tac_levels(quad);
        /* Bubble sort according to level */
        quad = linked_sort(quad);
        /* Write code header */
        write_preamble();
        current_fn = NULL;
        write_code(quad);
        write_activation_record_fn();
        if (has_used_fn_variable) write_register_fn_variable();
        write_epilogue();
        /* Perform basic optimisation */
        instructions = do_optimise(instructions);
        print_mips(instructions);
}
```

optimisation.c

```
#include "optimisation.h"

/**
*       optimisation.c by Henry Thacker
*
*       Basic code optimiser for MIPS code generation
*
*/

/* Should I continue removing nodes after a return statement, given the
    contents of the current node? */
int continue_removing_nodes(mips_instruction *current) {
```

```
        return !(current && current->operand1 && (current->
            operand1_type == OT_COMMENT || current->operand1_type ==
            OT_FN_LABEL || current->operand1_type == OT_LABEL));
}

/* Remove all statements after a return (JR) statement up until a label
    , function label or comment */
/* Rationale: Statements after jumps will never be executed */
mips_instruction *remove_after_return(mips_instruction *input) {
        mips_instruction *instruction;
        mips_instruction *previous_instruction;
        int deleting = 0;
        previous_instruction = NULL;
        instruction = input;
        while (instruction) {
                if (previous_instruction && deleting &&
                    continue_removing_nodes(instruction)) {
                        previous_instruction->next = instruction->next;
                        /* We don't need this instruction any more */
                        instruction = NULL;
                        free(instruction);
                        instruction = previous_instruction->next;
                }
                else {
                        /* Do we need to delete after this node? i.e.
                            jump return, or jump */
                        deleting = strcmp(instruction->operation, "jr")
                            ==0 || strcmp(instruction->operation, "j")
                            ==0;
                        previous_instruction = instruction;
                        instruction = instruction->next;
                }
        }
        return input;
}

/* Due to the way the code generation happens, often one can find the
    pattern move $X, $v0 followed directly by move $v0, $X */
/* Remove the redundant second line, the first is enough */
/* Rationale: remove redundant code */
mips_instruction *remove_redundant_move(mips_instruction *input) {
        mips_instruction *c_ins;
        mips_instruction *p_ins;
        p_ins = NULL;
        c_ins = input;
        while (c_ins) {
                if (c_ins && p_ins) {
```

```
                        int prev;
                        int curr;
                        prev = strcmp(p_ins->operation, "move")==0 &&
                           p_ins->operand2 && p_ins->operand2_type ==
                           OT_REGISTER && p_ins->operand2->reg == $v0;
                        curr = strcmp(c_ins->operation, "move")==0 &&
                           c_ins->operand1 && c_ins->operand1_type ==
                           OT_REGISTER && c_ins->operand1->reg == $v0;
                        if (prev && curr && p_ins->operand2->reg ==
                           c_ins->operand1->reg) {
                                /* Target couple of instructions found
                                   */
                                /* Circumvent the current instruction
                                   */
                                p_ins->next = c_ins->next;
                                free(c_ins);
                                c_ins = p_ins->next;
                                p_ins = c_ins->next;
                                continue;
                        }
                }
                p_ins = c_ins;
                c_ins = c_ins->next;
        }
        return input;
}

/* Due to the way the code generation happens, often one can find the
   pattern add $sp, $sp, 4 followed directly by sub $sp, $sp, 4 */
/* Remove both lines, the effect of both operations in tandem cancel
   each other out */
/* Rationale: remove redundant code */
mips_instruction *remove_redundant_sp_move(mips_instruction *input) {
        mips_instruction *c_ins, *p_ins, *n_ins;
        p_ins = NULL;
        c_ins = input;
        n_ins = c_ins->next;
        while (c_ins) {
                if (c_ins && p_ins && n_ins) {
                        int condition1, condition2;
                        condition1 = strcmp(c_ins->operation, "add")==0
                            && c_ins->operand1 && c_ins->operand1_type
                           == OT_REGISTER && c_ins->operand1->reg ==
                           $sp
                                                        && c_ins->
                                                           operand2 &&
                                                           c_ins->
```

```
                                                operand2_type
                                                 ==
                                                OT_REGISTER
                                                && c_ins->
                                                operand1->
                                                reg == $sp
                                                && c_ins->
                                                operand3
                                            && c_ins->
                                                operand3_type
                                                 ==
                                                OT_CONSTANT
                                                && c_ins->
                                                operand3->
                                                constant ==
                                                4;
        condition2 = strcmp(n_ins->operation, "sub")==0
            && n_ins->operand1 && n_ins->operand1_type
           == OT_REGISTER && n_ins->operand1->reg ==
           $sp
                                            && n_ins->
                                                operand2 &&
                                                n_ins->
                                                operand2_type
                                                 ==
                                                OT_REGISTER
                                                && n_ins->
                                                operand1->
                                                reg == $sp
                                                && n_ins->
                                                operand3
                                            && n_ins->
                                                operand3_type
                                                 ==
                                                OT_CONSTANT
                                                && n_ins->
                                                operand3->
                                                constant ==
                                                4;
        if (condition1 && condition2) {
                /* Target couple of instructions found
                   */
                /* Circumvent the current instructions
                   */
                free(c_ins);
                p_ins->next = n_ins->next;
                free(n_ins);
```

```
                                p_ins = p_ins->next;
                                c_ins = p_ins->next;
                                continue;
                        }
                }
                p_ins = c_ins;
                c_ins = c_ins->next;
                if (c_ins) n_ins = c_ins->next;
        }
        return input;
}

/* Perform optimisation functions */
mips_instruction *do_optimise(mips_instruction *input) {
        input = remove_after_return(input);
        input = remove_redundant_move(input);
        input = remove_redundant_sp_move(input);
        return input;
}
```

registers.c

```
#include "registers.h"

/**
*       registers.c by Henry Thacker
*
*       Utilities to work with state of MIPS registers while code
    generation takes place
*
*/

/* Initialise our global view of register allocation */
void init_register_view(register_contents **regs) {
        int i;
        for (i = 0; i < REG_COUNT; i++) {
                regs[i] = (register_contents *) malloc(sizeof(
                    register_contents));
                regs[i]->contents = NULL;
                regs[i]->accesses = 0;
                regs[i]->assignment_id = 0;
                regs[i]->modified = 0;
        }
}

/* Print the view of the registers */
void print_register_view(register_contents **regs) {
        int i;
```

```
        if (!DEBUG_ON) return;
        printf("Register View\n-----------------\n");
        for (i = 0; i < REG_COUNT; i++) {
                if (register_use_allowed(i) || is_argument_register(i))
                    {
                        printf("[%s] \t- Contents: [%p - %s] - Modified
                            : %d - Accesses: %d - Assignment order: %d\n
                            ", register_name(i), regs[i]->contents, regs
                            [i]->contents ? regs[i]->contents->
                            identifier : "EMPTY", regs[i]->modified,
                            regs[i]->accesses, regs[i]->assignment_id);
                }
        }
}

/* Clear registers */
void clear_regs(register_contents **regs) {
        int i = 0;
        for (i = 0; i < REG_COUNT; i++) {
                /* Do not save constants and only save back modified
                    values */
                if (regs[i]->contents) {
                        regs[i]->contents = NULL;
                        regs[i]->accesses = 0;
                        regs[i]->assignment_id = 0;
                        /* Modified value saved */
                        regs[i]->modified = 0;
                }
        }
}

/* Save a specific $t reg, if applicable */
void save_t_reg(register_contents **regs, int i, environment *
    current_env) {
        /* Do not save constants and only save back modified values */
        if (regs[i]->contents && regs[i]->modified && regs[i]->contents
            ->stored_in_env) {
                if (regs[i]->contents->stored_in_env->static_link ==
                    current_env) {
                        append_mips(mips("sw", OT_REGISTER, OT_OFFSET,
                            OT_UNSET, make_register_operand(i),
                            make_offset_operand($fp, -4 * (regs[i]->
                            contents->variable_number + 1)), NULL, "
                            Write out used local variable", 1));
                }
                else {
                        value *variable = regs[i]->contents;
```

```
                        int reg_id = choose_best_reg(regs, current_env)
                            ;
                        int depth = (current_env->nested_level -
                            variable->stored_in_env->nested_level) + 1;
                        int x = 0;
                        int num = variable->variable_number;
                        append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                            OT_UNSET, make_register_operand(reg_id),
                            make_offset_operand($s0, 0), NULL, "Move up
                            a static link", 1));
                        for (x = 1; x < depth; x++) {
                                append_mips(mips("lw", OT_REGISTER,
                                    OT_OFFSET, OT_UNSET,
                                    make_register_operand(reg_id),
                                    make_offset_operand(reg_id, 0), NULL
                                    , "Move up a static link", 1));
                        }
                        append_mips(mips("lw", OT_REGISTER, OT_OFFSET,
                            OT_UNSET, make_register_operand($s6),
                            make_offset_operand(reg_id, 12), NULL, "Load
                             framesize for static link", 1));
                        append_mips(mips("add", OT_REGISTER,
                            OT_REGISTER, OT_REGISTER,
                            make_register_operand($s6),
                            make_register_operand($s6),
                            make_register_operand(reg_id), "Seek to $fp
                            [end of AR]", 1));
                        append_mips(mips("sw", OT_REGISTER, OT_OFFSET,
                            OT_UNSET, make_register_operand(i),
                            make_offset_operand($s6, -4 * (num + 1)),
                            NULL, "Save distant modified variable", 1));
                }
                /* Modified value saved */
                regs[i]->modified = 0;
        }
}

/* Save pertinent $t regs before a fn call in case they get overwritten
    */
void save_t_regs(register_contents **regs, environment *current_env) {
        int i = 0;
        for (i = 0; i < REG_COUNT; i++) {
                save_t_reg(regs, i, current_env);
        }
}

/* Find the first free register, if any */
```

```
int first_free_reg(register_contents **regs) {
        int position = 0;
        for (position = 0; position < REG_COUNT; position++) {
                if (register_use_allowed(position) && !regs[position]->
                   contents) {
                        regs[position]->assignment_id = ++
                           regs_assignments;
                        regs[position]->contents = NULL;
                        regs[position]->accesses = 1;
                        regs[position]->modified = 0;
                        return position;
                }
        }
        return REG_NONE_FREE;
}

/* If no registers free, replace the oldest value */
int choose_best_reg(register_contents **regs, environment *env) {

        int free_reg = first_free_reg(regs);
        if (free_reg == REG_NONE_FREE) {
                int position = 0;
                int lowest_assignment_order = -1;
                int optimal_reg = -1;
                for (position = 0; position < REG_COUNT; position++) {
                        if (register_use_allowed(position)) {
                                if (lowest_assignment_order == -1 ||
                                   regs[position]->assignment_id <
                                   lowest_assignment_order) {
                                        lowest_assignment_order = regs[
                                           position]->assignment_id;
                                        optimal_reg = position;
                                }
                        }
                }
                /* Move out existing value that is being spilled */
                if (regs[optimal_reg]->contents->stored_in_env) {
                        save_t_reg(regs, optimal_reg, env);
                }
                regs[optimal_reg]->assignment_id = ++regs_assignments;
                regs[optimal_reg]->contents = NULL;
                regs[optimal_reg]->accesses = 1;
                regs[optimal_reg]->modified = 0;
                return optimal_reg;
        }
        return free_reg;
}
```

```
/* Is the value already in a register?? */
int already_in_reg(register_contents **regs, value *var, environment *
    env, int *has_used_fn_variable) {
        int position = 0;
        /* Check if this is a well known function, if so, pass back the
            address of the label in a register */
        if (var->value_type == VT_FUNCTN && var->data.func && var->data
            .func->node_value) {
                position = choose_best_reg(regs, env);
                /* Return address to function */
                append_mips(mips("la", OT_REGISTER, OT_LABEL, OT_UNSET,
                    make_register_operand($v0), make_label_operand("_%s
                    ", correct_string_rep(var)), NULL, "Store address of
                    function", 1));
                append_mips(mips("move", OT_REGISTER, OT_REGISTER,
                    OT_UNSET, make_register_operand($v1),
                    make_register_operand($s0), NULL, "Store static link
                    to call with", 1));
                append_mips(mips("jal", OT_LABEL, OT_UNSET, OT_UNSET,
                    make_label_operand("rfunc"), NULL, NULL, "Register
                    fn variable", 1));
                /* $v0 now contains fn descriptor, can be used to
                    execute the function in the right way */
                append_mips(mips("move", OT_REGISTER, OT_REGISTER,
                    OT_UNSET, make_register_operand(position),
                    make_register_operand($v0), NULL, "Return fn
                    descriptor address", 1));
                *has_used_fn_variable = 1;
                return position;
        }
        /* Machine dependent optimization - use the zero register where
            possible */
        if (is_constant(var) && to_int(NULL, var) == 0) return $zero;

        for (position = 0; position < REG_COUNT; position++) {
                /* Point to same variable OR are equivalent constants
                    */
                /* Must be sourced from a user accessible register, or
                    the special zero register */
                if (register_use_allowed(position) || position == 0 ||
                    is_argument_register(position)) {
                        if (regs[position]->contents == var) {
                                /* Increment accesses to track how
                                    popular this register is */
                                regs[position]->accesses = regs[
                                    position]->accesses + 1;
```

```
                                        regs[position]->assignment_id = ++
                                            regs_assignments;
                                        return position;
                                }
                        }
                }
        return REG_VALUE_NOT_AVAILABLE;
}
```

8.1.4 tacgen folder

tacgenerator.c

```
#include "tacgenerator.h"

/**
*       tacgenerator.c by Henry Thacker
*
*       Basic Three Address Code generator. TAC is generated directly
    from parsed AST.
*    The generated TAC is used to generate target code for MIPS.
*
*/

/* Register temporary in the specified environment */
value *register_temporary(environment *env, char *temp_name, value *
    null_value) {
        value *reference = NULL;
        if (!null_value) {
                reference = store(env, VT_VOID, temp_name, NULL, 0, 1,
                    0, 1);
        }
        else {
                reference = store(env, null_value->value_type,
                    temp_name, null_value, 0, 1, 0, 1);
        }
        assert(reference!=NULL, "Could not register temporary");
        return reference;
}

/* Generate a new temporary */
value *generate_temporary(environment *env, value *null_value) {
        char *tmp;
        static int temp_count = 0;
        tmp = malloc(sizeof(char) * 15);
        sprintf(tmp, "_t%d", ++temp_count);
        return register_temporary(env, tmp, null_value);
}
```

```
value *generate_untypechecked_temporary(environment *env) {
        return generate_temporary(env, untyped_value());
}

/* TAC to stdout */
void print_tac(tac_quad *quad) {
        if (quad==NULL) return;
        switch(quad->type) {
                case TT_LABEL:
                        printf("%s:\n", to_string(quad->operand1));
                        break;
                case TT_FN_DEF:
                        printf("_%s:\n", to_string(quad->operand1));
                        break;
                case TT_FN_CALL:
                        printf("%s = CallFn _%s\n", correct_string_rep(
                           quad->result), correct_string_rep(quad->
                           operand1));
                        break;
                case TT_INIT_FRAME:
                        printf("InitFrame %s\n", correct_string_rep(
                           quad->operand1));
                        break;
                case TT_POP_PARAM:
                printf("PopParam %s\n", quad->operand1->identifier);
                        break;
                case TT_PUSH_PARAM:
                        printf("PushParam %s\n", correct_string_rep(
                           quad->operand1));
                        break;
                case TT_IF:
                        printf("If %s Goto %s\n", correct_string_rep(
                           quad->operand1), correct_string_rep(quad->
                           result));
                        break;
                case TT_ASSIGN:
                        printf("%s %s %s\n", correct_string_rep(quad->
                           result), quad->op, correct_string_rep(quad->
                           operand1));
                        break;
                case TT_GOTO:
                        printf("Goto %s\n", correct_string_rep(quad->
                           operand1));
                        break;
                case TT_OP:
                        printf("%s = %s %s %s\n", correct_string_rep(
                           quad->result), correct_string_rep(quad->
```

```
                        operand1), quad->op, correct_string_rep(quad
                        ->operand2));
                    break;
                case TT_RETURN:
                    if (quad->operand1) {
                        printf("Return %s\n",
                            correct_string_rep(quad->operand1));
                    }
                    else {
                        printf("Return");
                    }
                    break;
                case TT_PREPARE:
                    printf("PrepareToCall %d\n", param_count(quad->
                        operand1));
                    break;
                case TT_BEGIN_FN:
                    printf("BeginFn %s\n", correct_string_rep(quad
                        ->operand1));
                    break;
                case TT_FN_BODY:
                    printf("FnBody\n");
                    break;
                case TT_END_FN:
                    printf("EndFn\n");
                    break;
                default:
                    fatal("Unknown TAC Quad type '%d'", quad->type)
                        ;
                    break;
        }
        print_tac(quad->next);
}

/* Add TAC quad onto end of generated code */
void append_code(tac_quad *quad) {
        if (tac_output == NULL) {
                tac_output = quad;
        }
        else {
                tac_quad *tmp = tac_output;
                while(1) {
                        if (tmp->next==NULL) break;
                        tmp = tmp->next;
                }
                tmp->next = quad;
        }
```

```
}

/* Make a TAC quad */
tac_quad *make_quad_value(char *op, value *operand1, value *operand2,
    value *result, int type, int subtype) {
        tac_quad *tmp_quad = (tac_quad *)malloc(sizeof(tac_quad));
        tmp_quad->op = (char *)malloc(sizeof(char) * (strlen(op) + 1));
        tmp_quad->operand1 = operand1;
        tmp_quad->operand2 = operand2;
        tmp_quad->result = result;
        tmp_quad->type = type;
        tmp_quad->subtype = subtype;
        strcpy(tmp_quad->op, op);
        return tmp_quad;
}

/* Convert operator tokens into TAC operator string equivalents */
char *type_to_string(int type) {
        char *tmp_type;
        switch(type) {
                case NE_OP:
                        return "!=";
                case EQ_OP:
                        return "==";
                case LE_OP:
                        return "<=";
                case GE_OP:
                        return ">=";
                default:
                        tmp_type = malloc(sizeof(char) * 3);
                        sprintf(tmp_type, "%c", type);
                        return tmp_type;
        }
}

/* Build the correct code in the correct place for the else part */
void build_else_part(environment *env, NODE *node, int true_part, int
    flag, int return_type) {
        if (node==NULL || type_of(node)!=ELSE) return;
        if (true_part) {
                make_simple(env, node->left, flag, return_type);
        }
        else {
                make_simple(env, node->right, flag, return_type);
        }
}
```

```
/* Generate jump label with given name */
tac_quad *prepare_fn(value *func) {
        return make_quad_value("", func, NULL, NULL, TT_PREPARE, 0);
}

/* Generate jump label with given name */
tac_quad *make_label(char *label_name) {
        return make_quad_value("", string_value(label_name), NULL, NULL
            , TT_LABEL, 0);
}

/* Generate IF statement from given condition and true jump label */
tac_quad *make_if(value *condition, char *true_label) {
        return make_quad_value("", condition, NULL, string_value(
            true_label), TT_IF, 0);
}

/* Generate IF statement from given condition and true jump label */
tac_quad *make_goto(char *label_name) {
        return make_quad_value("", string_value(label_name), NULL, NULL
            , TT_GOTO, 0);
}

/* Generate RETURN statement with given return value */
tac_quad *make_return(value *return_value) {
        return make_quad_value("", return_value, NULL, NULL, TT_RETURN,
             0);
}

/* Generate an END_FN statement */
tac_quad *make_end_fn(value *fn_def) {
        return make_quad_value("", fn_def, NULL, NULL, TT_END_FN, 0);
}

/* Generate an INIT_FRAME statement */
tac_quad *make_init_frame() {
        return make_quad_value("", int_value(0), NULL, NULL,
            TT_INIT_FRAME, 0);
}

/* Generate an BEGIN_FN statement */
tac_quad *make_begin_fn(value *fn_def) {
        return make_quad_value("", fn_def, NULL, NULL, TT_BEGIN_FN, 0);
}

/* Generate FN Definition label */
tac_quad *make_fn_def(value *fn_def) {
```

```
        return make_quad_value("", string_value(fn_def->identifier),
           NULL, NULL, TT_FN_DEF, 0);
}

/* Generate FN call */
tac_quad *make_fn_call(value *result, value *fn_def) {
        return make_quad_value("", fn_def, NULL, result, TT_FN_CALL, 0)
           ;
}

/* Generate FN body */
tac_quad *make_fn_body(value *fn_def) {
        return make_quad_value("", fn_def, NULL, NULL, TT_FN_BODY, 0);
}

/* Build necessary code for an if statement */
void build_if_stmt(environment *env, NODE *node, int if_count, tac_quad
    *false_jump, tac_quad *loop_jump, int flag, int return_type) {
        char *s_tmp;
        value *val1, *val2, *temporary;
        if (node==NULL || (type_of(node)!=IF && type_of(node)!=WHILE))
           return;
        /* LHS is condition */
        val1 = make_simple(env, node->left, flag, return_type);

        /* Generate if statement */
        s_tmp = malloc(sizeof(char) * 25);
        sprintf(s_tmp, "__if%dtrue", if_count);
        append_code(make_if(val1, s_tmp));

        /* Output false branch (i.e. else part) */
        if (type_of(node->right)==ELSE) {
                /* Build code for false part */
                build_else_part(env, node->right, 0, flag, return_type)
                   ;
        }

        /* Generate goto end of if statement */
        if (false_jump != NULL) {
                append_code(false_jump);
        }
        else {
                s_tmp = malloc(sizeof(char) * 25);
                sprintf(s_tmp, "__if%dend", if_count);
                append_code(make_goto(s_tmp));
        }
```

```
        /* Generate label for start of true branch */
        s_tmp = malloc(sizeof(char) * 25);
        sprintf(s_tmp, "__if%dtrue", if_count);
        append_code(make_label(s_tmp));

        /* Output true branch */
        if (type_of(node->right)==ELSE) {
                /* Build code for true part */
                build_else_part(env, node->right, 1, flag, return_type)
                    ;
        }
        else {
                /* True part is whole right branch */
                make_simple(env, node->right, flag, return_type);
        }

        /* Check if extra loop jump has been specified (for WHILE loops
            etc) */
        if (loop_jump) {
                append_code(loop_jump);
        }

        /* Generate end of IF stmt label */
        s_tmp = malloc(sizeof(char) * 25);
        sprintf(s_tmp, "__if%dend", if_count);
        append_code(make_label(s_tmp));
}

/* Build necessary code for a while statement */
void build_while_stmt(environment *env, NODE *node, int while_count,
    int if_count, int flag, int return_type) {
        char *s_tmp, *val1, *val2, *temporary;
        tac_quad *loop_jmp;
        if (node==NULL || type_of(node)!=WHILE) return;

        /* Generate label for start of while loop */
        s_tmp = malloc(sizeof(char) * 25);
        sprintf(s_tmp, "__while%d", while_count);
        append_code(make_label(s_tmp));

        /* Generate loop jump */
        s_tmp = malloc(sizeof(char) * 25);
        sprintf(s_tmp, "__while%d", while_count);
        loop_jmp = make_goto(s_tmp);

        /* End while loop stmt */
        s_tmp = malloc(sizeof(char) * 25);
```

```
        sprintf(s_tmp, "__while%dend", while_count);

        /* Build IF stmt for condition */
        build_if_stmt(env, node, if_count, make_goto(s_tmp), loop_jmp,
            flag, return_type);

        append_code(make_label(s_tmp));

}

/* Register params in environment */
void register_params(environment *env, value *param_list) {
        value *current_param;
        if (!param_list) return;
        current_param = param_list;
        while (current_param!=NULL) {
                value *param = NULL;
                value *param_name;
                param_name = string_value(current_param->identifier);
                switch(current_param->value_type) {
                        case VT_INTEGR:
                                param = assign(env, param_name,
                                    int_value(0), 1);
                                break;
                        case VT_VOID:
                                param = assign(env, param_name,
                                    void_value(), 1);
                                break;
                        case VT_FUNCTN:
                                param = assign(env, param_name, null_fn
                                    , 1);
                                break;
                        default:
                                fatal("Could not determine parameter
                                    type!");
                }
                append_code(make_quad_value("", param, NULL, NULL,
                    TT_POP_PARAM, 0));
                current_param = current_param->next;
        }
}

/* Push params on param stack in reverse order (recursively) */
tac_quad *push_params(environment *env, value *params_head) {
        if (!params_head) return NULL;
        if (params_head->next) {
                append_code(push_params(env, params_head->next));
```

```
        }
        if (params_head->value_type == VT_STRING) {
                value *tmp = get(env, correct_string_rep(params_head));
                return make_quad_value("", tmp, NULL, NULL,
                   TT_PUSH_PARAM, 0);
        }
        return make_quad_value("", params_head, NULL, NULL,
           TT_PUSH_PARAM, 0);
}

/* Declare variables underneath a declarator tree */
void declare_variables_tac(environment *env, NODE *node, int
   variable_type, int return_type) {
        value *variable_name = NULL;
        if (env == NULL || node == NULL) {
                return;
        }
        else if (type_of(node) == ',') {
                declare_variables_tac(env, node->left, variable_type,
                   return_type);
                declare_variables_tac(env, node->right, variable_type,
                   return_type);
                return;
        }
        else if (type_of(node) == '=') { /* Specific assignment */
                variable_name = make_simple(env, node->left, 0,
                   return_type);
        }
        else if (type_of(node) == LEAF) { /* Undefined assignment */
                variable_name = make_simple(env, node->left, 0,
                   return_type);
        }
        /* Assign variable */
        if (variable_name) {
                /* Assign a default initialization value for this type
                   */
                switch(variable_type) {
                        case INT:
                                assign(env, variable_name, int_value(0)
                                   , 1);
                                break;
                        case VOID:
                                assign(env, variable_name, void_value()
                                   , 1);
                                break;
                        case FUNCTION:
                                assign(env, variable_name, null_fn, 1);
```

```
                                        break;
                        }
                }
                else {
                        fatal("Could not ascertain variable name!");
                }
}

/* Go down the declarator tree initialising the variables, at this
   stage */
void register_variable_subtree_tac(environment *env, NODE *node, int
   return_type) {
        NODE *original_node = node;
        /* Ensure we have all required params */
        if (!env || !node || type_of(node) != '~') return;
        /* Skip over LEAF nodes */
        if (node->left != NULL && type_of(node->left) == LEAF) {
                node = node->left;
        }
        if (node->left != NULL && (type_of(node->left) == VOID ||
           type_of(node->left) == FUNCTION || type_of(node->left) ==
           INT)) {
                /* Find variable type */
                int variable_type = to_int(NULL, make_simple(env, node
                   ->left, 0, return_type));
                declare_variables_tac(env, original_node->right,
                   variable_type, return_type);
        }
}


/*
 * Make the given NODE simple - i.e. return a temporary for complex
    subtrees
 * The appropriate code is also generated and pushed onto the code
    stack
*/
value *make_simple(environment *env, NODE *node, int flag, int
   return_type) {
        int i_value = 0;
        char *s_tmp = NULL;
        value *val1 = NULL, *val2 = NULL, *temporary = NULL, *temp =
           NULL;
        static int if_count = 0;
        static int while_count = 0;
        tac_quad *temp_quad = NULL;
        environment *new_env = NULL;
```

```
if (node==NULL) return NULL;
switch(type_of(node)) {
        case LEAF:
                return make_simple(env, node->left, flag,
                    return_type);
        case CONSTANT:
                i_value = cast_from_node(node)->value;
                s_tmp = malloc(sizeof(char) * 25);
                sprintf(s_tmp, "%d", i_value);
                return int_value(i_value);
        case IDENTIFIER:
                return string_value(cast_from_node(node)->
                    lexeme);
        case IF:
                build_if_stmt(env, node, ++if_count, NULL, NULL
                    , flag, return_type);
                return NULL;
        case BREAK:
                s_tmp = malloc(sizeof(char) * 25);
                sprintf(s_tmp, "__while%dend", while_count);
                append_code(make_goto(s_tmp));
                return NULL;
        case CONTINUE:
                s_tmp = malloc(sizeof(char) * 25);
                sprintf(s_tmp, "__while%d", while_count);
                append_code(make_goto(s_tmp));
                return NULL;
        case WHILE:
                build_while_stmt(env, node, ++while_count, ++
                    if_count, flag, return_type);
                return NULL;
        case '=':
                if (flag == INTERPRET_FN_SCAN) return NULL;
                val1 = make_simple(env, node->left, flag,
                    return_type);
                val2 = make_simple(env, node->right, flag,
                    return_type);
                if (val2 && val2->value_type!=VT_INTEGR && val2
                    ->value_type!=VT_FUNCTN) {
                      if (val2->value_type == VT_STRING) {
                              val2 = get(env, val2->data.
                                  string_value);
                      }
                      else {
                              val2 = get(env, val2->
                                  identifier);
                      }
```

```
                if (!val2) fatal("Undeclared identifier
                    ");
            }
            /* Check the LHS variable has already been
                defined */
            temp = get(env, to_string(val1));
            assert(temp!=NULL, "Variable not defined");
            /* Type check the assignment */
            type_check_assignment(val1, val2,
                vt_type_convert(temp->value_type));
            temporary = assign(env, val1, val2, 0);
            if (flag != INTERPRET_FN_SCAN) append_code(
                make_quad_value("=", val2, NULL, temporary,
                TT_ASSIGN, 0));
            return NULL;
        case '*':
        case '/':
        case '>':
        case '<':
        case '%':
        case '-':
        case '+':
        case NE_OP:
        case LE_OP:
        case GE_OP:
        case EQ_OP:
            temporary = generate_temporary(env, int_value
                (0));
            val1 = make_simple(env, node->left, flag,
                return_type);
            val2 = make_simple(env, node->right, flag,
                return_type);
            if (val1->value_type==VT_STRING) val1 = get(env
                , correct_string_rep(val1));
            if (val2->value_type==VT_STRING) val2 = get(env
                , correct_string_rep(val2));
            assert(val1 != NULL, "Operand value 1 must not
                be null");
            assert(val2 != NULL, "Operand value 2 must not
                be null");
            if (flag != INTERPRET_FN_SCAN) append_code(
                make_quad_value(type_to_string(type_of(node)
                ), val1, val2, temporary, TT_OP, type_of(
                node)));
            return temporary;
        case '~':
```

```
            if (flag != INTERPRET_PARAMS && flag!=
               INTERPRET_FN_SCAN) {
                    /* Params should not be registered,
                       because at this point we're not in
                       the correct environment */
                    register_variable_subtree_tac(env, node
                       , VT_ANY);
            }
            val1 = make_simple(env, node->left, flag,
               return_type);
            val2 = make_simple(env, node->right, flag,
               return_type);
            if (flag == INTERPRET_PARAMS) {
                    return int_param(to_string(val2),
                       to_int(env, val1));
            }
            return NULL;
    case 'D':
            /* val1 is FN definition */
            /* val1 is executed in current environment */
            val1 = make_simple(env, node->left, flag,
               return_type);

            /* New FN body environment */
            new_env = create_environment(env);
            if (val1!=NULL) {
                    /* Point function to the correct fn
                       body */
                    val1->data.func->node_value = node->
                       right;
                    /* Store function definition in
                       environment */
                    val2 = store_function(env, val1,
                       new_env);
            }
            if (flag != INTERPRET_FN_SCAN) {
                    /* Write out FN Name label */
                    append_code(make_begin_fn(val2));
                    append_code(make_fn_def(val2));
                    /* Make init frame */
                    temp_quad = make_init_frame();
                    append_code(temp_quad);
                    /* Define parameters with default empty
                        values */
                    register_params(new_env, val2->data.
                       func->params);
                    append_code(make_fn_body(val2));
```

```
                /* Look inside fn body */
                val2 = make_simple(new_env, node->right
                    , EMBEDDED_FNS, val1->data.func->
                    return_type);
                /* Update prepare frame with
                    environment size */
                temp_quad->operand1 = int_value(
                    env_size(new_env));
                /* Write end of function marker */
                append_code(make_end_fn(val2));
        }
        return NULL;
case 'd':
        /* val1 is the type */
        val1 = make_simple(env, node->left, flag,
            return_type);
        /* val2 is fn name & params */
        val2 = make_simple(env, node->right, flag,
            return_type);
        /* Store return type */
        val2->data.func->return_type = to_int(env, val1
            );
        return val2;
case 'F':
        /* FN name in val1 */
        val1 = make_simple(env, node->left, flag,
            return_type);
        /* Pull our parameters */
        val2 = make_simple(env, node->right,
           INTERPRET_PARAMS, return_type);
        return build_function(env, val1, val2);
case RETURN:
        val1 = make_simple(env, node->left, flag,
            return_type);
        /* Provide lookup for non-constants */
        if (val1 && val1->value_type!=VT_INTEGR) {
                if (val1->value_type == VT_STRING) {
                        val1 = get(env, val1->data.
                            string_value);
                }
                else {
                        val1 = get(env, val1->
                            identifier);
                }
                if (!val1) fatal("Undeclared identifier
                    ");
        }
```

```
            type_check_return(val1, return_type);
            append_code(make_return(val1));
            return NULL;
        case ',':
            val1 = make_simple(env, node->left, flag,
                return_type);
            val2 = make_simple(env, node->right, flag,
                return_type);
            if (val1 && val2) {
                    return join(val1, val2);
            }
            return NULL;
        case APPLY:
            /* FN Name */
            val1 = make_simple(env, node->left, flag,
                return_type);
            /* Params */
            val2 = make_simple(env, node->right, flag,
                return_type);
            /* Lookup function */
            temp = search(env, to_string(val1), VT_FUNCTN,
                VT_ANY, 1);
            if (temp) {
                    int fn_return_type;
                    append_code(prepare_fn(val2));
                    append_code(push_params(env, val2));
                    /* If we can't typecheck, set a special
                        UNDEFINED flag to say we can't */
                    /* typecheck. This can happen with
                        function variables, we do not EASILY
                        know the */
                    /* return type of the functions they
                        are bound to until runtime. */
                    fn_return_type = UNDEFINED;
                    if (temp->data.func) {
                            fn_return_type = temp->data.
                                func->return_type;
                    }
                    /* Temporary for result (if any) */
                    switch(fn_return_type) {
                            case INT:
                                    temporary =
                                        generate_temporary(
                                        env, int_value(0));
                                    break;
                            case VOID:
```

```
                                        temporary =
                                            generate_temporary(
                                            env, NULL);
                                        break;
                                    case FUNCTION:
                                        temporary =
                                            generate_temporary(
                                            env, null_fn);
                                        break;
                                    default:
                                        temporary =
                                            generate_untypechecked_tempor
                                            (env);
                                        break;
                                }
                                append_code(make_fn_call(temporary,
                                    temp));
                                return temporary;
                            }
                            else {
                                fatal("Cannot find function '%s'",
                                    to_string(val1));
                            }
                            return NULL;
                        case FUNCTION:
                        case INT:
                        case VOID:
                            return int_value(type_of(node));
                        case ';':
                            make_simple(env, node->left, flag, return_type)
                                ;
                            make_simple(env, node->right, flag, return_type
                                );
                            return NULL;
                        default:
                            fatal("Unrecognised node type");
                            return NULL;
                    }
}


/* Start the TAC generator process at the top of the AST */
tac_quad *start_tac_gen(NODE *tree) {
        /* Do a scan for function definitions first */
        environment *initial_env;
        initial_env = create_environment(NULL);
        null_fn = build_null_function();
```

```
        make_simple(initial_env, tree, INTERPRET_FN_SCAN, INT);
        /* Actually generate the TAC */
        make_simple(initial_env, tree, 0, 0);
        return tac_output;
}
```

8.1.5 utils folder

conversion.c

```
#include "conversion.h"

/**
*       conversion.c by Henry Thacker
*       Version 2 - Rewritten 14/11/2009
*
*       Conversion functions for the --C language
*
*/

/* NODE -> TOKEN CAST */
TOKEN * cast_from_node(NODE *node) {
        return (node==NULL ? NULL : (TOKEN *)node);
}

/* Return type of node */
int type_of(NODE *node) {
        if (node==NULL) {
                return -1;
        }
        return node->type;
}

/* Join the values together - as a parameter list */
value *join(value *val1, value *val2) {
        val1->next = val2;
        return val1;
}


/* Convert from the VT types used in the environment into the parser
   generated token types */
int vt_type_convert(int type) {
        switch(type) {
                case VT_INTEGR:
                        return INT;
                case VT_FUNCTN:
                        return FUNCTION;
                case VT_VOID:
```

```
                        return VOID;
                default:
                        return -1;
        }
}

/* Convert int to string */
char *cmm_itoa(int int_val) {
        char *str;
        str = malloc(15 * sizeof(char));
        sprintf(str, "%d", int_val);
        return str;
}


/* ==== C TYPE -> VALUE UTILITIES ==== */

/* Make a string value out of a string */
value *string_value(char *val) {
        static int t_count = 0;
        value *tmp_value = calloc(1, sizeof(value));
        tmp_value->identifier = malloc(sizeof(char) * 15);
        sprintf(tmp_value->identifier, "_str%d", t_count);
        tmp_value->value_type = VT_STRING;
        tmp_value->data.string_value = (char *) malloc(sizeof(char) * (
           strlen(val) + 1));
        strcpy(tmp_value->data.string_value, val);
        t_count++;
        return tmp_value;
}

/* Make an int value out of an int */
value *int_value(int val) {
        static int t_count = 0;
        value *tmp_value = calloc(1, sizeof(value));
        tmp_value->identifier = malloc(sizeof(char) * 15);
        sprintf(tmp_value->identifier, "_int%d", t_count);
        tmp_value->value_type = VT_INTEGR;
        tmp_value->data.int_value = val;
        t_count++;
        return tmp_value;
}


/* Make an untyped value */
value *untyped_value() {
        static int t_count = 0;
```

```
        value *tmp_value = calloc(1, sizeof(value));
        tmp_value->identifier = malloc(sizeof(char) * 15);
        sprintf(tmp_value->identifier, "_ut%d", t_count);
        tmp_value->value_type = VT_UNTYPED;
        t_count++;
        return tmp_value;
}

/* Make an int param out of an int and an identifier */
value *int_param(char *identifier, int val) {
        value *tmp_value = calloc(1, sizeof(value));
        tmp_value->identifier = identifier;
        tmp_value->value_type = VT_INTEGR;
        tmp_value->data.int_value = val;
        return tmp_value;
}

/* ==== VALUE -> C TYPE UTILITIES ==== */
char *to_string(value *val) {
        if (val==NULL) return NULL;
        if (val->value_type!=VT_STRING) return NULL;
        return val->data.string_value;
}

int to_int(environment *env, value *val) {
        if (val==NULL) return UNDEFINED;
        if (val->value_type==VT_STRING && env) {
                /* It might be a variable that we should resolve */
                val = search(env, val->data.string_value, VT_INTEGR,
                   VT_ANY, 1);
                if (!val) fatal("Integer value expected");
                return val->data.int_value;
        }
        else if (val->value_type==VT_INTEGR) {
                return val->data.int_value;
        }
        fatal("Integer value expected");
        return UNDEFINED;
}

/* Return the correct string representation for a variable / string */
char *correct_string_rep(value *val) {
        if (val==NULL) return NULL;
        if (val->temporary) return val->identifier;
        if (val->value_type==VT_STRING) return to_string(val);
        if (val->value_type==VT_INTEGR && val->identifier[0]=='_')
           return cmm_itoa(to_int(NULL, val));
```

```
        return val->identifier;
}
```

environment.c

```
#include "environment.h"

/**
*       environment.c by Henry Thacker
*       Version 2 - Rewritten 14/11/2009
*
*       Environment structure utility functions for the --C language
*
*/

/* Create a new environment to store values in */
environment *create_environment(environment *static_link) {
        /* Assign space for environment */
        environment *new_environment = (environment *) calloc(1, sizeof
           (environment));
        /* Assign space to store values */
        new_environment->values = (value **) calloc(HASH_VALUE_SIZE,
           sizeof(value *));
        /* Assign static link ptr */
        new_environment->static_link = static_link;
        new_environment->env_size = 0;
        /* Record how nested this environment is */
        if (!static_link) {
                new_environment->nested_level = 0;
        }
        else {
                new_environment->nested_level = static_link->
                   nested_level + 1;
        }
        return new_environment;
}

/* ==== SEARCH Helper Fns ==== */

int matching_names(value *val, char *name) {
        if (!val || !name) return 0;
        return strcmp(val->identifier, name)==0;
}

int matching_types(value *val, int type) {
        if (!val) return 0;
        return type==VT_ANY || val->value_type==type;
}
```

```
int matching_return_type(value *val, int type) {
        if (!val) return 0;
        return type==VT_ANY || val->value_type!=VT_FUNCTN || val->data.
           func->return_type==type;
}

/* ==== End of SEARCH Helper Fns ==== */

/* Retrieve a value of a specific type from the environment */
/* Values which are in the nearest scope will be returned */
value *search(environment *env, char *identifier, int value_type, int
   return_type, int recursive) {
        /* Find out what position in the hashtable the value should be
           stored in */
        int hash_position = environment_hash(identifier);
        value *a_value;
        if (env == NULL) return NULL;
        a_value = env->values[hash_position];
        /* Try and find the matching value */
        while (a_value) {
                if (matching_names(a_value, identifier) &&
                   matching_types(a_value, value_type) &&
                   matching_return_type(a_value, return_type)) {
                        return a_value;
                }
                a_value = a_value->next;
        }
        /* Move up to the next environment */
        if (recursive) {
                return get(env->static_link, identifier);
        }
        return NULL;
}

value *last_if_evaluation(environment *env) {
        return search(env, IF_EVAL_SYMBOL, VT_INTEGR, VT_ANY, 0);
}

/* Retrieve a value of a ANY type from the environment */
/* Values which are in the nearest scope will be returned */
value *get(environment *env, char *identifier) {
        return search(env, identifier, VT_ANY, VT_ANY, 1);
}

char *return_type_as_string(int type) {
        switch(type) {
```

```
                case INT:
                        return "integer";
                case VOID:
                        return "void";
                case FUNCTION:
                        return "function";
        }
        return "unknown";
}

/* Recursively print values */
void debug_print_value(value *val) {
        if (DEBUG_ON && val) {
                if (val->temporary) {
                        printf("\ttemporary - identifier: %s -
                           var_number: %d\n", val->identifier, val->
                           variable_number);
                }
                else {
                        switch(val->value_type) {
                                case VT_VOID:
                                        printf("\tvoid - identifier: %s
                                            - var_number: %d\n", val->
                                           identifier, val->
                                           variable_number);
                                        break;
                                case VT_INTEGR:
                                        printf("\tint - identifier: %s,
                                            value: %d - var_number: %d\
                                           n", val->identifier, val->
                                           data.int_value, val->
                                           variable_number);
                                        break;
                                case VT_STRING:
                                        /* A string should never be
                                           stored in the hashtable -
                                           this language doesn't have
                                           strings */
                                        break;
                                case VT_FUNCTN:
                                        printf("\tfunc - identifier: %s
                                           , entry-point: %p, returns:
                                           %s, definition-env: %p,
                                           params: %d\n - var_number: %
                                           d\n", val->identifier, val->
                                           data.func->node_value,
                                           return_type_as_string(val->
```

```
                                        data.func->return_type), val
                                        ->data.func->definition_env,
                                         param_count(val), val->
                                        variable_number);
                                    break;
                                case VT_LINKED:
                                    break;
                            }
                        }
                        debug_print_value(val->next);
                }
        }

/* Register actual parameters in environment */
void define_parameters(environment *env, value *formal, value *actual,
    environment *search_env) {
        value *formal_param;
        value *actual_param;
        value *tmp;
        int dealtWith = 0;
        if (param_count(formal) != param_count(actual)) return;
        formal_param = formal->data.func->params;
        actual_param = actual;
        while(formal_param && actual_param) {
                /* Type check assignment */
                if (actual_param->value_type == VT_STRING) {
                        tmp = get(search_env, actual_param->data.
                            string_value);
                        if (!tmp) fatal("Could not look-up parameter
                            value");
                        type_check_assignment(string_value(formal_param
                            ->identifier), tmp, to_int(NULL,
                            formal_param));
                        store(env, tmp->value_type, formal_param->
                            identifier, tmp, 1, 1, 0, 0);
                }
                else {
                        type_check_assignment(string_value(formal_param
                            ->identifier), actual_param, to_int(NULL,
                            formal_param));
                        store(env, actual_param->value_type,
                            formal_param->identifier, actual_param, 1,
                            1, 0, 0);
                }
                formal_param = formal_param->next;
                actual_param = actual_param->next;
        }
```

```
}

/* Print environment debugging info */
void debug_environment(environment *env) {
        if (DEBUG_ON && env) {
                int i = 0;
                printf("----------\n");
                printf("Environment: %p\n", env);
                printf("Static link via: %p [Nested %d levels deep]\n",
                    env->static_link, env->nested_level);
                for (i = 0; i < HASH_VALUE_SIZE; i++) {
                        if (env->values[i]) {
                                printf("[%d]:\n", i);
                                debug_print_value(env->values[i]);
                        }
                }
        }
}

/* Return the size of the environment */
int env_size(environment *env) {
        int i = 0;
        int temp_size = 0;
        for (i = 0; i < HASH_VALUE_SIZE; i++) {
                if (env->values[i]) {
                        value *current = env->values[i];
                        while (current) {
                                temp_size++;
                                current = current->next;
                        }
                }
        }
        return temp_size;
}

/* Wrapper to store a function in the environment */
value *store_function(environment *env, value *func, environment *
   local_env) {
        /* Check we were passed valid data */
        if (!env || !func) return;
        if (func->value_type!=VT_FUNCTN) return;
        func->data.func->local_env = local_env;
        return store(env, VT_FUNCTN, func->identifier, func, 0, 1, 1,
           0);
}

/* Store variable in environment */
```

```
value *store(environment *env, int value_type, char *identifier, value
    *val, int is_param, int is_declarator, int is_fn_dec, int
    is_temporary) {
        value *new_value;
        int hash_position;
        /* Check entry will be valid */
        if (!identifier || (!val && value_type!=VT_VOID)) return NULL;
        if (val && val->value_type==VT_STRING) return NULL;
        /* Find out what position in the hashtable the value should be
            stored in */
        hash_position = environment_hash(identifier);
        /* The environment must be valid */
        if (!env) return NULL;
        /* Check for redefinition */
        if (!is_fn_dec && !is_declarator && value_type!=VT_FUNCTN && !
            is_param) {
                if (value_type==VT_UNTYPED) value_type = VT_ANY;
                /* Value already exists - overwrite */
                new_value = search(env, identifier, value_type, VT_ANY,
                    1);
                if (!new_value) {
                        fatal("Could not find identifier '%s' with type
                            %d", identifier);
                }
        }
        else if (is_fn_dec && value_type==VT_FUNCTN && search(env,
            identifier, value_type, VT_ANY, 1) && !is_param) {
                /* Functions may not be redefined if they exist
                    anywhere in the local / global scope */
                /* Only exception is if definition is EXACTLY the same,
                     this is the case if the entry point */
                /* of the two functions is the same. This can happen
                    when we do an initial pre-scan and then scan */
                /* over the whole AST again, for instance in the TAC
                    generator. */
                value * existing_fn = search(env, identifier,
                    value_type, VT_ANY, 1);
                if (existing_fn->data.func->node_value == val->data.
                    func->node_value) {
                        /* Return ptr to existing entry */
                        return existing_fn;
                }
                fatal("Function '%s' redefines another function.
                    Function redefinition is not allowed!", identifier);
                return NULL;
        }
        else {
```

```
        /* If is a declarator, we must NOT redefine an existing
            variable in the SAME local scope */
        if (!is_param && is_declarator && search(env,
           identifier, VT_ANY, VT_ANY, 0)) {
                fatal("Variable '%s' redeclares an existing
                   variable in local scope", identifier);
        }
        /* Build new value */
        new_value = (value *) calloc(1, sizeof(value));
        new_value->identifier = malloc((sizeof(char) * strlen(
           identifier)) + 1);
        strcpy(new_value->identifier, identifier);
        new_value->next = NULL;
        new_value->value_type = value_type;
        new_value->temporary = is_temporary;
        new_value->stored_in_env = env;
        if (is_declarator) {
                new_value->variable_number = env->env_size;
                env->env_size = env->env_size + 1;
        }
        /* Do we have any values in this position of the array?
            */
        if (!env->values[hash_position]) {
                /* Nothing exists here yet */
                env->values[hash_position] = new_value;
        }
        else {
                /* Value already in this hash position, append
                   to the end */
                value *current_val = env->values[hash_position
                   ];
                if (!current_val) fatal("Could not access
                   current value in hash position %d",
                   hash_position);
                for (;;) {
                        if (current_val->next==NULL) {
                                break;
                        }
                        current_val = current_val->next;
                }
                current_val->next = new_value;
        }
}
/* Assign correct value */
switch(value_type) {
        case VT_INTEGR:
```

```
                        new_value->data.int_value = val->data.int_value
                            ;
                        break;
                case VT_FUNCTN:
                        new_value->data.func = val->data.func;
                        break;
                case VT_VOID:
                default:
                        /* No value */
                        break;
        }
        debug_environment(env);
        return new_value;
}

/* ==== HASHTABLE UTILITIES ==== */

/* Find environment hashtable position to store the variable in */
int environment_hash(char *s) {
    int h = 0;
    while (*s != '\0') {
      h = (h<<4) ^ *s++;
    }
    return (0x7fffffff&h) % HASH_VALUE_SIZE;
}

/* Walk down a series of chained values to find the bottom one */
value *find_leaf_value(value *top) {
        if (top==NULL) return NULL;
        if (top->next==NULL) return top;
        return find_leaf_value(top->next);
}
```

output.c

```
#include "output.h"

/**
*       output.c by Henry Thacker
*       Version 2 - Rewritten 14/11/2009
*
*       System out functions for the --C language
*
*/

/* Fatal error */
void fatal(char *str, ...) {
        va_list arglist;
```

```
        char *str_arg;
        char *current_char;
        int i;
        va_start(arglist, str);
        printf("Fatal exception: ");
        /* Iterate through our format string */
        for (current_char = str; *current_char!='\0'; current_char++) {
                /* Check if a format specifier is coming up? */
                if (*current_char != '%') {
                        printf("%c", *current_char);
                        /* Move to the next char */
                        continue;
                }
                current_char++;
                switch(*current_char) {
                        /* What type of format specifier do we have? */
                        case 'c':
                                /* Pull out char as int */
                                i = va_arg(arglist, int);
                                printf("%c", i);
                                break;
                        case 'd':
                                /* Pull out an int */
                                i = va_arg(arglist, int);
                                printf("%d", i);
                                break;
                        case 's':
                                /* Pull out a string */
                                str_arg = va_arg(arglist, char *);
                                printf("%s", str_arg);
                                break;
                        case '%':
                                /* Allow percentage sign to be escaped
                                   */
                                printf("%%");
                                break;
                }
                va_end(arglist);
        }
        printf("\n");
        /* Fatal error, so exit with error code */
        exit(-1);
}

/* Print out debug info */
void debug(char *str) {
        if (DEBUG_ON) {
```

```
                printf("%s\n", str);
        }
}

/* Print return value */
void print_return_value(environment *env, value *val) {
        if (!val) return;
        switch(val->value_type) {
                case VT_INTEGR:
                        if (to_int(env, val) != UNDEFINED) printf("
                           Result: %d\n", to_int(env, val));
                        return;
                case VT_FUNCTN:
                        printf("Result: %p\n", val->data.func->
                           node_value);
                        return;
                default:
                        return;
        }
}
```

utilities.c

```
#include "utilities.h"

/**
*       utilities.c by Henry Thacker
*
*       Generic utility functions used throughout the project
*
*/

/* Assertion with error text */
void assert(int assertion, char *error) {
        if (!assertion) {
                printf("TAC Error: %s\n", error);
                exit(-1);
        }
}

/* Initialise a void placeholder value */
value *void_value(void) {
        static int t_count = 0;
        value *tmp_value = calloc(1, sizeof(value));
        char temporary_name[10];
        sprintf(temporary_name, "void%d", t_count);
        tmp_value->identifier = temporary_name;
        tmp_value->value_type = VT_VOID;
```

```
        t_count++;
        return tmp_value;
}

/* Check that the value can be returned by the function with a given
   declared return type */
void type_check_return(value *returned_value, int declared_return_type)
    {
        if (!returned_value && declared_return_type!=VOID) fatal("
           Expected a return value!");
        /* We can not EASILY check untyped return values without
           backtracking, so assume they are OK */
        if (returned_value->value_type == VT_UNTYPED) {
                return;
        }
        if (declared_return_type == INT) {
                if (returned_value->value_type != VT_INTEGR) {
                        fatal("Expected integer return value");
                }
        }
        if (declared_return_type == FUNCTION) {
                if (returned_value->value_type != VT_FUNCTN) {
                        fatal("Expected function return value");
                }
        }
}


/* Check that a variable with value: variable_value matches up with the
    declared type */
void type_check_assignment(value *variable_name, value *variable_value,
    int declared_type) {
        if (!variable_name || !variable_value) fatal("Not enough
           information to typecheck statement!");
        /* We can not EASILY check untyped values without backtracking,
            so assume they are OK */
        if (variable_value->value_type == VT_UNTYPED) {
                return;
        }
        /* VOID variables can never be assigned to! */
        if (declared_type == VOID) {
                fatal("Variable '%s' of type \"void\" can not be
                   assigned a value", to_string(variable_name));
        }
        if (declared_type == INT) {
                if (variable_value->value_type != VT_INTEGR) {
```

```
                        fatal("Variable '%s' of type \"int\" can not be
                            assigned a non-integer value", to_string(
                            variable_name));
                }
        }
        if (declared_type == FUNCTION) {
                if (variable_value->value_type != VT_FUNCTN) {
                        fatal("Variable '%s' of type \"function\" can
                            only point to functions", to_string(
                            variable_name));
                }
        }
}
```

8.2 Header files

8.2.1 interpreter folder

arithmetic.h

```
#ifndef __ARITH_H
#define __ARITH_H

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "C.tab.h"
#include "nodes.h"
#include "token.h"
#include "environment.h"
#include "conversion.h"

value *arithmetic(environment *, int, value *, value *);

#endif
```

interpreter.h

```
#ifndef __INTERP_H
#define __INTERP_H

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "C.tab.h"
#include "nodes.h"
#include "token.h"
#include "arithmetic.h"
#include "output.h"
```

```
#include "conversion.h"
#include "environment.h"
#include "utilities.h"

/* Interpretation flags */
#define INTERPRET_FULL INT_MIN + 1
#define INTERPRET_FN_SCAN INT_MIN + 2
#define INTERPRET_PARAMS INT_MIN + 3

/* Vars */
environment *initial_environment;
value *null_function;

/* Fn prototypes */
void start_interpret(NODE *);
value *string_temporary(char *);
void register_variable_subtree(environment *, NODE *, int);
value *int_temporary(int);
int param_count(value *);
char *to_string(value *);
value *evaluate(environment *, NODE *, int, int);
value *build_function(environment *, value *, value *);
value *assign(environment *, value *, value *, int);
value *build_null_function();

#endif
```

8.2.2 mips folder

codegen_utils.h

```
#ifndef __CGU_H
#define __CGU_H

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <time.h>
#include "environment.h"
#include "registers.h"
#include "tacgenerator.h"

#define OT_UNSET 2020
#define OT_REGISTER 3030
#define OT_OFFSET 4040
#define OT_CONSTANT 5050
#define OT_LABEL 6060
#define OT_FN_LABEL 7070
#define OT_ZERO_ADDRESS 8080
```

```
#define OT_COMMENT 9090

/* System registers enumeration */
enum sys_register {
        $zero = 0,
        $at = 1,
        $v0 = 2, $v1 = 3,
        $a0 = 4, $a1 = 5, $a2 = 6, $a3 = 7,
        $t0 = 8, $t1 = 9, $t2 = 10, $t3 = 11, $t4 = 12, $t5 = 13, $t6 =
            14,     $t7 = 15,
        $s0 = 16, $s1 = 17, $s2 = 18, $s3 = 19, $s4 = 20, $s5 = 21, $s6
            = 22, $s7 = 23,
        $t8 = 24, $t9 = 25,
        $k0 = 26, $k1 = 27,
        $gp = 28,
        $sp = 29,
        $fp = 30,
        $ra = 31
}sys_register;

typedef struct register_offset {
        enum sys_register reg;
        int offset;
}register_offset;

typedef union operand {
        enum sys_register reg;
        struct register_offset* reg_offset;
        int constant;
        char *label;
}operand;

typedef struct mips_instruction {
        char *operation;
        int operand1_type;
        int operand2_type;
        int operand3_type;
        union operand *operand1;
        union operand *operand2;
        union operand *operand3;
        char *comment;
        int indent_count;
        struct mips_instruction *next;
}mips_instruction;

char *register_name(enum sys_register);
void write_preamble();
```

```
void write_epilogue();
void write_activation_record_fn();
void write_register_fn_variable();
mips_instruction *mips_comment(operand *, int);
mips_instruction *mips(char *, int, int, int, operand *, operand *,
   operand *, char *, int);
operand *make_constant_operand(int);
operand *make_label_operand(char *, ...);
operand *make_register_operand(int);
operand *make_offset_operand();
mips_instruction *syscall(char *);

#endif
```

mips.h

```
#ifndef __MIPS_H
#define __MIPS_H

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "C.tab.h"
#include "nodes.h"
#include "conversion.h"
#include "token.h"
#include "registers.h"
#include "tacgenerator.h"
#include "interpreter.h"
#include "codegen_utils.h"
#include "optimisation.h"

/* Fn prototypes */
void code_gen(NODE *);

/* Store reference to entry point */
tac_quad *entry_point;

/* Location that code is written to */
mips_instruction *instructions;

/* Our compiler's view of all of the MIPS registers during compilation
   */
register_contents** regs;

/* Global variables */
int has_used_fn_variable;
value *current_fn;
```

```
#endif
```

optimisation.h

```
#ifndef __MOPT_H
#define __MOPT_H

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "C.tab.h"
#include "nodes.h"
#include "conversion.h"
#include "token.h"
#include "registers.h"
#include "codegen_utils.h"

mips_instruction *do_optimise(mips_instruction *);

#endif
```

registers.h

```
#ifndef __REGS_H
#define __REGS_H

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <time.h>
#include "environment.h"
#include "codegen_utils.h"

#define REG_COUNT 31

#define REG_NONE_FREE -500
#define REG_VALUE_NOT_AVAILABLE -501

typedef struct register_contents {
        value *contents; /* Value stored in the register */
        int accesses; /* How many times the value has been referenced
           */
        int assignment_id; /* What order this assignment was made */
        int modified;
}register_contents;

int already_in_reg(register_contents **, value *, environment *env, int
    *has_used_fn_variable);
```

```
int choose_best_reg(register_contents **, environment *);
int first_free_reg(register_contents **);
void save_t_regs(register_contents **, environment *);
void save_t_reg(register_contents **, int, environment *);
void clear_regs(register_contents **);
void print_register_view(register_contents **);
void init_register_view(register_contents **);

/* Count globally how many assignments we've made to a variable */
int regs_assignments;

#endif
```

8.2.3 tacgen folder

tacgenerator.h

```
#ifndef __TACGEN_H
#define __TACGEN_H

#define TT_IF 150
#define TT_GOTO 151
#define TT_LABEL 152
#define TT_OP 153
#define TT_ASSIGN 154
#define TT_RETURN 155
#define TT_BEGIN_FN 156
#define TT_END_FN 157
#define TT_FN_DEF 158
#define TT_INIT_FRAME 159
#define TT_PUSH_PARAM 160
#define TT_POP_PARAM 161
#define TT_FN_CALL 162
#define TT_PREPARE 163
#define TT_FN_BODY 164

#define EMBEDDED_FNS 300

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "C.tab.h"
#include "nodes.h"
#include "conversion.h"
#include "interpreter.h"
#include "token.h"

/* Fn prototypes */
```

```
struct tac_quad *make_quad_value(char *, value *, value *, value *, int
    , int);
struct tac_quad *start_tac_gen(NODE *);
value *make_simple(environment *, NODE *, int, int);
void print_tac(struct tac_quad *);

/* TAC structure */
typedef struct tac_quad {
        char *op;
        value *operand1;
        value *operand2;
        value *result;
        int type;
        int subtype;
        int level; /* Used for MIPS code generation */
        struct tac_quad *next;
}tac_quad;

typedef struct simple {
        char *label;
        struct tac_quad *code;
}simple;

value *null_fn;

tac_quad *tac_output;

#endif
```

8.2.4 utils folder

conversion.h

```
#ifndef __CONV_H
#define __CONV_H

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "C.tab.h"
#include "nodes.h"
#include "token.h"
#include "environment.h"

#define UNDEFINED INT_MIN + 3

char *to_string(value *);
int to_int(environment *, value *);
TOKEN * cast_from_node(NODE *);
```

```
value *int_value(int);
value *int_param(char *, int);
char *correct_string_rep(value *);
value *string_value(char *);
value *untyped_value();
value *join(value *, value *);
int type_of(NODE *);
int vt_type_convert(int);

#endif
```

environment.h

```
#ifndef __ENV_H
#define __ENV_H

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include "nodes.h"
#include "token.h"
#include "C.tab.h"

#define HASH_VALUE_SIZE 1000

#define DEBUG_ON 1

/* Value type - Integer */
#define VT_INTEGR INT_MIN + 1
/* Value type - String (for identifiers) */
#define VT_STRING INT_MIN + 2
/* Value type - Function ptr */
#define VT_FUNCTN INT_MIN + 3
/* Value type - linked values - for param lists, etc */
#define VT_LINKED INT_MIN + 4
/* Value type - void */
#define VT_VOID INT_MIN + 5

/* Any type - special type used in searches */
#define VT_ANY INT_MIN + 6

/* Temporary that cannot be EASILY type-checked at runtime */
#define VT_UNTYPED INT_MIN + 7

/* Special value which stores last if evaluation in environment */
#define IF_EVAL_SYMBOL "$IF"
#define CONTINUE_EVAL_SYMBOL "$CONTINUE"
```

```
#define BREAK_EVAL_SYMBOL "$BREAK"

/* Value structure */
typedef struct value {
        char *identifier;
        struct value *next;
        int value_type;
        int temporary;
        int variable_number;
        struct environment *stored_in_env;
        union {
                char *string_value;
                int int_value;
                struct value *linked_value;
                struct function_declaration *func;
        }data;
} value;

/* Environment structure */
typedef struct environment {
        int env_size;
        struct value **values;
        struct environment *static_link;
        int nested_level;
} environment;

/* Function declaration struct */
typedef struct function_declaration {
        int return_type;
        struct value *params;
        struct environment *definition_env;
        struct environment *local_env;
        NODE *node_value; /* Function entry point */
} function_declaration;

/* Fn prototypes */
int env_size(environment *env);
environment *create_environment(environment *);
value *find_leaf_value(value *);
value *get(environment *, char *);
extern value *string_value(char *);
void debug_print_value(value *);
value *last_if_evaluation(environment *);
void define_parameters(environment *, value *, value *, environment *);
value *store(environment *, int, char *, value *, int, int, int);
value *search(environment *, char *, int, int, int);
value *store_function(environment *, value *, environment *);
```

```
int environment_hash(char *);

#endif
```

output.h

```
#ifndef __OUTP_H
#define __OUTP_H

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdarg.h>
#include "C.tab.h"
#include "nodes.h"
#include "token.h"
#include "environment.h"
#include "conversion.h"

void fatal(char *, ...);
void debug(char *);
void print_return_value(environment *, value *);

#endif
```

utilities.h

```
#ifndef __UTIL_H
#define __UTIL_H

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "C.tab.h"
#include "nodes.h"
#include "conversion.h"
#include "token.h"

/* Fn prototypes */
void assert(int, char *);
void type_check_assignment(value *, value *, int);
void type_check_return(value *, int);
value *void_value(void);

#endif
```

Bibliography

[Aho et al., 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

[Dandamudi, 2005] Dandamudi, S. (2005). *Guide to Risc Processors for Programmers and Engineers*. Springer, Berlin.

[Grune, 2000] Grune, D. (2000). *Modern Compiler Design*. Wiley, New York.

[Muchnick, 1997] Muchnick, S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Diego.