Memory Ordering and Coherency Validation Language MOC-V

Final Report

Ben Panning
bjp2122

Table of Contents

1. Introduction

1.1. Summary

Increasing complexity and parallelism within the design of x86 processors has created a requirement for ever more sophisticated methods for validating memory ordering and coherency of those designs.

There are two main methods currently used to ensure functional correctness of x86 processors in terms of memory ordering and coherency. These methods are random instruction generators and focused tests.

Random instruction generators generate sequences of x86 instructions, and compare the resulting memory map to a simulated result for the same sequence. This has the benefit of a high degree of randomization, however many x86 logic bugs can be found much more quickly through directing aspects of the test sequence.

Focused testing involves creating a specific sequence of instructions to hit specific test cases. This method has the benefit of being directed towards cases of interest, however lacks the randomization required to be effective against wide ranges of potential logical bugs.

A third, as of yet untested, potential method for creating sequences of code to test memory ordering and coherency would be to develop a language that has the benefit of providing engineers with the ability to easily define a directed memory test with the added benefit of being able to describe a high-degree of randomization.

A language for Memory Ordering and Coherency Validation, MOC-V, addresses these needs.

1.2. Language Requirements

The proposed language is one that allows users to easily create stressful code sequences to test memory coherency and ordering. As such, it must meet the following requirements:

1. Ability to specify ranges of memory to be tested and easily access this range
    a. Arrays that allow unaligned accesses to be performed
    b. Arrays that are indexed “safely”, where the index wraps around if it goes beyond the bounds of the array
    c. Arrays can be accessed with a data size of any supported value
2. Ability to specify code randomization
    a. Segments of code can be specified by the user as Random Code Blocks
    b. Random Code Blocks can be called from within regular routines with a literal identifying the probability of it being called within the program

The language thus has two main requirements: allow programmers to easily interact with memory regions, and allow programmers to specify code randomization.

1.3. Implementation Scope

The implementation of MOC-V is narrow, in order to allow for completion within a single semester. The implementation is much like a proof of concept, in that it could not effectively be used to validate memory

ordering and coherency within an x86 processor. It does, however, demonstrate the possibility of such a language being used to achieve this.

The implementation of MOC-V compiles a source program that greatly resembles C into x86 assembly. The x86 assembly generated follows the format used by Microsofts assembler, MASM. This assembly is meant to be run in virtual-8086 mode, which allows it to be executed from any Window's or DOS command prompt that supports running virtual-8086 mode executables.

2. Language Tutorial

2.1. A Brief How-To for MOC-V

MOC-V is very similar to C. It is a procedural language that allows users to construct a program in C-like syntax, and arrays make a very important part of the language. Arrays in MOC-V are safe in that if indexed outside of the range of the array, the index will "wrap around" and index back into the array. Arrays are also declared as a size in bytes, and can be easily indexed using any array size (byte, word, dword). Arrays are always indexed at the byte level regardless of the data access size. The following are a few examples, with explanations given in comments (MOC-V comments follow the C-style enclosing comment).

```
array x[40]; /* a 40-byte array named x */
x[36]ord4 = 4; /* set the 4-byte ordinal (dword) at byte 36 to 4 */

ord2 y;
/* set 2-byte variable y to the value in array x at offset I, if I > 40, wrap */
y = x[I]ord2;
```

Functions also play an important role in MOC-V. Every program must have a "main" function as the entry point. Functions follow a C-like syntax in that they take a variable number of arguments and return either nothing or a single value. Arrays are always allocated within the data segment, and because of this arrays are can not be passed as function arguments. It is important to note that an array declared globally, outside of any function block, may be accessed by any function, whereas an array declared locally inside a function block is only visible within that block – even though they are both allocated within the data segment. The following are a few function examples.

```
array x[10];
array y[10];

void main()
{
        array z[20];
        ord4 a = 2;
        ord4 b = 3;
        /* x, y, z visible */

        a = foo(a, b);
}

ord4 foo(ord4 a, ord4 b)
{
        /* x, y visible */
        ord4 tmp;
```

```
        tmp = a + b + x[2]ord4 + y[b]ord4;
        return tmp;
}
```

MOC-V supports many mathematical operators that are supported in C, such as addition, subtraction, multiplication, division, remainder division, binary operators, and comparators. These may be used in any fashion within an expression. In terms of type checking, all variables and array accesses used within an expression must be the same size. There is no promotion or demotion of data types within MOC-V.

For more information on how to program within MOC-V, please see the Language Reference Manual in the following section.

2.2. Compiling and Running a MOC-V Program

Once you have created a MOC-V program, there are a few steps required to turn it into an executable program.

2.2.1. MOC-V Execution Environment

Before learning the steps for compilation, it is first important to understand the execution environment used in MOC-V. MOC-V compiles x86 programs intended to run in 8086 mode. The most convenient environment for running 8086 programs is virtual-8086 mode. This is something that can be executed from any Windows command prompt when running in 32-bit protected mode. It is also possible to boot natively to virtual-8086 mode using DOS and execute a MOC-V program there.

2.2.2. Restriction on Virtual-8086 Mode

MOC-V uses a 16-bit code segment, which means that all addresses and values are natively 16-bit. However, MOC-V assembles with the Microsoft Assembler (MASM) to support 32-bit extensions within these 16-bit code segments. This means that ord4 (dword) is a supported variable even though the execution environment is inherently 16-bit. The one hard-set restriction on the use of 32-bit values, is that a 32-bit value when used as an index into memory may not be larger than a 16-bit value – or else a general protection fault will be generated. 32-bit values may be used as index, but may not index more than 64KB of data.

2.2.3. Additional Restrictions on MOC-V

MOC-V makes the additional restrictions that code, data, and stack segment are limited to a single 4KB page. The data segment also reserves 16-bytes of data for display purposes, and is therefore limited to 4KB – 16 bytes. This should be sufficient for any proof of concept programs that are needed to be run on MOC-V, but should a program use more resources than a 4KB page can provide, either a general protection or a stack fault will occur.

2.2.4. Compiling the MOC-V Compiler

The MOC-V compiler comes with a Makefile. The MOC-V compiler can itself be compiled simply by extracting the source files to a directory and performing the "make" command within that directory. The source files within MOC-V include the following:

- scanner.mll
- parser.mly
- ast.mli
- tac_types.ml
- tac_display.ml
- tacgen.ml
- asm_types.ml
- asm_display.ml
- asmgen.ml
- mocv.ml
- Makefile

2.2.5. Compiling a MOC-V Program

A MOC-V program is compiled in several steps. The first step, after creating a MOC-V program and compiling the MOC-V compiler, is to compile the program into assembly using the "mocv" command as follows.

mocv.exe <filename> <seed value>

This follows the format of executing this on windows. On Linux, the "exe" extension can be dropped. The filename is the MOC-V file to compile, while the seed value is the Random Code Block seed to use. More on the RCB seed will be explained in the LRM.

Once the program is compiled, if there are no errors, an x86 assembly file will be generated with the same name as the filename input, but with the "asm" extension. This assembly file can be assembled into an x86 binary object using the Microsoft Assember, MASM. The command for this is as follows.

masm <assembly file>

The "ML" (MASM and LINK) command may also be used, however the linker portion of it will generate errors (the object file is still created).

Once the object file has been created, it can be linked into an executable **using the 16-bit linker "link16".** It is very important to use the 16-bit linker, as this is 16-bit code and will generate errors with the 32-bit linker.

Once the 16-bit linker is used to create an executable (and optionally address mapping information), the executable may be run simply by executing it within a Windows command prompt or DOS prompt.

3. Language Reference Manual

3.1. Data Types, Variables, and Operators

3.1.1. Data Types

MOC-V is concerned only with a limited number of data types. This language is concerned only with writing patterns of data to memory, and therefore does not implement any form of floating point numbers. The data types incorporated in this language are as follows:

void	The undefined type
ord1	One byte data type used in numerical calculations and data access
ord2	Two byte data type used in numerical calculations and data access
ord4	Four byte data type used in numerical calculations and data access

3.1.2. Variable Names and Declarations

Variable names within MOC-V are a series of consecutive non-white space characters that start with a letter, including an underscore, and contain one or more letters or numbers. Variables may not take the same name as any keywords defined within the language.

Variables must be declared before they can be used. Also, variables may not be declared more than once within the same scope. For instance, if a variable is declared within the global score, that variable may not be re-declared within the global scope or re-declared in any narrower scope contained within the global scope.

A variable declaration consists of a data type followed by a variable name, and an optional initialization. The format of this is as follows:

```
data_type variable_id = initial_value
```

So for instance, to declare a four byte variable named “x” and initialize it to the value 0x1234, the user would follow the form:

```
ord4 x = 0x1234 ;
```

Variables must be declared at the head of a function, or in the case of global variables at the head of the program.

3.1.3. Arrays

Arrays can be implemented to store collections of data types. The most common uses are for storing arrays of characters which can be interpreted as text and creating regions of memory to be tested. Arrays have the property of being safe, in that if they are indexed beyond their range they wrap-around and begin indexing from the beginning of the array. Arrays are defined as follows:

```
array array_id[size_in_ord4]
```

The first thing to notice is that an array has no data type associated with it. Arrays are always declared with a size given in bytes, and are always accessed on a byte boundary – regardless of the data type being

associated with them. This is to allow unaligned accesses when using ord2 and ord4 data types. Arrays are accessed as follows:

array_id[byte_offset]data_type

3.1.4. Arithmetic Operators

Both logical and bitwise arithmetic operations are supported in MOC-V. The set of valid arithmetic operators within MOC-V are as follows:

+	Addition
-	Subtraction
*	Multiplication
/	Integer division
%	Remainder division
^	Bitwise Exclusive-Or
|	Bitwise Or
&	Bitwise And
=	Assignment

There are also a number of valid comparators within MOC-V. These are operators which return one if the comparison is true or zero is the comparison is false. These operators are as follows:

==	Equal
!=	Not Equal
>	Greater than
<	Less than
>=	Greater than or Equal
<=	Less than or Equal

Finally, there are operators which are used to index into arrays and break expressions into explicit expression grouping – thereby guaranteeing an ordering of operations. These operators are as follows:

( )	Expression Grouping
[ ]	Array Indexing and RCB Blocks

All operators are left associative except for the assignment operator which is right associative. The precedence of operators is as follows:

Highest Precedence:	( )
	* / %
	+ -
	^ | &
	== != > < >= <=
Lowest Precedence:	=

3.2. Expressions and Program Structure Constructs

3.2.1. Comments

Comments are indicated either using an enclosing /* */ structure or using a single line double slash, //. MOC-V also supports nesting enclosed comments. For instance, the comment:

```
/*  This is /* a valid */ comment */
```

3.2.2. Expressions

Expressions are any constant, variable, or series of constants, variables, and operators that result in a value which can be assigned to a variable or used for comparison.

3.2.3. Statements

Statements consist of combinations of expressions that identify a single programmatic action. Statements must be ended with a semicolon, which represents a sequencing between individual statements. This sequencing between statements guarantees that one statement will be completed before the next statement is executed.

3.2.4. Conditional Constructs

MOC-V supports conditional constructs that allow if-then-else structures to be created. These constructs are as follows:

if(expr) stmt	If-statement with no concluding else-statement
if(expr) stmt else stmt	If-statement including else-statement

3.2.5. Looping Constructs

Looping constructs are supported both in the form of while-loops and for-loops. These may be constructed as follows:

while(expr) stmt	While statement that executes until expr evaluates to 0
for(expr ; expr ; expr) stmt	For statement with pre-executed expression, evaluated expression, and in loop executed expression

It should be noted that both the for and the while constructs require the expressions listed – they are not optional. In the case of the while loop, the expression is intended to be a comparison or calculation that evaluates to non-zero when the loop is to continue and zero when the loop is to terminate.

For the case of the for-loop, the expressions are intended to be an initialization, a calculation to determine if the loop terminates or continues, and an expression to be executed following each loop, respectively.

3.2.6. Functions

Functions are used within MOC-V to break programs into modular units that contain similar functionality. Functions are defined by declaring a type associated with the return value, which can be void to represent no return, followed by an identifier and a parameter list. The parameter list may contain void, but may not be empty. This is defined as follows:

```
data_type function_identifier( opt_param_list )
```

Where opt_param_list is a list of input parameters in the form of data_type followed by identifiers. Functions must be declared before being used, however, the declaration of a function can be the same as its definition. In the case where a function must be declared before it is defined, the syntax is as follows:

```
data_type function_id( opt_decl_list );
```

The difference between the function call and the function declaration is that the declaration must include the data types of each input parameter associated with the function. The form of this is a data type followed by an identifier. The identifier specified within the function declaration is then treated as a variable local to that function block – following the same scoping rules as a local variable declared within a function block.

3.2.7. Variables as Input Parameters

When a function takes a data type as input that is not an array, that variable is always passed by value. There is no concept of passing values by reference within MOC-V, which means that any variable passed to a function is guaranteed not to be modified or corrupted by that function.

3.2.8. Passing Arrays to Functions

Arrays in MOC-V are always declared globally within the data segment. Pointers are not supported within MOC-V, and because of this the only way to pass arrays between functions is to declare them globally and use them in functions using the global scope. This suffices for MOC-V since the language is intended to be

a language for validation, and is not intended to develop extensive libraries or become widely used commercially outside of the x86 validation world.

It should be noted that because arrays are always passed to functions as global symbols, it is not possible to create multi-threaded, re-entrant functions which take arrays as input.

3.2.9. Return Values from Functions

Functions may return any data type including void, but may not return arrays. Only a single instance of any returnable data type may be returned from any function.

Any function that requires multiple return values can do so using a globally defined array variable and a common interfacing to it between functions.

3.3. Program Structure and Scoping

3.3.1. Statement Blocks

Program structure within MOC-V is defined through statement blocks, function blocks, and random code blocks. Statement blocks consist of a set of statements between two brackets, as follows:

```
{ stmt_list }
```

The stmt_list in this format is a single statement or a list of statements. White space within this form is neglected.

Scoping rules within MOC-V maintain only two scopes: a global scope visible to every function and a local scope visible only within the function a variable is declared in. Scoping is static within MOC-V, and depends only on the program text and not run-time behavior.

3.3.2. Function Blocks

Function blocks include the first statement block following a function declaration. Function blocks may not be defined within other function blocks, and must also be declared before they are used within program order. All functions are declared at the global level, and are therefore visible to all other code segments following the functions declaration.

3.3.3. Random Code Blocks

MOC-V allows users to define random code blocks. These are code blocks which, when randomly selected for inclusion within the current compilation, are treated much like a normal function block – except with the restriction that it cannot accept a list of parameters or return any data value.

Random code blocks are declared using the following syntax:

```
rcode rcode_block_id;
```

The term rcode is a keyword which defines a random code block. A random code block is defined in the same manner that a function block is defined – which includes the rcode keyword followed by an identifier and then a statement block.

```
rcode rcode_block_id { stmt_list }
```

3.3.4. Random Code

One of the key features of MOC-V is that it allows the programmer to specify an amount of randomness within a program. This randomness is defined using random code blocks and the random code sequences are generated at compile time based on an input seed value.

Random code sequences are defined using the block declarations previously specified. The random code blocks act much like a function block, except that they are not guaranteed to run, and because they do not accept nor return values, cannot be integrated within the core functionality of a MOC-V program.

Once a random code block is defined, it may be used within the program structure as follows:

```
[ (weight)rcode_block_id ]
```

Where weight is a value that determines the odds that a given random code block is inserted at compile time. This weight must be an integer value between 0 and 100, where a value of 0 indicates that the random code block will never be inserted, and a value of 100 means that the code block will be inserted at every compilation.

The selection of random code blocks for insertion within the program can also be specified as selection from a list of random code blocks. This can be done using the following syntax:

```
[ (weight1)rcode_block_id1, (weight2)rcode_block_id2 … ]
```

Using this syntax, at compile time the compiler will select one or none of the listed random code blocks for insertion within the program. The weights specified within this syntax must sum to a value less than 100, and determination of which random code block is selected is based upon the weight of each code block using the same rules specified for a single random code block.

This form of selecting random code blocks may also be used within the random code block definitions themselves – allowing for programmers to randomize the random code blocks themselves.

It should be noted that random code blocks may reference other random code blocks, however it is not recommended to attempt recursion or cycles within random code blocks – as these lead to infinite loops within program structure.

3.3.5. Additional Scoping Rules

In addition to the block level scoping rules, it is also required that all variables are declared before they are used within each block. There is a single namespace within MOC-V that includes variable names, function names, and random code block names. Identical names, even when referring to objects of different types or usage, will be considered conflicting.

Statement blocks always use open scoping rules. This includes function blocks as well as random code blocks. This open scoping means that each block of code inherits the symbol tables of its calling blocks, including every symbol table in the chain all the way to the global scope.

4. Project Plan

4.1. Specification and Development Planning Process

Planning the specification for MOC-V involved taking experiences that I've had in the area of memory coherency validation and thinking about how it could be done in a better way. The basic requirements for creating a successful testing environment are the ability to specify at low level operations on memory, and the ability to provide randomization surrounding these memory accesses.

Because being able to interact with memory and resources at a low level is of high importance, the language design took a C-like direction in that it is entirely procedural and not object oriented. Originally, interacting with memory was to be done using pointers, however the decision was later made to stick with a simpler implementation using only arrays.

The requirement for a directed form of randomization lead to the creation of Random Code Blocks. Originally it was intended that Random Code Blocks (RCBs) directly populate the calling code with x86 instructions. Later, for simplicity and avoiding infinite loops of code generation in the case of circular RCB calls, RCBs were modeled after simplified functions.

4.1.1. Development

Planning development and testing of the project was relatively simple. I was a team of only one, so no extra coordination or communication was required like would have been with a larger group. Development was divided into four separate parts: scanner, parser, three address code generation, and assembly generation.

Each part was worked on in order of dependency, with the scanner being the first item completed and the assembly generation being the last item of dependency. The testing plan involved testing each phase starting with the parser. A series of display routines was developed for the parser, three address code, and assembly portions (it was later removed from the parser segment when it was clear there were no further issues there).

4.1.2. Testing

Testing was done upon completion of each segment on that segment using the display functionality. The parser required little initial debug, whereas the three address code and assembly segments required much more in-depth initial visual debug.

Following initial visual debug of the x86 assembly code, programs were compiled, assembled, and the final product testing using a hand written assembly display function for displaying 4-byte values in hexadecimal form. This was by far the most intensive debug and required several weeks to complete.

4.2. Programming Style

Coming up with consistent programming style was difficult for this project – since I have never before worked with the OCaml programming language. As I worked more with the project and my OCaml experience grew, the quality of the code I was producing also improved. Regardless of my inexperience with the language, there were several programming style topics I attempted to follows.

First, naming conventions were chosen to aptly describe each variable and function that was developed. For instance, lists are labeled directly as lists, or the are prefixed with an "l" to indicate that they are a list. Also, as elements of the compiled language are passed between segments, distinct types are used to prevent any errors with consistency or type mismatches.

Finally, although OCaml does not require the use of low level asserts to ensure pointers are set, the use of checks throughout the compile process at various stages were used to ensure the data being processed is consistent with what is expected. These checks can be seen commonly in many functions throughout the compiler, and most often are used for checking symbol table consistency.

4.3. Project Timeline

This project was worked on continuously throughout the semester. The original timeline I had anticipated included 2-3 weeks for completion of both the scanner and the parser, 3 weeks for three address code generation, 3 weeks for assembly generation, and the final 3 weeks to complete x86 emulation work and add additional features, including improving efficiency of compiled code.

The scanner and parser were both completed on schedule, and three address code generation was only a single week behind schedule. The assembly generation took several weeks longer than I had anticipated, however, and debug of the compiler took an additional several weeks. This resulted in the emulation and additional features being pushed too far back to complete in time – so these features had to be dropped.

4.4. Roles and Responsibilities

I am the only person who worked on this project, and because of this all development, testing, and reporting was my responsibility.

4.5. Software Developers Environment

The software developers environment used was vim (it actually has color coding for the OCaml language) and a command prompt. To compile the project, a Makefile was created and maintained. To assemble MOC-V code compiled MASM was used with its included 16-bit linker.

5. Architectural Design

5.1. Component Interactions

The MOC-V compiler is composed of four separate components: the scanner, the parser, the three address intermediate code generator, and the assembly generator. Each segment of the compiler in the order listed manipulates the source text and passes it to the next phase, until finally following the assembly generator phase x86 assembly code is available which can be assembled, linked, and run.

5.1.1. The Scanner and Parser

The scanner provides tokens which are interpreted by the parser. The parser takes these tokens and from them generates an abstract syntax tree for all expressions contained therein. The parser is also what creates the initial list of symbols which includes symbols declared within the program text. These symbols are collected and later used during three address intermediate code generation.

5.1.2. The Three Address Code Generator

The parser passes the collection of abstract syntax trees representing each expression as well as the list of declarations to the three address code generator. The three address code generator is responsible for taking the abstract syntax trees and from them generating a list of procedural three address code that properly executes the abstract syntax tree.

The first thing that the three address code generator does is construct a symbol table based on the parsers list of variable declarations. This symbol table is simply a mapping of symbol names to a type which gives information on the symbol, including the symbol name, size, parameter information, and type – including whether the symbol is a temporary, immediate, or declared value. These symbol tables are constructed for both the global scope and the local scope, with the local scope being inclusive of the global scope – yet the two remaining distinct and separate.

Once the symbol tables have been generated, the three address code generator then takes the list of abstract syntax trees, which represent the list of expressions at the statement level, performs a depth first search to generate a list of procedural three address code for each one, and appends each one of these statements to TAC list. These lists are created for each function and each RCB within the program. These lists along with the symbol tables are then passed to the assembly generator.

5.1.3. The Assembly Generator

The assembly generator is the final stage of the compilation process. This stage takes the TAC lists and symbol table information from the three address code generator and from them generate x86 16-bit assembly that executes the represented program.

The first thing the assembly generator does is creates its own symbol table, called the Symbol Alias Table, or SAT, and populates it with information based on the inputs from the TAC symbol table. The SAT contains unique information specific to assembly generation, including the name, symbol type, segment where the data is located, offset into the segment, and current register usage information.

Following construction of the symbol table, the assembly generator begins processing the individual TAC elements provided by the three address code generator. This process happens for each function and RCB that is passed to the assembly generator.

Processing the TAC elements involves taking the TAC element and, based upon the TAC operation and the operands used, generating from it a list of strings that are assembly code executing the TAC operation. A single TAC operation can involve multiple assembly strings, not only because x86 assembly involves two

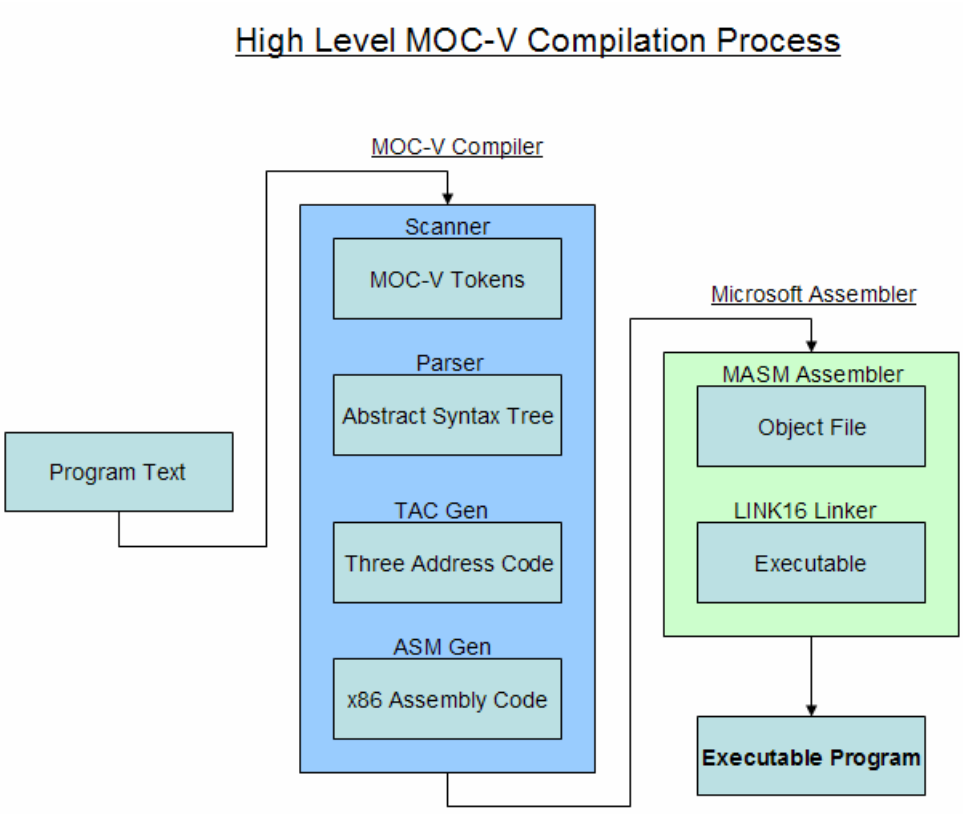
operands whereas the TAC operations involve three, but also because memory and register management is required to move symbols around so that they may be operated on.

During the entire process of generating assembly strings the Symbol Alias Table must be maintained to represent where all of the current symbols are in memory and in registers. The state of the symbol table is by far the most complex aspect of translating a three address code intermediate representation into an x86 assembly representation. This state must be maintained not only from instruction to instruction, but also through all possible execution paths. For instance, the state of the register set must be the exact same at a point in code regardless of what conditional branches were taken to get there. To simplify this, an operation was added to the list of TAC operations called "evict all", which would evict all symbols currently residing in memory back to memory and clear the register table. This was used quite liberally to preserve state around conditional and looping constructs.

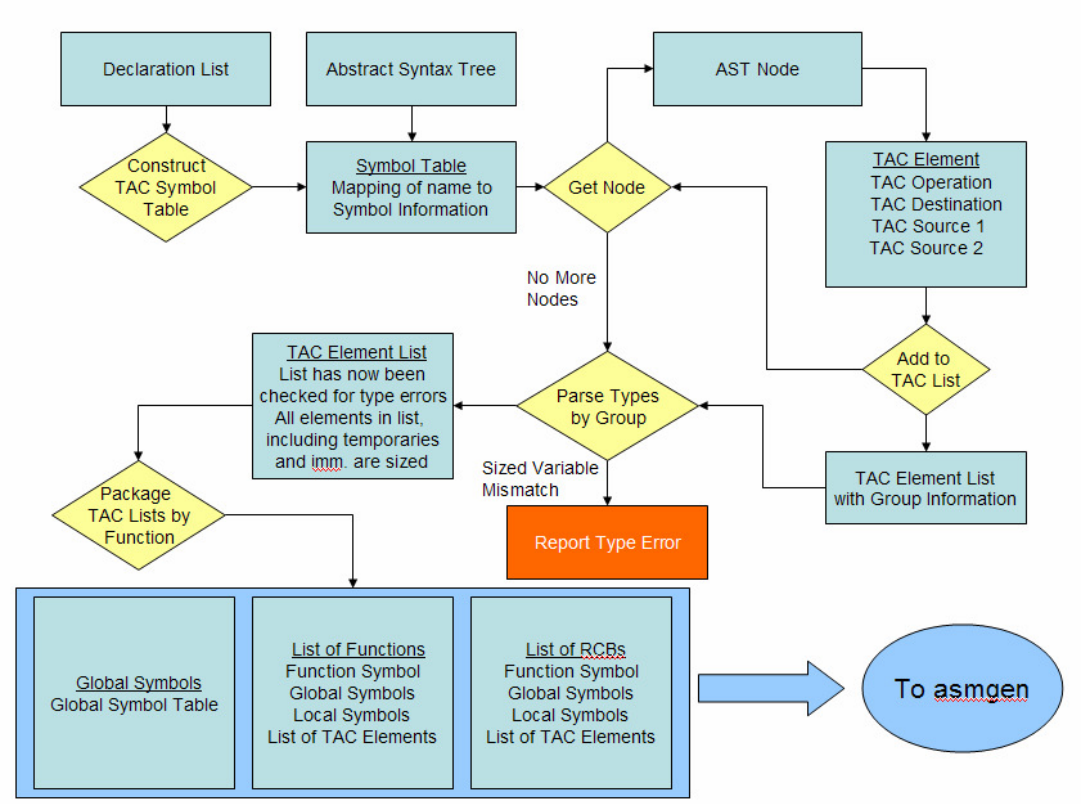
After creating a list of assembly strings that represent each function and RCB within the program, the assembly generator wrappers these items with the necessary assembly code to guarantee functionality – such as process definitions and stack save / restore functionality. The entire program is then wrappered with additional assembly code to guarantee correct operation within virtual-8086 mode, such as the segment declarations for stack, data, and code, as well as a start symbol and a jump to the "main" function.

The final piece of assembly added to this before it is provided to the main MOC-V function is the assembly required to print hexadecimal values. This is appended at the end within the same code segment as the compiled program. The reason not to place in it a separate code segment is to maintain consistency between calls and jumps within the program – all calls and jumps are near, and do not change the code segment register.

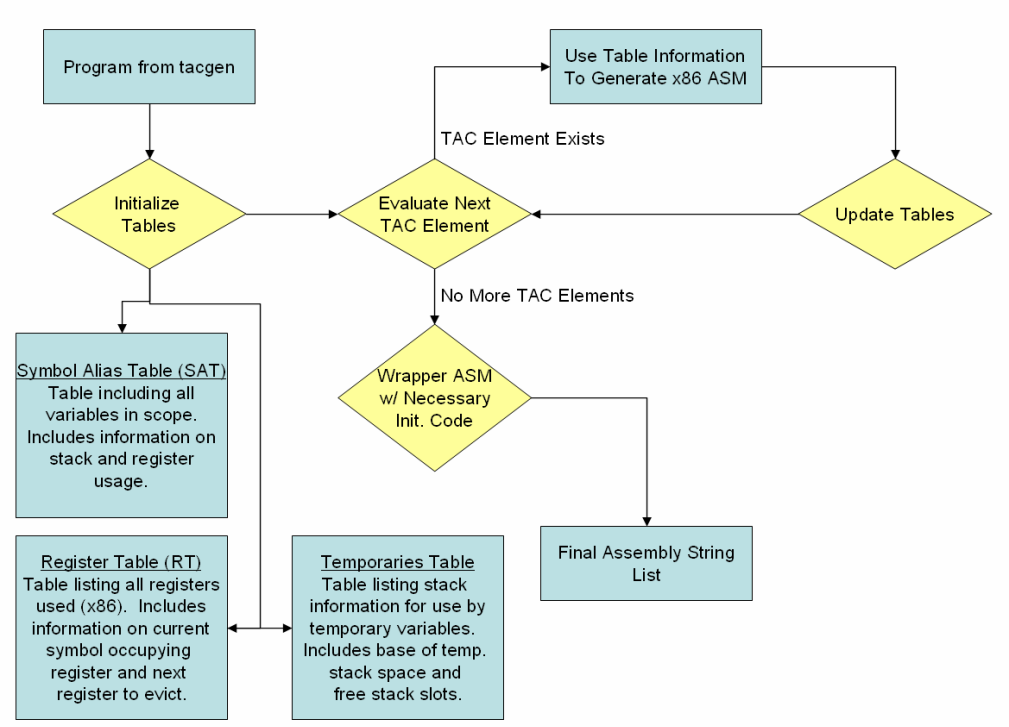
Flow of the Entire Compiler



Flow of the TAC Generator



Flow of the Assembly Generator



5.2. Interface Between Segments

As described within the architectural section above, the interface between segments of the compiler is simply passing a list of data to be processed by the next segment. The scanner passes a list of tokens to the parser. The parser processes these and passes a list of variable declarations as well as an abstract syntax tree to the three address code generator. The three address code generator takes the list of symbols and the abstract syntax tree, and from it generates a list of TAC elements. It also separates these elements into functions, RCBs, and a separate symbol table. All of these items are then passed to the assembly generator which has the sole output of a list of strings representing the assembly program.

For more information on the exact structures used to interface between segments, see the files tac_types.ml, asm_types.ml, and ast.mli within the appendix.

6. Test Plan

6.1. Testing Direction

Testing on MOC-V was performed after completion of each individual segment, starting with the parser. The parser was tested to make sure that input text produced an expected abstract syntax tree. This was all done using visual inspection.

The three address code generator was tested in much the same way. Programs were written in MOC-V and the TAC lists generated were visually inspected to ensure that they matched the expected TAC output for each program.

The most debug, however, was done on the assembly generator. At this stage, I was able to produce complete tests that could be run, and then output functional results from the run and inspect them for accuracy. This was done for a variety of tests. Some were smaller negative tests that just made sure that error checking worked correctly. Other tests were somewhat larger and tested things such as function calls and expression evaluation.

The types of programs that caught the most bugs in the compiler, however, were lengthy programs that performed complex calculations. The results from these calculations could be visually inspected and it was known right away whether there were compilation errors. A few examples of these include a recursive factorial calculator, a primitive bubble sort algorithm that worked on an array of data, and a program which found the shortest path between two nodes given a list of edges with corresponding distances (all using arrays, of course).

I also tended to focus test development to stress register usage and remapping, stack usage, and many conditional and looping constructs. Almost all of the bugs I found and corrected in some way had to do with the stack or symbol table not being maintained, or state not properly being maintained for every path of a conditional loop.

Provided in the appendix are a few examples of test programs used. Also, attached in the same e-mail as this report is a larger group of tests that may be used to demo the compiler (I am a CVN student, I realize I will not be able to demo this for you in person).

6.2. Automation Within Testing

My efforts to provide automation within the testing environment were unfortunately unsuccessful. I was unable to find an automated testing environment that worked well with the compilation and testing process I was using. I found the easiest way to do testing was to use the display functionality within MOC-V and make sure mathematical programs continued to display correct results after making compiler changes.

7. Lessons Learned

There were a large number of lessons learned from this process. First and foremost, the largest lesson that I learned during this process is that a compiler, even of a procedural low level language, is far more complex than I had originally anticipated. The amount of state that needs to be tracked during the compilation process is much larger than I originally thought – especially when going to x86 assembly. Had I known this when I started, I would have narrowed the original scope of the project considerably.

Another big lesson that I learned was not to go to the "final product" too quickly. I decided to produce assembly strings within the assembly generator segment of the compiler. This seemed like an appropriate format at the time, but later on I realized that had I used one more level of abstraction with the x86 assembly it would have been very easy to make some general efficiency improvements within the assembly code – however this was not possible within the string form. There was not time to make any efficiency improvements within the x86 code, but had there been time it wouldn't have been possible given the string format I jumped to too quickly.

A final lesson that this taught me was that there is a lot more to think about when producing assembly code than when producing code of a higher level. Before this compiler, I hadn't really thought about state needing to be maintained between looping and conditional branching constructs. I hadn't really thought about this before when I had hand-coded x86 assembly, but this compiler forced me to think about the starting architectural state of each loop, getting back to that state at the end of the loop, and also making sure that leaving the loop the architectural state is always the same. This took a lot of debug and work to get straight and get working.

7.1. Advice for Future Groups

My best advice for future groups, especially future CVN students working alone, would be to plan a fully featured project you think you can get done in a single semester – and then cut 1/3 of the features from it before making the proposal. This is especially important for the CVN students – things that seem trivial when you are first thinking about them are often times quite complex, and require quite a bit of effort to get working.

```
(* Header Section *)
{ open Parser }

(* Definitions Section *)
let digit = ['0' - '9']
let digit_h = ['0' - '9' 'a' - 'f' 'A' - 'F']
let number = ("0x")digit_h+ | digit+
let letter = ['a' - 'z' 'A' - 'Z' '_']
let id = letter ( letter | digit )*

(* Rules Section *)
rule token = parse
| [' ' '\t' '\n' '\r'] { token lexbuf }
| "/*"            { comment 0 lexbuf }
| '('             { LPAREN }
| ')'             { RPAREN }
| '{'             { LBRACE }
| '}'             { RBRACE }
| '['             { LBRACKET }
| ']'             { RBRACKET }
| ';'             { SEMI }
| ','             { COMMA }
| '+'             { PLUS }
| '-'             { MINUS }
| '*'             { TIMES }
| '/'             { DIVIDE }
| '%'             { RDIVIDE }
| '^'             { XOR }
| '|'             { OR }
| '&'             { AND }
| '!'             { NOT }
| '='             { ASSIGN }
| "=="            { EQ }
| "!="            { NEQ }
| '<'             { LT }
| "<="            { LEQ }
| '>'             { GT }
| ">="            { GEQ }
| "if"            { IF }
| "else"          { ELSE }
| "for"           { FOR }
| "while"         { WHILE }
| "return"        { RET }
| "void"          { VOID }
| "char"          { CHAR }
| "ord1"          { ORD1 }
| "ord2"          { ORD2 }
| "ord4"          { ORD4 }
| "array"         { ARRAY }
| "rcode"         { RCB }
| "#test_region"  { TREGION }
| number as lxm   { LITERAL(int_of_string lxm) }
| id as lxm       { ID(lxm) }
| eof             { EOF }
| _ as invd       { raise (Failure("Illegal character " ^ Char.escaped invd)) }

(* Support for Multi-Level Comments *)
and comment level = parse
| "*/"            { if level = 0 then token lexbuf
                    else comment (level - 1) lexbuf
                  }
| "/*"            { comment (level + 1) lexbuf }
| _               { comment level lexbuf       }
| eof             { print_endline "File ends within comment";
                    raise End_of_file
```

```
                    }
```

```
%{

open Ast

%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
%token PLUS MINUS TIMES DIVIDE RDIVIDE XOR OR AND NOT
%token ASSIGN
%token EQ NEQ GT GEQ LT LEQ
%token IF ELSE FOR WHILE
%token RET
%token VOID CHAR ORD1 ORD2 ORD4 ARRAY
%token RCB TREGION
%token EOF
%token <int> LITERAL
%token <string> ID

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN              /*  lowest precedence */
%left EQ NEQ LT LEQ GT GEQ
%left XOR OR AND NOT
%left PLUS MINUS
%left TIMES DIVIDE RDIVIDE
%nonassoc UMINUS           /*  highest precedence */

%start program
%type <Ast.program> program

%%

/* top level program */
program:
  /* nothing */      { [], [], [], [], [], [] }
| program vdecl      { $2 :: (fun (x, _, _, _, _, _) -> x) $1,
                       (fun (_, x, _, _, _, _) -> x) $1,
                       (fun (_, _, x, _, _, _) -> x) $1,
                       (fun (_, _, _, x, _, _) -> x) $1,
                       (fun (_, _, _, _, x, _) -> x) $1,
                       (fun (_, _, _, _, _, x) -> x) $1
                     }
| program fdecl      { (fun (x, _, _, _, _, _) -> x) $1,
                       $2 :: ((fun (_, x, _, _, _, _) -> x) $1),
                       (fun (_, _, x, _, _, _) -> x) $1,
                       (fun (_, _, _, x, _, _) -> x) $1,
                       (fun (_, _, _, _, x, _) -> x) $1,
                       (fun (_, _, _, _, _, x) -> x) $1
                     }
| program rcbdecl    { (fun (x, _, _, _, _, _) -> x) $1,
                       (fun (_, x, _, _, _, _) -> x) $1,
                       $2 :: ((fun (_, _, x, _, _, _) -> x) $1),
                       (fun (_, _, _, x, _, _) -> x) $1,
                       (fun (_, _, _, _, x, _) -> x) $1,
                       (fun (_, _, _, _, _, x) -> x) $1
                     }
| program tregdecl   { (fun (x, _, _, _, _, _) -> x) $1,
                       (fun (_, x, _, _, _, _) -> x) $1,
                       (fun (_, _, x, _, _, _) -> x) $1,
                       $2 :: ((fun (_, _, _, x, _, _) -> x) $1),
                       (fun (_, _, _, _, x, _) -> x) $1,
                       (fun (_, _, _, _, _, x) -> x) $1
                     }
| program fdef       { (fun (x, _, _, _, _, _) -> x) $1,
```

```
                        (fun (_, x, _, _, _, _) -> x) $1,
                        (fun (_, _, x, _, _, _) -> x) $1,
                        (fun (_, _, _, x, _, _) -> x) $1,
                        $2 :: ((fun (_, _, _, _, x, _) -> x) $1),
                        (fun (_, _, _, _, _, x) -> x) $1
                      }
| program rcbdef      { (fun (x, _, _, _, _, _) -> x) $1,
                        (fun (_, x, _, _, _, _) -> x) $1,
                        (fun (_, _, x, _, _, _) -> x) $1,
                        (fun (_, _, _, x, _, _) -> x) $1,
                        (fun (_, _, _, _, x, _) -> x) $1,
                        $2 :: ((fun (_, _, _, _, _, x) -> x) $1)
                      }


/* variable declaration */
vdecl:
  data_type ID SEMI                                        { Uninitvardecl($1, $2) }
| ARRAY ID LBRACKET LITERAL RBRACKET SEMI                  { Uninitarraydecl($2, $4) }
| data_type ID ASSIGN LITERAL SEMI                         { Initvardecl($1, $2, $4) }
| ARRAY ID LBRACKET LITERAL RBRACKET ASSIGN LITERAL SEMI { Initarraydecl($2, $4, $7) }



/* function declaration */
fdecl:
  data_type ID LPAREN opt_paramsdecl RPAREN SEMI {
          { fdecl_name = $2;
            fdecl_params = List.rev $4;
            fdecl_rtype = $1 }
        }

/* optional parameter declarations */
opt_paramsdecl:
  /* nothing */   { [] }
| paramsdecl_list { $1 }

/* list of parameter declarations */
paramsdecl_list:
  paramdecl { [$1] }
| paramsdecl_list COMMA paramdecl { $3 :: $1 }

/* parameter declaration */
paramdecl:
  data_type ID                          { Pdecl($1, $2); }
| ARRAY ID LBRACKET LITERAL RBRACKET    { Pdecla($2, $4); }

/* random code block declaration */
rcbdecl:
  RCB ID SEMI { $2 }

/* region of memory being tested declaration */
tregdecl:
  TREGION ID SEMI { $2 }


/* a function definition */
fdef:
  data_type ID LPAREN opt_paramsdecl RPAREN LBRACE opt_vdecl_list opt_stmt_list RBRACE
  {
          { fdef_name = $2;
            fdef_params = $4;
            fdef_vdecls = $7;
            fdef_body =  $8;
            fdef_rtype = $1 }
```

```
  }

/* a random code block definition */
rcbdef:
  RCB ID LBRACE opt_vdecl_list opt_stmt_list RBRACE {
          { rcbdef_name = $2;
            rcbdef_vdecls = $4;
            rcbdef_body = $5 }
  }

/* optional parameters */
opt_params:
        /* nothing */ { [] }
| params_list { List.rev $1 }

/* a list of paramters */
params_list:
        param { [$1] }
| params_list COMMA param { $3 :: $1 }

/* a single parameter */
param:
  ID { Id($1) }
| LITERAL { Literal($1) }


opt_stmt_list:
  /* nothing */ { [] }
| stmt_list     { List.rev $1 }

opt_vdecl_list:
  /* nothing */ { [] }
| vdecl_list    { List.rev $1 }

stmt_list:
  stmt { [$1] }
| stmt_list stmt { $2 :: $1 }

vdecl_list:
  vdecl { [$1] }
| vdecl_list vdecl { $2 :: $1 }


/* statements */
stmt:
  expr SEMI                                    { Expr($1) }
| RET expr SEMI                                { Return($2) }
| LBRACE opt_stmt_list RBRACE                  { Block($2) }
| IF LPAREN expr RPAREN stmt %prec NOELSE       { If($3, $5, Nostmt) }
| IF LPAREN expr RPAREN stmt ELSE stmt          { If($3, $5, $7) }
| FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt { For($3, $5, $7, $9) }
| WHILE LPAREN expr RPAREN stmt                 { While($3, $5) }
| LBRACKET rcb_list RBRACKET SEMI               { Rcbcall($2) }

/* expressions */
expr:
  LITERAL                                      { Literal($1) }
| ID                                           { Id($1) }
| ID LBRACKET expr RBRACKET data_type          { Array($1, $3, $5) }
| expr PLUS expr                               { Binop($1, Add, $3) }
| expr MINUS expr                              { Binop($1, Sub, $3) }
| expr TIMES expr                              { Binop($1, Mult, $3) }
| expr DIVIDE expr                             { Binop($1, Div, $3) }
| expr RDIVIDE expr                            { Binop($1, Rdiv, $3) }
| expr XOR expr                                { Binop($1, Xor, $3) }
```

```
| expr OR expr                                        { Binop($1, Or, $3) }
| expr AND expr                                       { Binop($1, And, $3) }
| expr EQ expr                                        { Binop($1, Equal, $3) }
| expr NEQ expr                                       { Binop($1, Neq, $3) }
| expr LT expr                                        { Binop($1, Less, $3) }
| expr LEQ expr                                       { Binop($1, Leq, $3) }
| expr GT expr                                        { Binop($1, Greater, $3) }
| expr GEQ expr                                       { Binop($1, Geq, $3) }
| MINUS expr %prec UMINUS                             { Uniop(Uminus, $2) }
| NOT expr                                            { Uniop(Not, $2) }
| ID ASSIGN expr                                      { Assign($1, $3) }
| ID LBRACKET expr RBRACKET data_type ASSIGN expr { Arrayassign($1, $3, $5, $7) }
| ID LPAREN opt_params RPAREN                         { Call($1, $3) }
| LPAREN expr RPAREN                                  { Paren($2) }


/* a list of random code block identifiers */
rcb_list:
  rcb_list COMMA rcb  {  $3 :: $1 }
| rcb                 { [$1] }

/* a single random code block
 *   this contains both the identifier for the random code block
 *   (potentially) being inserted, as well as a numerical value
 *   representing the "weight" of it being inserted.
 *
 *   this has format:  (num)id -> (23)my_rcb_1
 */
rcb:
  LPAREN LITERAL RPAREN ID   { ($2, $4) }

/* the data types - character and ordinals */
data_type:
  CHAR { Char }
| ORD1 { Ord1 }
| ORD2 { Ord2 }
| ORD4 { Ord4 }
| VOID { Void }
```

```
type binop = Add | Sub | Mult | Div | Rdiv | Xor | Or
   | And | Equal | Neq | Less | Leq | Greater | Geq

type unop = Uminus | Not

type data_type = Char | Ord1 | Ord2 | Ord4 | Void

type rcb = int * string

type expr =
        Literal of int
      | Id of string
      | Array of string * expr * data_type
      | Binop of expr * binop * expr
      | Uniop of unop * expr
      | Assign of string * expr
      | Arrayassign of string * expr * data_type * expr
      | Call of string * expr list
      | Paren of expr
      | Noexpr

type vdecl =
        Initvardecl of data_type * string * int
      | Uninitvardecl of data_type * string
      | Initarraydecl of string * int * int
      | Uninitarraydecl of string * int


type paramdecl =
        Pdecl of data_type * string
      | Pdecla of string * int

type stmt =
        Block of stmt list
      | Expr of expr
      | Return of expr
      | If of expr * stmt * stmt
      | For of expr * expr * expr * stmt
      | While of expr * stmt
      | Rcbcall of rcb list
      | Nostmt

type rcb_def = {
        rcbdef_name : string;
        rcbdef_vdecls : vdecl list;
        rcbdef_body : stmt list;
      }

type func_def = {
        fdef_name : string;
        fdef_params : paramdecl list;
        fdef_vdecls : vdecl list;
        fdef_body : stmt list;
        fdef_rtype : data_type;
      }

type rcb_decl = string

type treg_decl = string

type func_decl = {
        fdecl_name : string;
        fdecl_params : paramdecl list;
        fdecl_rtype : data_type;
      }
```

```
type program = vdecl list * func_decl list * rcb_decl list * treg_decl list * func_def
 list * rcb_def list
```

```
open Ast

module NameMap = Map.Make(struct
        type t = string
        let compare x y = Pervasives.compare x y
        end)

type symbol_type = Svar | Srcb | Sfunc | Sarray

type symbol = {
        symb_name : string;
        symb_stype : symbol_type;
        symb_dtype : data_type;
        symb_offset : int;
        symb_init : int;
        symb_size : int;
        symb_isparam : bool;
        symb_paramnum : int;
        symb_paramsizes : int list
        }

type dest_src_type = Dvar | Darray | Dimm | Dtmp

type tac_dest_src = {
        tds_type : dest_src_type;
        tds_name : string;
        tds_value : int;
        tds_size : int
        }

type tac_op = Tadd | Tsub | Tmul | Tdiv | Trdiv | Txor | Tor | Tand | Teq | Tne
            | Tlt | Tgt | Tle | Tge | Tuminus | Tnot | Tjmp | Tjmpe | Tjmpne
            | Tjmpgt | Tjmplt | Tjmpge | Tjmple | Tlabel | Tarray | Tasn | Taasn
            | Tcmp | Tret | Tparam | Tcall | Tevictall | Tespadd | Tevicteax
            | Tevictebx | Tevictecx | Tevictedx

let get_tac_op_from_binop op =
        match op with
            Add -> Tadd
          | Sub -> Tsub
          | Mult -> Tmul
          | Div -> Tdiv
          | Rdiv -> Trdiv
          | Xor -> Txor
          | Or -> Tor
          | And -> Tand
          | Equal -> Teq
          | Neq -> Tne
          | Less -> Tlt
          | Leq -> Tle
          | Greater -> Tgt
          | Geq -> Tge

let get_tac_op_from_uniop op =
        match op with
            Uminus -> Tuminus
          | Not -> Tnot

type tac_element = {
        to_op : tac_op;
        to_dest : tac_dest_src;
        to_src1 : tac_dest_src;
        to_src2 : tac_dest_src
        }
```

```
type tac_rcb = {
        rcb_symb : symbol;
        rcb_globals : symbol NameMap.t;
        rcb_locals : symbol NameMap.t;
        rcb_ltac : tac_element list
        }

type tac_func = {
        func_symb : symbol;
        func_globals : symbol NameMap.t;
        func_locals : symbol NameMap.t;
        func_ltac : tac_element list
        }

type tac_program = {
        prog_globals : symbol NameMap.t;
        prog_lrcb : tac_rcb list;
        prog_lfunc : tac_func list
        }

type tac_env = {
        te_label : int;          (* current label  *)
        te_trslt : int;          (* tmp result num *)
        te_seed : int;           (* seed for RCB elements *)
        te_group : int           (* group that tac element belongs too *)
        }

let dest_src_invld = {
        tds_type = Dvar;
        tds_name = "invld";
        tds_value = 0;
        tds_size = 0
        }

let dest_src_zero = {
        tds_type = Dimm;
        tds_name = "invld";
        tds_value = 0;
        tds_size = 1
        }
```

```
(*
 * tac_display.ml
 *)

(*
 * Routines to display three address code in a user friendly form.
 * This is intended only for debug.
 *)

open Tac_types
open Ast


let debug_print = true

(*
 * Return a string representing the TAC Op
 *)

(*
 * Return a string from a Three Address Code Operation
 *)
let string_from_tac op =
        match op with
            Tadd -> "Add"
          | Tsub -> "Sub"
          | Tmul -> "Mul"
          | Tdiv -> "Div"
          | Trdiv -> "RDiv"
          | Txor -> "Xor"
          | Tor -> "Or"
          | Tand -> "And"
          | Teq -> "Equal"
          | Tne -> "NEqual"
          | Tlt -> "Less"
          | Tgt -> "Greater"
          | Tle -> "LTE"
          | Tge -> "GTE"
          | Tuminus -> "Uminus"
          | Tnot -> "Not"
          | Tjmp -> "Jmp"
          | Tjmpe -> "Jmpe"
          | Tjmpne -> "Jmpne"
          | Tjmpgt -> "Jmpgt"
          | Tjmplt -> "Jmplt"
          | Tjmpge -> "Jmpge"
          | Tjmple -> "Jmple"
          | Tlabel -> "Label"
          | Tarray -> "Array"
          | Tasn -> "Assign"
          | Taasn -> "ArrAsn"
          | Tcmp -> "Cmp"
          | Tret -> "Ret"
          | Tparam -> "Param"
          | Tcall -> "Call"
          | Tespadd -> "esp add"
          | Tevictall -> "evict all"
          | Tevicteax -> "evict eax"
          | Tevictebx -> "evict ebx"
          | Tevictecx -> "evict ecx"
          | Tevictedx -> "evict edx"

(*
 * Return a string format of a dest / src
 *)
```

```
let string_from_dest_src ds =
        if ds.tds_type == Dvar then
                ("Var = " ^ ds.tds_name)
        else if ds.tds_type == Darray then
                ("Array = " ^ ds.tds_name ^ "[" ^ (string_of_int ds.tds_size) ^ "]")
        else if ds.tds_type == Dimm then
                ("Imm = " ^ (string_of_int ds.tds_value))
        else
                ("TRslt = " ^ (string_of_int ds.tds_value))

(*
 * Get string from a data_type
 *)
let string_from_data_type dt =
        match dt with
            Void -> "Void"
          | Char -> "Char"
          | Ord1 -> "Ord1"
          | Ord2 -> "Ord2"
          | Ord4 -> "Ord4"

(*
 * Return a string from a TAC dest/src type
 *)
let string_from_tds_type tdstype =
        match tdstype with
           Dvar -> "Dvar"
         | Darray -> "Darray"
         | Dimm -> "Dimm"
         | Dtmp -> "Dtmp"

(*
 * Print a TAC Element
 *)
let print_tac_dest_src dest_src =
        print_endline ("\t\tTYPE: " ^
        (string_from_tds_type dest_src.tds_type) ^
        "    NAME: " ^ dest_src.tds_name ^
        "    VALUE: " ^ (string_of_int dest_src.tds_value) ^
        "    SIZE: " ^ (string_of_int dest_src.tds_size))

(*
 * Print a single Three Address Code Element
 *)
let print_tac_el tac =
        print_endline ("\t< " ^ (string_from_tac tac.to_op) ^ ", " ^
        (string_from_dest_src tac.to_dest) ^ ", " ^
        (string_from_dest_src tac.to_src1) ^ ", " ^
        (string_from_dest_src tac.to_src2) ^ " >")

(*
 * Print a single group Three Address Code Element
 *)
let print_gtac_el gtac =
        let g = fst gtac in
        let tac = snd gtac in
        print_endline ("\t( " ^ (string_of_int g) ^
        ", < " ^ (string_from_tac tac.to_op) ^ ", " ^
        (string_from_dest_src tac.to_dest) ^ ", " ^
        (string_from_dest_src tac.to_src1) ^ ", " ^
        (string_from_dest_src tac.to_src2) ^ " > )")

(*
 * Print information on a Three Address Code element in the
 * case of an error.
```

```
 *)
let print_tac_error tac =
        print_endline "\tPRINT_TAC_ERROR";
        print_tac_el tac;
        print_endline "\tDESTINATION:";
        print_tac_dest_src tac.to_dest;
        print_endline "\tSOURCE 1:";
        print_tac_dest_src tac.to_src1;
        print_endline "\tSOURCE 2:";
        print_tac_dest_src tac.to_src2

(*
 * Print a list of Three Address Code elements
 *)
let rec print_tac_list tl =
        if tl == [] then print_endline "ENDFUNC"
        else
                begin
                let el = (List.hd tl) in
                print_tac_el el;
                print_tac_list (List.tl tl)
                end

(*
 * Print a list of grouped Three Address Code elements
 *)
let rec print_gtac_list gtl =
        if gtl == [] then print_endline "ENDFUNC"
        else
                begin
                let el = (List.hd gtl) in
                print_gtac_el el;
                print_gtac_list (List.tl gtl)
                end

(*
 * Print Three Address Code information for a function
 *)
let print_tac_func f =
        let fs = f.func_symb in
        print_endline ("FUNC:  " ^ (string_from_data_type fs.symb_dtype) ^
        " " ^ fs.symb_name);
        print_tac_list f.func_ltac

(*
 * Print Three Address Code information for a Random Code Block
 *)
let print_tac_rcb r =
        let rs = r.rcb_symb in
        print_endline ("RCB:  " ^ rs.symb_name);
        print_tac_list r.rcb_ltac

(*
 * Print Three Address Code information for an entire program
 *)
let print_tac_program p =
        List.map (fun x-> print_tac_func x) p.prog_lfunc;
        List.map (fun x -> print_tac_rcb x) p.prog_lrcb

(*
 * Debug Prints
 *)

(*
 * Print grouped TAC information if debug_print is true
```

```
 *)
let debug_print_gtacl gtac_list =
        if debug_print then
                print_gtac_list (List.rev gtac_list)
```

```
(*
 * tacgen.ml
 *)

open Ast
open Tac_types
open Tac_display

(*
 * Routines to generate Three Address Code
 *)

(*****************************************************************
 *
 * Routines for environment maintainence and sizing
 *
 *****************************************************************)

(*
 * get_seed
 *
 *      Get a seed value for an RCB call.  This both updates the
 *      "random" seed for the current call and provides it to the
 *      calling function.  Seed values are tracked in the environment.
 *
 * Output:  (env, seed)
 *)
let get_seed env =
        let next_seed = (((env.te_seed * 4297) + 391) mod 100) in
        let env = {
                te_label = env.te_label;
                te_trslt = env.te_trslt;
                te_seed = next_seed;
                te_group = env.te_group } in
        (env, next_seed)

(*
 * get_size_from_data_type
 *
 *      Get an integer representation of a data type size from an
 *      input MOC-V supported data type.
 *
 * Output:  (size)
 *)
let get_size_from_data_type dt =
       match dt with
           Char -> 1
         | Ord1 -> 1
         | Ord2 -> 2
         | Ord4 -> 4
         | Void -> 0

(*****************************************************************
 *
 * Routines to Generate Symbol Tables
 *
 *****************************************************************)

(*
 * populate_vdecl_symbols
 *
 *      Populate a symbols map given a list of variable declarations
 *      provided by the parser.  The symbols map is a mapping between
 *      symbol name and symbols information, provided in the form
 *      of the "symbol" type (see tac_types.ml).
```

```
 *
 * Output:  (symbol map)
 *)
let rec populate_vdecl_symbols list_in smap =
        if list_in == [] then smap
        else let curr_vdecl = List.hd list_in in
        match curr_vdecl with
            Initvardecl(dt, n, i) ->
                if NameMap.mem n smap then
                       raise (Failure ("variable already exists " ^ n))
                else
                  let curr_size = get_size_from_data_type dt in
                  let curr_symb = { symb_name = n;
                                    symb_stype = Svar;
                                    symb_dtype = dt;
                                    symb_offset = 0;
                                    symb_init = i;
                                    symb_size = curr_size;
                                    symb_isparam = false;
                                    symb_paramnum = 0;
                                    symb_paramsizes = [] } in
                  let smap = NameMap.add n curr_symb smap in
                  populate_vdecl_symbols (List.tl list_in) smap
          | Uninitvardecl(dt, n) ->
                if NameMap.mem n smap then
                       raise (Failure ("variable already exists " ^ n))
                else
                  let curr_size = get_size_from_data_type dt in
                  let curr_symb = { symb_name = n;
                                    symb_stype = Svar;
                                    symb_dtype = dt;
                                    symb_offset = 0;
                                    symb_init = 0;
                                    symb_size = curr_size;
                                    symb_isparam = false;
                                    symb_paramnum = 0;
                                    symb_paramsizes = [] } in
                  let smap = NameMap.add n curr_symb smap in
                  populate_vdecl_symbols (List.tl list_in) smap
          | Initarraydecl(n, s, i) ->
                if NameMap.mem n smap then
                       raise (Failure ("variable already exists " ^ n))
                else
                  let curr_symb = { symb_name = n;
                                    symb_stype = Sarray;
                                    symb_dtype = Ord1;
                                    symb_offset = 0;
                                    symb_init = i;
                                    symb_size = s;
                                    symb_isparam = false;
                                    symb_paramnum = 0;
                                    symb_paramsizes = [] } in
                  let smap = NameMap.add n curr_symb smap in
                  populate_vdecl_symbols (List.tl list_in) smap
          | Uninitarraydecl(n, s) ->
                if NameMap.mem n smap then
                       raise (Failure ("variable already exists " ^ n))
                else
                  let curr_symb = { symb_name = n;
                                    symb_stype = Sarray;
                                    symb_dtype = Ord1;
                                    symb_offset = 0;
                                    symb_init = 0;
                                    symb_size = s;
                                    symb_isparam = false;
```

```
                                        symb_paramnum = 0;
                                        symb_paramsizes = [] } in
                        let smap = NameMap.add n curr_symb smap in
                        populate_vdecl_symbols (List.tl list_in) smap

(*
 * populate_func_symbols
 *
 *      Populate a symbols map with function symbols.  These symbols
 *      follow the same format as other variables, and are provided
 *      by the list of function declarations from the parser.
 *
 * Output:  (symbol map)
 *)
let rec populate_func_symbols list_in smap =
        if list_in == [] then smap
        else let curr_fdecl = List.hd list_in in
        if NameMap.mem curr_fdecl.fdecl_name smap then
                raise(Failure ("function name already exists " ^ curr_fdecl.fdecl_name
))
        else
                begin
                let get_size_from_param p =
                        match p with
                            Pdecl(dt, n) -> (get_size_from_data_type dt)
                          | Pdecla(n, s) -> s
                in
                let rec get_param_list_sizes pl sl =
                        if pl == [] then sl
                        else
                                begin
                                let cp = List.hd pl in
                                let pl = List.tl pl in
                                let sl = (get_size_from_param cp) :: sl in
                                get_param_list_sizes pl sl
                                end
                in
                let pss = get_param_list_sizes curr_fdecl.fdecl_params [] in
                let curr_symb = { symb_name = curr_fdecl.fdecl_name;
                                symb_stype = Sfunc;
                                symb_dtype = curr_fdecl.fdecl_rtype;
                                symb_offset = 0;
                                symb_init = 0;
                                symb_size = 0;
                                symb_isparam = false;
                                symb_paramnum = 0;
                                symb_paramsizes = (List.rev pss) } in
                let smap = NameMap.add curr_fdecl.fdecl_name curr_symb smap in
                populate_func_symbols (List.tl list_in) smap
                end

(*
 * populate_rcb_symbols
 *
 *      Populate a symbols map with RCB symbol information provided
 *      by the parser.
 *
 * Output:  (symbol map)
 *)
let rec populate_rcb_symbols list_in smap =
        if list_in == [] then smap
        else let curr_rcb = List.hd list_in in
        if NameMap.mem curr_rcb smap then
                raise(Failure ("rcb name already exists " ^ curr_rcb))
        else let curr_symb = { symb_name = curr_rcb;
```

```
                                symb_stype = Srcb;
                                symb_dtype = Void;
                                symb_offset = 0;
                                symb_init = 0;
                                symb_size = 0;
                                symb_isparam = false;
                                symb_paramnum = 0;
                                symb_paramsizes = [] } in
        let smap = NameMap.add curr_rcb curr_symb smap in
        populate_rcb_symbols (List.tl list_in) smap

(*
 * populate_param_symbols
 *
 *      Populate a symbols map with symbol information from function
 *      parameters provided by the parser.
 *
 * Output:  (symbol map)
 *)
let rec populate_param_symbols list_in smap paramnum =
        if list_in == [] then smap
        else let curr_param = List.hd list_in in
        match curr_param with
            Pdecl(dt, n) ->
                    if NameMap.mem n smap then
                            raise (Failure ("param name already exists " ^ n))
                    else let curr_size = get_size_from_data_type dt in
                    let curr_symb = { symb_name = n;
                                      symb_stype = Svar;
                                      symb_dtype = dt;
                                      symb_offset = 0;
                                      symb_init = 0;
                                      symb_size = curr_size;
                                      symb_isparam = true;
                                      symb_paramnum = paramnum;
                                      symb_paramsizes = [] } in
                    let smap = NameMap.add n curr_symb smap in
                    populate_param_symbols (List.tl list_in) smap (paramnum - 1)
          | Pdecla(n, s) ->
                    if NameMap.mem n smap then
                            raise (Failure ("param array name already exists " ^ n))
                    else let curr_symb = { symb_name = n;
                                           symb_stype = Sarray;
                                           symb_dtype = Ord1;
                                           symb_offset = 0;
                                           symb_init = 0;
                                           symb_size = s;
                                           symb_isparam = true;
                                           (* these don't go on the stack *)
                                           symb_paramnum = 0;
                                           symb_paramsizes = [] } in
                    let smap = NameMap.add n curr_symb smap in
                    populate_param_symbols (List.tl list_in) smap paramnum



(*
 * populate_symbols
 *
 *      Populate all symbols from a program given a list of variable
 *      declarations, a list of function declarations, a list of
 *      RCB declarations, and a list of parameters.  This routine can
 *      be called for populating both global and local symbol maps.
 *
 * Output:  (symbol map)
```

```
 *)
let rec populate_symbols lvdecl_in lfdecl_in lrcbdecl_in params_in smap =
        if lvdecl_in != [] then
                let smap = populate_vdecl_symbols lvdecl_in smap in
                populate_symbols [] lfdecl_in lrcbdecl_in params_in smap
        else if lfdecl_in != [] then
                let smap = populate_func_symbols lfdecl_in smap in
                populate_symbols lvdecl_in [] lrcbdecl_in params_in smap
        else if lrcbdecl_in != [] then
                let smap = populate_rcb_symbols lrcbdecl_in smap in
                populate_symbols lvdecl_in lfdecl_in [] params_in smap
        else if params_in != [] then
                let paramnum = List.length params_in in
                let smap = populate_param_symbols params_in smap paramnum in
                populate_symbols lvdecl_in lfdecl_in lrcbdecl_in [] smap
        else smap


(*************************************************************
 *
 * Routines for Modifying Three Address Code Structures
 *
 *************************************************************)

(*
 * append_tac_list
 *
 *      Append a Three Address Code element to a list of such
 *      elements.
 *
 * Output:  (ltac)
 *)
let append_tac_list op d s1 s2 tac_list =
        let tac_el = {
                to_op = op;
                to_dest = d;
                to_src1 = s1;
                to_src2 = s2 } in
        tac_el :: tac_list

(*
 * append_tac_list_g
 *
 *      Append a Three Address Code element with grouping information
 *      to a list of such elements.
 *
 * Output:  (lgtac)
 *)
let append_tac_list_g g op d s1 s2 tac_list_g =
        let tac_el = {
                to_op = op;
                to_dest = d;
                to_src1 = s1;
                to_src2 = s2 } in
        (g, tac_el) :: tac_list_g

(*
 * get_label_num
 *
 *      Get a unique label number.
 *
 * Output:  (label number, env)
 *)
let get_label_num env =
        let lab = env.te_label in
```

```
        let next_env = {
                te_label = env.te_label + 1;
                te_trslt = env.te_trslt;
                te_seed = env.te_seed;
                te_group = env.te_group } in
        (lab, next_env)

(*
 * get_label_dest
 *
 *      Get a TAC dest/src type for a label based on the
 *      current environment.  This supplies the caller with
 *      a TAC dest/src specifying a unique label value.
 *
 * Output:  (label dest/src, env)
 *)
let get_label_dest env =
        let (num, env) = get_label_num env in
        let label_dest = {
                tds_type = Dimm;
                tds_name = "invld";
                tds_value = num;
                tds_size = 0 } in
        (label_dest, env)

(*
 * get_trslt_num
 *
 *      Get a unique number for a temporary result.
 *
 * Output:  (trslt number)
 *)
let get_trslt_num env =
        let tr = env.te_trslt in
        let next_env = {
                te_label = env.te_label;
                te_trslt = env.te_trslt + 1;
                te_seed = env.te_seed;
                te_group = env.te_group } in
        (tr, next_env)

(*
 * get_trslt_dest
 *
 *      Get a destination representing a unique temporary result.
 *
 * Output:  (trslt dest/src)
 *)
let get_trslt_dest env sz =
        let (num, env) = get_trslt_num env in
        let trslt_dest = {
                tds_type = Dtmp;
                tds_name = "invld";
                tds_value = num;
                tds_size = sz } in
        (trslt_dest, env)


(*************************************************************
 *
 * Routines for Type Checking - Which in MOC-V is size checking
 *
 *************************************************************)

(*
```

```
 * tac_get_expr_size_for_group
 *
 *      Determine an expression size for a particular group.  Every
 *      group identifies an expression, and every element within a
 *      single expression should have the same size.  This function
 *      determines what that size should be based on the variables
 *      contained within the expression.
 *
 * Output:  (size)
 *)
let rec tac_get_expr_size_for_group group tac_list =
        if tac_list == [] then 4 (* no sized variables, leave it 4 *)
        else if fst (List.hd tac_list) != group then
                (tac_get_expr_size_for_group group (List.tl tac_list))
        else
                begin
                let tace = snd (List.hd tac_list) in
                let op = tace.to_op in
                let d = tace.to_dest in
                let s1 = tace.to_src1 in
                let s2 = tace.to_src2 in
                if (op == Tadd || op == Tsub || op == Tmul || op == Tdiv ||
                   op == Trdiv || op == Txor || op == Tor || op == Tand ||
                   op == Teq || op == Tne || op == Tlt || op == Tgt ||
                   op == Tle || op == Tge) then
                        (*
                         * These are binary ops with the format:
                         *
                         *   < op, dest, src1, src2 >
                         *)
                        if d.tds_type == Dvar then
                                (* variable - sizing must hold *)
                                (d.tds_size)
                        else if d.tds_type == Darray then
                                (* array - sizing must also hold *)
                                (d.tds_value)  (* array access size *)
                        else if s1.tds_type == Dvar then
                                (s1.tds_size)
                        else if s1.tds_type == Darray then
                                (s1.tds_value)
                        else if s2.tds_type == Dvar then
                                (s2.tds_size)
                        else if s2.tds_type == Darray then
                                (s2.tds_value)
                        else (tac_get_expr_size_for_group group (List.tl tac_list))
                else if (op == Tuminus || op == Tnot || op == Tasn) then
                        (*
                         * These are unary ops with the format:
                         *
                         *      < op, dest, src, invld >
                         *)
                        if d.tds_type == Dvar then
                                (d.tds_size)
                        else if d.tds_type == Darray then
                                (d.tds_value)
                        else if s1.tds_type == Dvar then
                                (s1.tds_size)
                        else if s1.tds_type == Darray then
                                (s1.tds_value)
                        else (tac_get_expr_size_for_group group (List.tl tac_list))
                else if (op == Taasn) then
                        (*
                         * This is an array assignment with format:
                         *
                         *      < op, dest, invld, src >
```

```
                        *)
                        if d.tds_type != Darray then raise (Failure ("Taasn op " ^
                                "lacks array dest!"))
                        else (d.tds_value)
                else    (* Not an interesting tac op for sizing, continue *)
                        (tac_get_expr_size_for_group group (List.tl tac_list))
                end

(*
 * make_dest_src_size
 *
 *      Take an input TAC dest/src type and make it the specified
 *      size.
 *
 * Output:  (TAC dest/src)
 *)
let make_dest_src_size dest_src size =
        let dstype = dest_src.tds_type in
        let dsname = dest_src.tds_name in
        let dssize =
                (if dstype == Darray then dest_src.tds_size
                else size) in
        let dsvalue =
                (if dstype == Darray then size
                else dest_src.tds_value) in
        let out = {
                tds_type = dstype;
                tds_name = dsname;
                tds_size = dssize;
                tds_value = dsvalue } in
        (out)

(*
 * tac_populate_group_sizes
 *
 *      Take a list of grouped TAC elements and perform the size
 *      checking for each group within.  Also, size elements with
 *      flexible sizes within the group, such as immediates and
 *      temporaries.
 *
 * Output:  (lgtac)
 *)
let rec tac_populate_group_sizes group size lgtac_in lgtac_out =
        if lgtac_in == [] then (lgtac_out)
        else
        let gtac_in = List.hd lgtac_in in
        if (fst gtac_in) != group then
                begin
                (* this is not the same group *)
                let lgtac_out = gtac_in :: lgtac_out in
                let lgtac_in = List.tl lgtac_in in
                (tac_populate_group_sizes group size lgtac_in lgtac_out)
                end
        else
                begin
                (* this is the same group - check the op *)
                let tac = snd gtac_in in
                let op = tac.to_op in
                let d = tac.to_dest in
                let s1 = tac.to_src1 in
                let s2 = tac.to_src2 in
                if (op == Tadd || op == Tsub || op == Tmul || op == Tdiv ||
                   op == Trdiv || op == Txor || op == Tor || op == Tand ||
                   op == Teq || op == Tne || op == Tlt || op == Tgt ||
                   op == Tle || op == Tge) then
```

```
                begin
                (* < op, dest, src, src > *)
                let dsize =
                        (if (d.tds_type == Darray) then
                                d.tds_value else d.tds_size) in
                let s1size =
                        (if (s1.tds_type == Darray) then
                                s1.tds_value else s1.tds_size) in
                let s2size =
                        (if (s2.tds_type == Darray) then
                                s2.tds_value else s2.tds_size) in
                if ((d.tds_type == Darray || d.tds_type == Dvar) &&
                   (dsize != size)) then
                        let _ = Tac_display.print_tac_error tac in
                        raise (Failure ("size mismatch: expected size " ^
                                (string_of_int size)))
                else if ((s1.tds_type == Darray || s1.tds_type == Dvar) &&
                   (s1size != size)) then
                        let _ = Tac_display.print_tac_error tac in
                        raise (Failure ("size mismatch: expected size " ^
                                (string_of_int size)))
                else if ((s2.tds_type == Darray || s2.tds_type == Dvar) &&
                   (s2size != size)) then
                        let _ = Tac_display.print_tac_error tac in
                        raise (Failure ("size mismatch: expected size " ^
                                (string_of_int size)))
                else
                        (* all sizes checked out *)
                        let d = make_dest_src_size d size in
                        let s1 = make_dest_src_size s1 size in
                        let s2 = make_dest_src_size s2 size in
                        let tac = {
                                to_op = op;
                                to_dest = d;
                                to_src1 = s1;
                                to_src2 = s2 } in
                        let gtac = (group, tac) in
                        let lgtac_out = gtac :: lgtac_out in
                        let lgtac_in = List.tl lgtac_in in
                        (tac_populate_group_sizes group size lgtac_in
                                lgtac_out)
                end
        else if (op == Tuminus || op == Tnot || op == Tasn) then
                begin
                (* < op, dest, src, inv > *)
                let dsize =
                        (if (d.tds_type == Darray) then
                                d.tds_value else d.tds_size) in
                let s1size =
                        (if (s1.tds_type == Darray) then
                                s1.tds_value else s1.tds_size) in
                if ((d.tds_type == Darray || d.tds_type == Dvar) &&
                   (dsize != size)) then
                        let _ = Tac_display.print_tac_error tac in
                        raise (Failure ("size mismatch: expected size " ^
                                (string_of_int size)))
                else if ((s1.tds_type == Darray || s1.tds_type == Dvar) &&
                   (s1size != size)) then
                        let _ = Tac_display.print_tac_error tac in
                        raise (Failure ("size mismatch: expected size " ^
                                (string_of_int size)))
                else
                        (* all sizes checked out *)
                        let d = make_dest_src_size d size in
                        let s1 = make_dest_src_size s1 size in
```

```
                                let tac = {
                                        to_op = op;
                                        to_dest = d;
                                        to_src1 = s1;
                                        to_src2 = s2 } in
                                let gtac = (group, tac) in
                                let lgtac_out = gtac :: lgtac_out in
                                let lgtac_in = List.tl lgtac_in in
                                (tac_populate_group_sizes group size lgtac_in
                                        lgtac_out)
                        end
                else if (op == Taasn) then
                        begin
                        (* < op, array, inv, src > - inv == don't care *)
                        let dsize =
                                (if (d.tds_type == Darray) then
                                        d.tds_value else d.tds_size) in
                        let s2size =
                                (if (s2.tds_type == Darray) then
                                        s2.tds_value else s2.tds_size) in
                        if ((d.tds_type == Darray || d.tds_type == Dvar) &&
                           (dsize != size)) then
                                let _ = Tac_display.print_tac_error tac in
                                raise (Failure ("size mismatch"))
                        else if ((s2.tds_type == Darray || s2.tds_type == Dvar) &&
                           (s2size != size)) then
                                let _ = Tac_display.print_tac_error tac in
                                raise (Failure ("size mismatch"))
                        else
                                (* all sizes checked out *)
                                let d = make_dest_src_size d size in
                                let s2 = make_dest_src_size s2 size in
                                let tac = {
                                        to_op = op;
                                        to_dest = d;
                                        to_src1 = s1;
                                        to_src2 = s2 } in
                                let gtac = (group, tac) in
                                let lgtac_out = gtac :: lgtac_out in
                                let lgtac_in = List.tl lgtac_in in
                                (tac_populate_group_sizes group size lgtac_in
                                        lgtac_out)
                        end
                else (* not interesting for check, but still size it *)
                        begin
                        let d = make_dest_src_size d size in
                        let s1 = make_dest_src_size s1 size in
                        let s2 = make_dest_src_size s2 size in
                        let tac = {
                                to_op = op;
                                to_dest = d;
                                to_src1 = s1;
                                to_src2 = s2 } in
                        let gtac = (group, tac) in
                        let lgtac_out = gtac :: lgtac_out in
                        let lgtac_in = List.tl lgtac_in in
                        (tac_populate_group_sizes group size lgtac_in
                                lgtac_out)
                        end
                end

(*
 * get_max_group
 *
 *      Get the maximum group number in the list of gtac elements.
```

```
 *
 * Output:  (max number)
 *)
let rec get_max_group max lgtac =
        if lgtac == [] then max
        else
        let group = fst (List.hd lgtac) in
        let max =
                (if group > max then group else max) in
        let lgtac = List.tl lgtac in
        (get_max_group max lgtac)

(*
 * tac_populate_sizes
 *
 *      Populate element sizes for all groups within the list of
 *      grouped TAC elements.
 *
 * Output:  (lgtac)
 *)
let rec tac_populate_sizes group max lgtac =
        if group > max then lgtac
        else
                begin
                let size = tac_get_expr_size_for_group group lgtac in
                let lgtac = tac_populate_group_sizes group size
                        lgtac [] in
                let lgtac = List.rev lgtac in
                let group = group + 1 in
                (tac_populate_sizes group max lgtac)
                end

let tac_expr_sizing lgtac =
        let max = get_max_group 0 lgtac in
        let lgtac = tac_populate_sizes 0 max lgtac in
        let ltac = List.map snd lgtac in
        (ltac)


(*****************************************************************
 *
 * Routines for Evaluating Parser ASTs and Creating TAC Lists
 *
 *****************************************************************)

(*
 * tac_eval_expr
 *
 *      Evaluate an expression provided by the parser, and from it
 *      append elements to a list of Three Address Code.  The expressions
 *      are evaluated, and returned is the updated environment, the
 *      currend TAC destination where the result is stored (if a temporary
 *      result is required), and the appended TAC list.
 *
 * Output:  (env, TAC dest/src, ltac)
 *)
let rec tac_eval_expr env gsymb lsymb expr_in tac_list =
        match expr_in with
            Literal(i) ->
                    let curr_dest = {
                            tds_type = Dimm;
                            tds_name = "invld";
                            tds_value = i;
                            tds_size = 4 } in
                    (env, curr_dest, tac_list)
```

```
        | Id(n) ->
                let curr_symb =
                    (if NameMap.mem n gsymb then NameMap.find n gsymb
                     else if NameMap.mem n lsymb then NameMap.find n lsymb
                     else raise (Failure ("undeclared variable " ^ n))) in
                if curr_symb.symb_stype != Svar then
                        raise (Failure ("non-variable used as variable: " ^ n))
                else let curr_dest = {
                        tds_type = Dvar;
                        tds_name = n;
                        tds_value = 0;
                        tds_size = curr_symb.symb_size } in
                (env, curr_dest, tac_list)
        | Array(n, e, dt) ->
                (* evaluate the index expression *)
                let ind_env = {
                        te_label = env.te_label;
                        te_trslt = env.te_trslt;
                        te_seed = env.te_seed;
                        te_group = env.te_group + 1 } in
                let temp_group = env.te_group in
                let (env, ind_dest, tac_list) =
                        tac_eval_expr ind_env gsymb lsymb e tac_list in
                (* restore the current group *)
                let env = {
                        te_label = env.te_label;
                        te_trslt = env.te_trslt;
                        te_seed = env.te_seed;
                        te_group = temp_group } in
                let curr_symb =
                    (if NameMap.mem n gsymb then NameMap.find n gsymb
                     else if NameMap.mem n lsymb then NameMap.find n lsymb
                     else raise (Failure ("undeclared variable " ^ n))) in
                if curr_symb.symb_stype != Sarray then
                        raise (Failure ("non-array used as array: " ^ n))
                else
                    let ind_size =
                            curr_symb.symb_size / (get_size_from_data_type
                                    curr_symb.symb_dtype) in
                    let ind_size_src = {
                            tds_type = Dimm;
                            tds_name = "invld";
                            tds_value = ind_size;
                            tds_size = 0 } in
                    (* get a trslt for the index, and place it in there *)
                    let (ind_trslt, env) = get_trslt_dest env 4 in
                    (* < asn, dest, src, invld > *)
                    let tac_list = append_tac_list_g (env.te_group+1) Tasn
                            ind_trslt ind_dest dest_src_invld tac_list in
                    (* < rdiv, ind_dest, ind_dest, ind_size_src > *)
                    let tac_list = append_tac_list_g (env.te_group+1) Trdiv
                            ind_trslt ind_trslt ind_size_src tac_list in
                    let dsz = get_size_from_data_type dt in
                    let (trslt, env) = get_trslt_dest env dsz in
                    let array_dest = {
                            tds_type = Darray;
                            tds_name = n;
                            tds_value = dsz;
                            tds_size = curr_symb.symb_size } in
                    (* < array, trslt, array, index > *)
                    let tac_list = append_tac_list_g env.te_group Tarray
                            trslt array_dest ind_trslt tac_list in
                    (env, trslt, tac_list)
        | Binop(e1, op, e2) ->
             let (env, src1, tac_list) = tac_eval_expr env gsymb lsymb e1
```

```
                        tac_list in
                let (env, src2, tac_list) = tac_eval_expr env gsymb lsymb e2
                        tac_list in
                let (trslt, env) = get_trslt_dest env 4 in
                (* < op, trslt, src1, src2 > *)
                let tacop = get_tac_op_from_binop op in
                let tac_list = append_tac_list_g env.te_group tacop trslt src1 src2 ta
c_list in
                (env, trslt, tac_list)
          | Uniop(op, e) ->
                let (env, src1, tac_list) = tac_eval_expr env gsymb lsymb e
                        tac_list in
                let (trslt, env) = get_trslt_dest env 4 in
                let tacop = get_tac_op_from_uniop op in
                (* < op, trslt, src1, invld > *)
                let tac_list = append_tac_list_g env.te_group tacop trslt src1 dest_sr
c_invld
                        tac_list in
                (env, trslt, tac_list)
          | Assign(n, e) ->
                let (env, src1, tac_list) = tac_eval_expr env gsymb lsymb e
                        tac_list in
                let curr_symb =
                        (if NameMap.mem n gsymb then NameMap.find n gsymb
                         else if NameMap.mem n lsymb then NameMap.find n lsymb
                         else raise (Failure ("undeclared variable " ^ n))) in
                if curr_symb.symb_stype != Svar then
                        raise (Failure ("non-variable used as variable: " ^ n))
                else let curr_dest = {
                        tds_type = Dvar;
                        tds_name = n;
                        tds_value = 0;
                        tds_size = curr_symb.symb_size } in
                (* < Tasn, dest, src1, invld > *)
                let tac_list = append_tac_list_g env.te_group Tasn curr_dest src1
                        dest_src_invld tac_list in
                (env, curr_dest, tac_list)
          | Arrayassign(n, e1, dt, e2) ->
                let (env, src2, tac_list) = tac_eval_expr env gsymb lsymb e2
                        tac_list in
                (* index source is in a seperate group for sizings *)
                let temp_group = env.te_group in
                let env = {
                        te_label = env.te_label;
                        te_trslt = env.te_trslt;
                        te_seed = env.te_seed;
                        te_group = env.te_group + 1 } in
                let (env, ind_src, tac_list) = tac_eval_expr env gsymb lsymb e1
                        tac_list in
                (* restore the current group *)
                let env = {
                        te_label = env.te_label;
                        te_trslt = env.te_trslt;
                        te_seed = env.te_seed;
                        te_group = temp_group } in
                let curr_symb =
                        (if NameMap.mem n gsymb then NameMap.find n gsymb
                         else if NameMap.mem n lsymb then NameMap.find n lsymb
                         else raise (Failure ("undeclared variable " ^ n))) in
                if curr_symb.symb_stype != Sarray then
                        raise (Failure ("non-array used as array: " ^ n))
                else
                        let ind_size =
                                curr_symb.symb_size / (get_size_from_data_type
                                        curr_symb.symb_dtype) in
```

```
                        let ind_size_src = {
                                tds_type = Dimm;
                                tds_name = "invld";
                                tds_value = ind_size;
                                tds_size = 0 } in
                        (* get a trslt for the index, and place it in there *)
                        let (ind_trslt, env) = get_trslt_dest env 4 in
                        (* < asn, dest, src, invld > *)
                        let tac_list = append_tac_list_g (env.te_group+1) Tasn
                                ind_trslt ind_src dest_src_invld tac_list in
                        (* < rdiv, ind_src, ind_src, ind_size_src > *)
                        let tac_list = append_tac_list_g (env.te_group+1) Trdiv
                                ind_trslt ind_trslt ind_size_src tac_list in
                        let (trslt, env) = get_trslt_dest env 4 in
                        let array_dest = {
                                tds_type = Darray;
                                tds_name = n;
                                tds_value = (get_size_from_data_type dt);
                                tds_size = curr_symb.symb_size } in
                        (* < aasn, array, ind_reg, src2 > *)
                        let tac_list = append_tac_list_g env.te_group Taasn array_dest
                                ind_trslt src2 tac_list in
                        (* < asn, trslt, src2, invld > *)
                        let tac_list = append_tac_list_g env.te_group Tasn trslt src2
                                dest_src_invld tac_list in
                        (env, trslt, tac_list)
        | Call(n, el) ->
                let el = List.rev el in
                let fsymb =
                        (if NameMap.mem n gsymb then NameMap.find n gsymb
                         else raise (Failure ("undeclared function " ^ n))) in
                if fsymb.symb_stype != Sfunc then
                         raise (Failure ("non-variable used as variable: " ^ n))
                else
                        begin
                        let rec push_params env el psl tac_list =
                                if el == [] then
                                        begin
                                        if psl != [] then
                                                raise (Failure ("Function called
                                                without correct number of arguments:
                                                " ^ n))
                                        else (env, dest_src_invld, tac_list)
                                        end
                                else
                                        begin
                                        if psl == [] then
                                                raise (Failure ("Function called
                                                with incorrect number of arguments: "
                                                ^ n))
                                        else
                                                begin
                                                let ce = List.hd el in
                                                let el = List.tl el in
                                                let ps = List.hd psl in
                                                let psl = List.tl psl in
                                                let (env, trslt, tac_list) =
                                                      tac_eval_expr env gsymb lsymb
                                                            ce tac_list in
                                                let trslt = {
                                                              tds_type = trslt.tds_type;
                                                              tds_name = trslt.tds_name;
                                                              tds_value = trslt.tds_value;
                                                              tds_size = ps } in
                                                (* < Tparam, src_param, invld, invld > *)
```

```
                                        let tac_list = append_tac_list_g env.te_gr
oup
                                              Tparam trslt dest_src_invld
                                              dest_src_invld tac_list in
                                        push_params env el psl tac_list
                                        end
                                end
                        in
                        let (env, _, tac_list) = push_params env el
                                fsymb.symb_paramsizes tac_list
                        in
                        let num_params = List.length fsymb.symb_paramsizes in
                        let sz = get_size_from_data_type fsymb.symb_dtype in
                        let (rdest, env) = get_trslt_dest env sz in
                        let callsrc = {
                                tds_type = Dvar;
                                tds_name = n;
                                tds_value = 0;
                                tds_size = sz } in
                        (* finally, we make the call *)
                        (* < call, return_dest, func name, invld > *)
                        let tac_list = append_tac_list_g env.te_group Tcall rdest call
src
                                dest_src_invld tac_list in
                        (* following return, we must adjust the stack *)
                        (* add 4 for every parameter *)
                        let espsrc = {
                                tds_type = Dimm;
                                tds_name = "invld";
                                tds_value = (4 * num_params);
                                tds_size = 0
                                } in
                        (* < Tespadd, value, invld, invld > *)
                        let tac_list = append_tac_list_g env.te_group Tespadd espsrc
                                dest_src_invld dest_src_invld tac_list in
                        (env, rdest, tac_list)
                        end
          | Paren(e) ->
                let (env, trslt, tac_list) = tac_eval_expr env gsymb lsymb e
                        tac_list in
                (env, trslt, tac_list)
          | Noexpr -> (env, dest_src_invld, tac_list)


let rec append_lists l1 l2 =
        (* note: list 2 should be reversed *)
        if l2 == [] then l1
        else
                let l1 = (List.hd l2) :: l1 in
                (append_lists l1 (List.tl l2))

(*
 * tac_eval_stmt
 *
 *      Evaluate a statement provided by the parser in the form of an AST
 *      and from it generate a list of TAC elements representing the
 *      statements execution.
 *
 *      Note here that for consistency, these statements return a TAC
 *      dest/src, just like the expression evaluation does.  This is
 *      not used, however.
 *
 * Output:  (env, TAC dest/src, ltac)
 *)
```

```
let rec tac_eval_stmt env gsymb lsymb stmt_in tac_list =
        match stmt_in with
            Block(sl) ->
                    if sl == [] then (env, dest_src_invld, tac_list)
                    else
                        let (env, _, tac_list) =
                            tac_eval_stmt env gsymb lsymb (List.hd sl) tac_list
                        in
                        let (env, _, tac_list) =
                            tac_eval_stmt env gsymb lsymb (Block(List.tl sl))
                                tac_list in
                        (env, dest_src_invld, tac_list)
          | Expr(e) ->
                    let (env, _, gtac_list) =
                            tac_eval_expr env gsymb lsymb e [] in
                    let tmp_tac_list = tac_expr_sizing gtac_list in
                    let tac_list = append_lists tac_list (List.rev tmp_tac_list)
                    in
                    (env, dest_src_invld, tac_list)
          | Return(e) ->
                    let (env, dest, gtac_list) =
                            tac_eval_expr env gsymb lsymb e [] in
                    if dest.tds_type == Darray then
                           raise (Failure ("Return type array " ^
                                dest.tds_name))
                    else
                    (* < label, label_dest, invld, invld > *)
                    let gtac_list = append_tac_list_g env.te_group Tret dest
                            dest_src_invld dest_src_invld gtac_list in
                    let tmp_tac_list = tac_expr_sizing gtac_list in
                    let tac_list = append_lists tac_list (List.rev tmp_tac_list)
                    in
                    (env, dest_src_invld, tac_list)
          | If(e, s1, s2) ->
                    if s2 == Nostmt then
                        let (env, de, gtac_list) =
                                tac_eval_expr env gsymb lsymb e [] in
                        (* evict all registers *)
                        let gtac_list = append_tac_list_g env.te_group Tevictall
                                dest_src_invld dest_src_invld  dest_src_invld gtac_lis
t in
                        (* < cmp, invld, de, zero > *)
                        let gtac_list = append_tac_list_g env.te_group Tcmp dest_src_i
nvld de
                              dest_src_zero gtac_list in
                        let (jlbl, env) = get_label_dest env in
                        (* < jmpe, L$, invld, invld > *)
                        let gtac_list = append_tac_list_g env.te_group Tjmpe jlbl
                              dest_src_invld dest_src_invld gtac_list in
                        let tmp_tac_list = tac_expr_sizing gtac_list in
                        let tac_list = append_lists tac_list (List.rev tmp_tac_list)
                        in
                        let (env, _, tac_list) =
                              tac_eval_stmt env gsymb lsymb s1 tac_list in
                        (* evict all registers before leaving loop body *)
                        let tac_list = append_tac_list Tevictall dest_src_invld
                                dest_src_invld dest_src_invld tac_list in
                        (* < label, L$, invld, invld > *)
                        let tac_list = append_tac_list Tlabel jlbl
                              dest_src_invld dest_src_invld tac_list in
                        (env, dest_src_invld, tac_list)
                    else
                        let (env, de, gtac_list) =
                                tac_eval_expr env gsymb lsymb e [] in
                        (* evict all registers before entering loop body *)
```

```
                        let gtac_list = append_tac_list_g env.te_group Tevictall
                                dest_src_invld dest_src_invld dest_src_invld
                                gtac_list in
                        (* < cmp, invld, de, zero > *)
                        let gtac_list = append_tac_list_g env.te_group Tcmp dest_src_i
nvld de
                              dest_src_zero gtac_list in
                        let tmp_tac_list = tac_expr_sizing gtac_list in
                        let tac_list = append_lists tac_list (List.rev tmp_tac_list)
                        in
                        let (jlbl, env) = get_label_dest env in
                        (* < jmpe, L$, invld, invld > *)
                        let tac_list = append_tac_list Tjmpe jlbl
                              dest_src_invld dest_src_invld tac_list in
                        let (env, _, tac_list) =
                              tac_eval_stmt env gsymb lsymb s1 tac_list in
                        let (jlbl2, env) = get_label_dest env in
                        (* evict all registers before leaving this if block *)
                        let tac_list = append_tac_list Tevictall dest_src_invld
                                dest_src_invld dest_src_invld tac_list in
                        (* < jmp, jlbl2, invld, invld > *)
                        let tac_list = append_tac_list Tjmp jlbl2
                              dest_src_invld dest_src_invld tac_list in
                        (* < label, L$, invld, invld > *)
                        let tac_list = append_tac_list Tlabel jlbl
                              dest_src_invld dest_src_invld tac_list in
                        let (env, _, tac_list) =
                              tac_eval_stmt env gsymb lsymb s2 tac_list in
                        (* evict all registers before leaving this if block *)
                        let tac_list = append_tac_list Tevictall dest_src_invld
                                dest_src_invld dest_src_invld tac_list in
                        (* < lable, jlbl2, invld, invld > *)
                        let tac_list = append_tac_list Tlabel jlbl2
                              dest_src_invld dest_src_invld tac_list in
                        (env, dest_src_invld, tac_list)
          | For(e1, e2, e3, s) ->
                        (* start evaluation of expressions *)
                        let (env, _, gtac_list) = tac_eval_expr env gsymb lsymb
                                e1 [] in
                        (* Evict all registers before starting loop *)
                        (* < Tevictall, invld, invld, invld > *)
                        let gtac_list = append_tac_list_g env.te_group Tevictall dest_
src_invld
                                dest_src_invld dest_src_invld gtac_list in
                        let tmp_tac_list = tac_expr_sizing gtac_list in
                        let tac_list = append_lists tac_list (List.rev tmp_tac_list)
                        in
                        let (jlbl, env) = get_label_dest env in
                        (* < label, jlbl, invld, invld > *)
                        let tac_list = append_tac_list Tlabel jlbl
                                dest_src_invld dest_src_invld tac_list in
                        let (env, trslt, gtac_list) = tac_eval_expr env gsymb
                                lsymb e2 [] in
                        (* < cmp, invld, trslt, zero > *)
                        let gtac_list = append_tac_list_g env.te_group Tcmp dest_src_i
nvld
                                trslt dest_src_zero gtac_list in
                        let tmp_tac_list = tac_expr_sizing gtac_list in
                        let tac_list = append_lists tac_list (List.rev tmp_tac_list)
                        in
                        let (jlbl2, env) = get_label_dest env in
                        (* < jmpe, jlbl2, invld, invld > *)
                        let tac_list = append_tac_list Tjmpe jlbl2
                                dest_src_invld dest_src_invld tac_list in
                        let (env, _, tac_list) = tac_eval_stmt env gsymb lsymb
```

```
                        s tac_list in
                    let (env, trslt2, gtac_list) = tac_eval_expr env gsymb
                        lsymb e3 [] in
                    let tmp_tac_list = tac_expr_sizing gtac_list in
                    let tac_list = append_lists tac_list (List.rev tmp_tac_list)
                    in
                    (* Evict before looping back *)
                    (* < Tevictall, invld, invld, invld > *)
                    let tac_list = append_tac_list Tevictall dest_src_invld
                        dest_src_invld dest_src_invld tac_list in
                    (* < jmp, jlbl, invld, invld > *)
                    let tac_list = append_tac_list Tjmp jlbl dest_src_invld
                        dest_src_invld tac_list in
                    (* < label, jlbl2, invld, invld > *)
                    let tac_list = append_tac_list Tlabel jlbl2
                        dest_src_invld dest_src_invld tac_list in
                    (env, dest_src_invld, tac_list)
          | While(e, s) ->
                    (* Evict all registers before starting loop *)
                    (* < Tevictall, invld, invld, invld > *)
                    let tac_list = append_tac_list Tevictall dest_src_invld
                        dest_src_invld dest_src_invld tac_list in
                    let (jlbl1, env) = get_label_dest env in
                    (* < label, jlbl1, invld, invld > *)
                    let tac_list = append_tac_list Tlabel jlbl1
                        dest_src_invld dest_src_invld tac_list in
                    let (env, trslt, gtac_list) =
                        tac_eval_expr env gsymb lsymb e [] in
                    (* < cmp, invld, trslt, zero > *)
                    let gtac_list = append_tac_list_g env.te_group Tcmp dest_src_i
nvld trslt
                        dest_src_zero gtac_list in
                    let tmp_tac_list = tac_expr_sizing gtac_list in
                    let tac_list = append_lists tac_list (List.rev tmp_tac_list)
                    in
                    let (jlbl2, env) = get_label_dest env in
                    (* < jmpe, jlbl2, invld, invld > *)
                    let tac_list = append_tac_list Tjmpe jlbl2
                        dest_src_invld dest_src_invld tac_list in
                    let (env, _, tac_list) =
                        tac_eval_stmt env gsymb lsymb s tac_list in
                    (* Evict all registers before looping *)
                    (* < Tevictall, invld, invld, invld > *)
                    let tac_list = append_tac_list Tevictall dest_src_invld
                        dest_src_invld dest_src_invld tac_list in
                    (*
                     * We must evict again before jumping back - this
                     * guarentees that between loop iterations the assembly
                     * is in the same state.
                     *)
                    (* < Tevictall, invld, invld, invld > *)
                    let tac_list = append_tac_list Tevictall dest_src_invld
                        dest_src_invld dest_src_invld tac_list in
                    (* < jmp, jlbl1, invld, invld > *)
                    let tac_list = append_tac_list Tjmp jlbl1 dest_src_invld
                        dest_src_invld tac_list in
                    (* < label, jlbl2, invld, invld > *)
                    let tac_list = append_tac_list Tlabel jlbl2
                        dest_src_invld dest_src_invld tac_list in
                    (env, dest_src_invld, tac_list)
          | Rcbcall(rl) ->
                    let (env, seed) = get_seed env in
                    let rec determine_rcb seed rl sum =
                        if rl == [] then (0, "invld")
                        else let r = List.hd rl in
```

```
                                if (sum + (fst r) > seed) then (1, (snd r))
                                else determine_rcb seed (List.tl rl)
                                        (sum + (fst r))
                        in
                        let (use, n) = determine_rcb seed rl 0 in
                        (* if we are not calling an RCB, return *)
                        if use == 0 then (env, dest_src_invld, tac_list)
                        else let rsymb =
                                (if NameMap.mem n gsymb then NameMap.find n gsymb
                                else raise (Failure ("undeclared rcb " ^ n))) in
                        let callsrc = {
                                tds_type = Dvar;
                                tds_name = n;
                                tds_value = 0;
                                tds_size = 0 } in
                        (* this becomes just a call to the RCB *)
                        (*
                         * even an RCB must specify a tmp return - even though
                         * it wont use it.
                         *)
                        let (rdest, env) = get_trslt_dest env 4 in
                        (* < call, return_dest, call_src, invld > *)
                        let tac_list = append_tac_list Tcall rdest
                                callsrc dest_src_invld tac_list in
                        (env, dest_src_invld, tac_list)
          | Nostmt -> (env, dest_src_invld, tac_list)


let rec tac_eval_stmt_list env gsymb lsymb sl tac_list =
        if sl == [] then (env, dest_src_invld, tac_list)
        else
                let s = List.hd sl in
                let (env, _, tac_list) =
                        tac_eval_stmt env gsymb lsymb s
                        tac_list in
                let (env, _, tac_list) =
                        tac_eval_stmt_list env gsymb lsymb
                                (List.tl sl) tac_list in
                (env, dest_src_invld, tac_list)


(************************************************************
 *
 * Routines for Generating TAC Elements at the Function+ Level
 *
 ************************************************************)

(*
 * create_func
 *
 *      Create a function using the function type defined in
 *      tac_types.ml.  This will populate a function with
 *      the function symbol, globals, locals, and a list of
 *      TAC elements representing the function.
 *
 * Output:  (env, function type)
 *)
let create_func env gsymb fin =
        let n = fin.fdef_name in
        let fsymb =
                (if NameMap.mem n gsymb then NameMap.find n gsymb
                 else raise (Failure ("undeclared function " ^ n))) in
        let lsymb = NameMap.empty in
        let lsymb = populate_symbols fin.fdef_vdecls [] [] fin.fdef_params lsymb
        in
```

```
        let (env, _, tac_list) = tac_eval_stmt_list env gsymb lsymb
                                        fin.fdef_body [] in
        let tac_list = List.rev tac_list in
        let func_out = {
                func_symb = fsymb;
                func_globals = gsymb;
                func_locals = lsymb;
                func_ltac = tac_list } in
        (env, func_out)


(*
 * create_rcb
 *
 *      Create an RCB from the AST provided by the parser.
 *
 * Output:  (env, RCB type)
 *)
let create_rcb env gsymb rin =
        let n = rin.rcbdef_name in
        let rsymb =
                (if NameMap.mem n gsymb then NameMap.find n gsymb
                else raise (Failure ("undeclared rcb " ^ n))) in
        let lsymb = NameMap.empty in
        let lsymb = populate_symbols rin.rcbdef_vdecls [] [] [] lsymb in
        let (env, _, tac_list) = tac_eval_stmt_list env gsymb lsymb
                                        rin.rcbdef_body [] in
        let tac_list = List.rev tac_list in
        let rcb_out = {
                rcb_symb = rsymb;
                rcb_globals = gsymb;
                rcb_locals = lsymb;
                rcb_ltac = tac_list } in
        (env, rcb_out)


(*
 * tac_create_prog
 *
 *      Generate an entire program from the tupple generated by the
 *      parser.  This routine is the top-level call which actually
 *      evaluates and provides Three Address Code representation for
 *      all elements.
 *
 * Output:  (program)
 *)
let one_of_six = (fun (x, _, _, _, _, _) -> x)
let two_of_six = (fun (_, x, _, _, _, _) -> x)
let three_of_six = (fun (_, _, x, _, _, _) -> x)
let four_of_six = (fun (_, _, _, x, _, _) -> x)
let five_of_six = (fun (_, _, _, _, x, _) -> x)
let six_of_six = (fun (_, _, _, _, _, x) -> x)

let tac_create_prog pin seed =
        let env = {
                te_label = 0;
                te_trslt = 0;
                te_seed = seed;
                te_group = 0
                } in
        let gsymb = NameMap.empty in
        let gsymb = populate_symbols (one_of_six pin) (two_of_six pin)
                        (three_of_six pin) [] gsymb in
        let rec create_rcbs env rcbl_in rcbl_out =
                if rcbl_in == [] then (env, rcbl_out)
```

```
        else let (env, curr_rcb) = create_rcb env gsymb
                                            (List.hd rcbl_in) in
        create_rcbs env (List.tl rcbl_in) (curr_rcb :: rcbl_out)
in
let (env, rcbs) = create_rcbs env (six_of_six pin) [] in
let rec create_funcs env fl_in fl_out =
        if fl_in == [] then (env, fl_out)
        else let (env, curr_f) = create_func env gsymb
                                            (List.hd fl_in) in
        create_funcs env (List.tl fl_in) (curr_f :: fl_out)
in
let (env, funcs) = create_funcs env (five_of_six pin) [] in
let prog_out = {
        prog_globals = gsymb;
        prog_lrcb = rcbs;
        prog_lfunc = funcs } in
(prog_out)
```

```
open Tac_types


module IntMap = Map.Make(struct
        type t = int
        let compare x y = Pervasives.compare x y
        end)

type asmgen_env = {
        age_implabel : int;
        age_soff : int;          (* stack offset *)
        age_doff : int;          (* data segment offset *)
        age_nlocals : int        (* number of locals in a function *)
        }

type register_resource = Eax | Ebx | Ecx | Edx | Reg_none
type symbol_alias_type = Sa_tmp | Sa_var | Sa_invld
type symbol_alias_location = Sa_loc_stack | Sa_loc_data | Sa_loc_invld

type symbol_alias_element = {
        sat_name : string;
        sat_type : symbol_alias_type;
        sat_size : int;
        sat_reg : register_resource;
        sat_loc : symbol_alias_location;
        sat_soff : int;
        sat_nextuse : int
        }

let invld_sat_symb = {
        sat_name = "invld";
        sat_type = Sa_invld;
        sat_size = 0;
        sat_reg = Reg_none;
        sat_loc = Sa_loc_invld;
        sat_soff = 0;
        sat_nextuse = 0
        }

type register_status = Rs_free | Rs_used

type register_table = {
        rt_eax : string * register_status;
        rt_ebx : string * register_status;
        rt_ecx : string * register_status;
        rt_edx : string * register_status;
        rt_nextevict : register_resource
        }

type temporary_table = {
        tt_base : int;           (* base of temporaries with respect to ss:[ebp] *)
        tt_free : int list       (* free temp stack spots (not including base) *)
        }
```

```
open Asm_types
open Tac_types


(* print the input sat *)
let print_sat sat =
        let lsate = NameMap.fold (fun k v l -> v :: l) sat [] in
        let rec print_lsate lsate =
                if lsate == [] then (print_endline "Sat Table Complete")
                else
                        begin
                        let sate = List.hd lsate in
                        let n_str = sate.sat_name in
                        let t_str =
                                (match sate.sat_type with
                                    Sa_tmp -> "tmp"
                                  | Sa_var -> "var"
                                  | Sa_invld -> "invld") in
                        let r_str =
                                (match sate.sat_reg with
                                    Eax -> "eax"
                                  | Ebx -> "ebx"
                                  | Ecx -> "ecx"
                                  | Edx -> "edx"
                                  | Reg_none -> "none") in
                        let l_str =
                                (match sate.sat_loc with
                                    Sa_loc_stack -> "stack"
                                  | Sa_loc_data -> "data"
                                  | Sa_loc_invld -> "invld") in
                        let o_str = (string_of_int sate.sat_soff) in
                        let nu_str = (string_of_int sate.sat_nextuse) in
                        print_endline ("\tNAME: " ^ n_str ^ " TYPE: " ^ t_str ^
                                " REG: " ^ r_str ^ " LOC: " ^ l_str ^
                                " OFFSET: " ^ o_str ^ " NEXTUSE: " ^
                                nu_str);
                        print_lsate (List.tl lsate)
                        end
        in
        let _ = print_endline ("PRINTING SYMBOL ALIAS TABLE:") in
        let _ = (print_lsate lsate) in
        print_endline ("FINISHED PRINTING SYMBOL ALIAS TABLE")



let string_of_reg_status s =
        match s with
           Rs_free -> "free"
         | Rs_used -> "used"

let string_of_reg r =
        match r with
           Eax -> "eax"
         | Ebx -> "ebx"
         | Ecx -> "ecx"
         | Edx -> "edx"
         | Reg_none -> "none"


let debug_print_rt rt =
       print_endline ("REGISTER TABLE (RT) DEBUG PRINT:" ^
                "\n\tEAX: " ^ (fst rt.rt_eax) ^
                "   STATUS: " ^ (string_of_reg_status (snd rt.rt_eax)) ^
                "\n\tEBX: " ^ (fst rt.rt_ebx) ^
                "   STATUS: " ^ (string_of_reg_status (snd rt.rt_ebx)) ^
```

```
                "\n\tECX: " ^ (fst rt.rt_ecx) ^
                "   STATUS: " ^ (string_of_reg_status (snd rt.rt_ecx)) ^
                "\n\tEDX: " ^ (fst rt.rt_edx) ^
                "   STATUS: " ^ (string_of_reg_status (snd rt.rt_edx)) ^
                "\n\tNEXT EVICT: " ^ (string_of_reg rt.rt_nextevict))
```

```
(*
 * asmgen.ml
 *)

open Tac_types
open Tacgen
open Tac_display
open Asm_types
open Asm_display
open Hex_print

(*
 * This source provides routines that turn a program specified by symbol
 * tables and Three Address Code (TAC) lists into assembly.
 *)

(*
 * sat_name_from_tac_ds
 *
 *      Provide a name for the assembly symbol alias table (SAT) based
 *      on the TAC dest/src provided.
 *
 * Output:  (name)
 *)
let sat_name_from_tac_ds ds =
        if ds.tds_type == Dvar then
                ("_var_" ^ ds.tds_name)
        else if ds.tds_type == Dtmp then
                ("_tmp_" ^ (string_of_int ds.tds_value))
        else if ds.tds_type == Darray then
                ("_arr_" ^ ds.tds_name)
        else
                let _ = print_tac_dest_src ds in
                raise (Failure ("Invalid type for sat name request"))

(*
 * sate_from_tds
 *
 *      Provide a SAT element based on an input TAC dest/src.
 *
 * Output:  (sate)
 *)
let sate_from_tds ds sat =
        let name = sat_name_from_tac_ds ds in
        let sate = (if NameMap.mem name sat then
                (NameMap.find name sat)
                else
                (raise (Failure ("Name not found in sat: " ^
                        name)))) in
        (sate)

(*
 * sat_name_from_tac_symb
 *
 *      Provide a SAT name from a TAC symbol.
 *
 * Output:  (name)
 *)
let sat_name_from_tac_symb symb =
        if symb.symb_stype == Svar then
                ("_var_" ^ symb.symb_name)
        else if symb.symb_stype == Sarray then
                ("_arr_" ^ symb.symb_name)
        else
                raise (Failure ("sat_name_from_tac_symb requests name" ^
```

```
                        " for non var/array: " ^ symb.symb_name))

(*
 * exists_in_future
 *
 *      Determine whether or not a temporary value exists in the
 *      future TAC list.  If it does not, then stack space allocated
 *      for the temporary variable may be reclaimed.
 *
 * Output:  (bool)
 *)
let rec exists_in_future tmpval ltac =
        if ltac == [] then false
        else let ctac = List.hd ltac in
        let destt = ctac.to_dest.tds_type in
        let src1t = ctac.to_src1.tds_type in
        let src2t = ctac.to_src2.tds_type in
        let destv = ctac.to_dest.tds_value in
        let src1v = ctac.to_src1.tds_value in
        let src2v = ctac.to_src2.tds_value in
        if src1t == Dtmp && src1v == tmpval then true
        else if src2t == Dtmp && src2v == tmpval then true
        else if destt == Dtmp && destv == tmpval then true
        else exists_in_future tmpval (List.tl ltac)



(*
 * get_stack_space_for_temps
 *
 *      Determine the amount of stack space required for temporary
 *      values.  This is meant to be used with functions in order to
 *      determine stack management parameters when doing initialization.
 *
 *      NOTE:  We always set the maxinuse to 3 more than returned, this
 *      means that even if no temporaries are used beyond their current
 *      expression, stack space is available for all 3 dest/src if requested.
 *
 * Output: (maxinuse)
 *)
let rec get_stack_space_for_temps ltac inuse maxinuse =
        if ltac == [] then (maxinuse + 3)
        else
                begin
                let tac = List.hd ltac in
                (* if the destination is a tmp, we incr inuse *)
                let dest_is_tmp = (tac.to_dest.tds_type == Dtmp) in
                let dest_future =
                        (exists_in_future tac.to_dest.tds_value (List.tl ltac))
                in
                let inuse = if dest_is_tmp && dest_future then
                        (inuse + 1) else inuse in
                (* dec if a source is not used in future *)
                let src1_is_tmp = (tac.to_src1.tds_type == Dtmp) in
                let src1_future =
                        (exists_in_future tac.to_src1.tds_value (List.tl ltac))
                in
                let inuse = if src1_is_tmp && (src1_future == false) then
                        (inuse - 1) else inuse in
                let src2_is_tmp = (tac.to_src2.tds_type == Dtmp) in
                let src2_future =
                        (exists_in_future tac.to_src2.tds_value (List.tl ltac))
                in
                let inuse = if src2_is_tmp && (src2_future == false) then
                        (inuse - 1) else inuse in
```

```
                (* if the current inuse is greater than the max, set it *)
                let maxinuse = (if inuse > maxinuse then inuse else maxinuse) in
                get_stack_space_for_temps (List.tl ltac) inuse maxinuse
                end


(*************************************************************************
 *
 *              METHODS TO SET UP SYMBOL ALIAS TABLE
 *
 *************************************************************************)

(*
 * add_global_sat_symbols
 *
 *      Add global symbols to the current SAT symbol table.
 *
 * Output:  (sat, env)
 *)
let rec add_global_sat_symbols lsymb sat env =
        if lsymb == [] then (sat, env)
        else let symb = List.hd lsymb in
        let lsymb = List.tl lsymb in
        if symb.symb_stype == Srcb || symb.symb_stype == Sfunc then
                (* functions and rcb's don't take up data space *)
                (add_global_sat_symbols lsymb sat env)
        else if symb.symb_stype == Svar then
                begin
                (* first we create the symbol name *)
                let sname = sat_name_from_tac_symb symb in
                let sat_symb = {
                        sat_name = sname;
                        sat_type = Sa_var;
                        sat_size = symb.symb_size;
                        sat_reg = Reg_none;
                        sat_loc = Sa_loc_data;
                        sat_soff = env.age_doff;
                        sat_nextuse = 0 } in
                (* this variable takes 4 bytes in the data segment *)
                (* increment the environment data offset by 4 *)
                let env = {
                        age_implabel = env.age_implabel;
                        age_soff = env.age_soff;
                        age_doff = env.age_doff + 4;
                        age_nlocals = env.age_nlocals } in
                let sat =
                        (if NameMap.mem sname sat then
                                (raise (Failure ("sat name already appears" ^
                                        " within sat: " ^ sname)))
                        else (NameMap.add sname sat_symb sat)) in
               (add_global_sat_symbols lsymb sat env)
               end
        else if symb.symb_stype == Sarray then
               begin
               (* first create the symbol name *)
               let sname = sat_name_from_tac_symb symb in
               let sat_symb = {
                       sat_name = sname;
                       sat_type = Sa_var;
                       sat_size = symb.symb_size;
                       sat_reg = Reg_none;
                       sat_loc = Sa_loc_data;
                       sat_soff = env.age_doff;
                       sat_nextuse = env.age_nlocals } in
               (*
```

```
                * we reserve an extra 4-bytes of data space following the array
                * in order to allow unaligned accesses of the final bytes.
                *)
               let env = {
                       age_implabel = env.age_implabel;
                       age_soff = env.age_soff;
                       age_doff = env.age_doff + symb.symb_size + 4;
                       age_nlocals = env.age_nlocals } in
               let sat =
                       (if NameMap.mem sname sat then
                                (raise (Failure ("sat name already appears" ^
                                        " within sat: " ^ sname)))
                        else (NameMap.add sname sat_symb sat)) in
               (add_global_sat_symbols lsymb sat env)
               end
       else
                raise (Failure ("unknown type passed to " ^
                        "add_global_sat_symbols"))


(*
 * Local symbols will make use of the stack almost exculsively.  The stack
 * within MOCV will look as follows:
 *
 *      +---------------+
 *      | Parameter1    | <- [ebp+16]
 *      +---------------+
 *      | Parameter2    | <- [ebp+12]
 *      +---------------+
 *      | Parameter3    | <- [ebp+8]
 *      +---------------+
 *      | Return Addr.  | <- [ebp+4]
 *      +---------------+
 *      | Pushed EBP    | <- [ebp],  ebp = esp (following push ebp)
 *      +---------------+
 *      | Local1        | <- [ebp-4]
 *      +---------------+
 *      | Local2        | <- [ebp-8]
 *      +---------------+
 *      | Temp1         | <- [ebp-12]
 *      +---------------+
 *      | Temp2         | <- [ebp-16]
 *      +---------------+
 *           ...
 *
 * During any function call, the code setting up the stack is as follows:
 *
 *      push    ebp
 *      mov     ebp,    esp
 *      sub     esp,    < size of locals + temps >
 *
 *      ; Then at the conclusion of the program, the stack manipulation is
 *      ; as follows
 *
 *      add     esp,    < size of locals + temps >
 *      pop     ebp
 *      ret
 *
 * This should maintain the stack properly regardless of the function calls.
 *)

(*
 * add_local_sat_symbols
 *
 *      Add local symbols to a SAT symbol table.
```

```
 *
 * Output:  (sat, env)
 *)
let rec add_local_sat_symbols lsymb sat env =
        if lsymb == [] then (sat, env)
        else let symb = List.hd lsymb in
        let lsymb = List.tl lsymb in
        if symb.symb_isparam == true then
                begin
                (* this is a parameter, it has a special location *)
                (*
                 * NOTE: In V8086 mode, addresses are alwasy 2 bytes, which
                 * means the parameters start at [ebp+6] and NOT [ebp+8]
                 *)
                let soff = 2 + 4 * symb.symb_paramnum in
                let sname = sat_name_from_tac_symb symb in
                let sat_symb = {
                        sat_name = sname;
                        sat_type = Sa_var;
                        sat_size = symb.symb_size;
                        sat_reg = Reg_none;
                        sat_loc = Sa_loc_stack;
                        sat_soff = soff;
                        sat_nextuse = 0 } in
                let sat =
                        (if NameMap.mem sname sat then
                                (raise (Failure ("sat name already appears" ^
                                        " within sat: " ^ sname)))
                        else (NameMap.add sname sat_symb sat)) in
                (add_local_sat_symbols lsymb sat env)
                end
        else if symb.symb_stype == Srcb || symb.symb_stype == Sfunc then
                (* there shouldn't be any local function decls... *)
                raise (Failure ("function declared locally: " ^ symb.symb_name))
        else if symb.symb_stype == Sarray then
                begin
                (* all arrays are global - so we need to add this globally *)
                let g_lsymb = [symb] in
                let (sat, env) = add_global_sat_symbols g_lsymb sat env in
                (add_local_sat_symbols lsymb sat env)
                end
        else if symb.symb_stype == Svar then
                begin
                (* all local variables go on the stack *)
                let sname = sat_name_from_tac_symb symb in
                let sat_symb = {
                        sat_name = sname;
                        sat_type = Sa_var;
                        sat_size = symb.symb_size;
                        sat_reg = Reg_none;
                        sat_loc = Sa_loc_stack;
                        sat_soff = env.age_soff;
                        sat_nextuse = 0 } in
                (* stack is ebp relative, going down *)
                let sat =
                        (if NameMap.mem sname sat then
                                (raise (Failure ("sat name already appears" ^
                                        " within sat: " ^ sname)))
                        else (NameMap.add sname sat_symb sat)) in
                (* the number of locals goes up by one, stack offset down by 4 *)
                let env = {
                        age_implabel = env.age_implabel;
                        age_soff = env.age_soff - 4;
                        age_doff = env.age_doff;
                        age_nlocals = env.age_nlocals + 1 } in
```

```
                (add_local_sat_symbols lsymb sat env)
                end
        else
                raise (Failure ("unknown symbol type in add_local_sat_symbols"))

(*
 * add_tmp_sat_symbols
 *
 *      Add temporary symbols to a SAT symbol table.
 *
 * Output: (sat)
 *)
let rec add_tmp_sat_symbols sat tt ltac =
        if ltac == [] then sat
        else
                begin
                let tac = List.hd ltac in
                (* if dest is tmp, add it to the table *)
                let dest_is_tmp = (tac.to_dest.tds_type == Dtmp) in
                let in_table = (if (dest_is_tmp == false) then false
                        else (NameMap.mem
                        (sat_name_from_tac_ds tac.to_dest) sat)) in
                let (sat, tt) =
                       (if (dest_is_tmp && (in_table == false)) then
                              begin
                              let nu = 0 in
                              let n = sat_name_from_tac_ds tac.to_dest in
                              let t = Sa_tmp in
                              let r = Reg_none in
                              let l = Sa_loc_stack in
                              let o = ((List.hd tt.tt_free) + tt.tt_base) in
                              let sate = {
                                      sat_name = n;
                                      sat_type = t;
                                      sat_size = 4; (* TODO *)
                                      sat_reg = r;
                                      sat_loc = l;
                                      sat_soff = 0;
                                      sat_nextuse = nu } in
                              let sat =
                                      (if NameMap.mem n sat then
                                              (raise (Failure ("name already " ^
                                                "exists in map: " ^ n)))
                                      else NameMap.add n sate sat) in
                              (* space allocated only if used in future *)
                              let ttfree =
                                      (if nu != 0 then (List.tl tt.tt_free)
                                      else tt.tt_free) in
                              let tt = {
                                      tt_base = tt.tt_base;
                                      tt_free = ttfree } in
                              (sat, tt)
                              end
                       else (sat, tt)) in
                (* check the sources for something to free *)
                let update_tt_for_source sat tt ltac src =
                        let src_is_tmp = (src.tds_type == Dtmp) in
                        let tt =
                                (if src_is_tmp then
                                        begin
                                        (* de-allocate if not used again *)
                                        let nu = 0 in
                                        let n = sat_name_from_tac_ds src in
                                        let tt =
                                            (if nu == 0 then
```

```
                                                let sate =
                                                        (if NameMap.mem n sat
                                                        then (NameMap.find n sat)
                                                        else
                                                        raise (Failure ("Could not fin
d name: " ^
                                                        n))) in
                                                let o = sate.sat_soff in
                                                let o = o - tt.tt_base in
                                                (* add it to free list *)
                                                let ttfree = o :: tt.tt_free in
                                                let tt = {
                                                        tt_base = tt.tt_base;
                                                        tt_free = ttfree } in
                                                (tt)
                                            else (tt)) in
                                        (tt)
                                        end
                                else (tt)) in
                        (tt)
                in
                let tt = update_tt_for_source sat tt (List.tl ltac) tac.to_src1
                in
                let tt = update_tt_for_source sat tt (List.tl ltac) tac.to_src2
                in
                add_tmp_sat_symbols sat tt (List.tl ltac)
                end


(*
 * generate_local_init_assembly
 *
 *      If any of the local variables were declared initialized,
 *      provide assembly code to perform the initialization.
 *
 * Output:  (lasm)
 *)
let rec generate_local_init_assembly sat lsymb lasm =
        if lsymb == [] then lasm
        else
                begin
                let symb = List.hd lsymb in
                let lsymb = List.tl lsymb in
                let sname = sat_name_from_tac_symb symb in
                let satsymb =
                        (if NameMap.mem sname sat then NameMap.find sname sat
                        else raise (Failure ("generate_local_init_assembly:" ^
                                " symbol not found: " ^ sname))) in
                if symb.symb_init != 0 then
                        begin
                        let off = satsymb.sat_soff in
                        let off_str =
                              (if satsymb.sat_soff > 0 then
                                    ("+" ^ (string_of_int off))
                              else (string_of_int off)) in
                        let asm = ("\tmov\tdword ptr ss:[ebp" ^ off_str ^
                                "],\t" ^ (string_of_int symb.symb_init)) in
                        let lasm = asm :: lasm in
                        (generate_local_init_assembly sat lsymb lasm)
                        end
                else (generate_local_init_assembly sat lsymb lasm)
                end


(*
```

```
 * update_rt
 *
 *      Update the Register Table based on information provided.
 *
 * Output:  (Register Table)
 *)
let update_rt reg symb rt free =
        if free then
                match reg with
                  Eax -> { rt_eax = ("invld", Rs_free);
                           rt_ebx = rt.rt_ebx;
                           rt_ecx = rt.rt_ecx;
                           rt_edx = rt.rt_edx;
                           rt_nextevict = Ebx }
                | Ebx -> { rt_eax = rt.rt_eax;
                           rt_ebx = ("invld", Rs_free);
                           rt_ecx = rt.rt_ecx;
                           rt_edx = rt.rt_edx;
                           rt_nextevict = Ecx }
                | Ecx -> { rt_eax = rt.rt_eax;
                           rt_ebx = rt.rt_ebx;
                           rt_ecx = ("invld", Rs_free);
                           rt_edx = rt.rt_edx;
                           rt_nextevict = Edx }
                | Edx -> { rt_eax = rt.rt_eax;
                           rt_ebx = rt.rt_ebx;
                           rt_ecx = rt.rt_ecx;
                           rt_edx = ("invld", Rs_free);
                           rt_nextevict = Eax }
                | Reg_none -> raise (Failure ("Request made to free Reg_non"))
        else
                match reg with
                  Eax -> { rt_eax = (symb.sat_name, Rs_used);
                           rt_ebx = rt.rt_ebx;
                           rt_ecx = rt.rt_ecx;
                           rt_edx = rt.rt_edx;
                           rt_nextevict = rt.rt_nextevict }
                | Ebx -> { rt_eax = rt.rt_eax;
                           rt_ebx = (symb.sat_name, Rs_used);
                           rt_ecx = rt.rt_ecx;
                           rt_edx = rt.rt_edx;
                           rt_nextevict = rt.rt_nextevict }
                | Ecx -> { rt_eax = rt.rt_eax;
                           rt_ebx = rt.rt_ebx;
                           rt_ecx = (symb.sat_name, Rs_used);
                           rt_edx = rt.rt_edx;
                           rt_nextevict = rt.rt_nextevict }
                | Edx -> { rt_eax = rt.rt_eax;
                           rt_ebx = rt.rt_ebx;
                           rt_ecx = rt.rt_ecx;
                           rt_edx = (symb.sat_name, Rs_used);
                           rt_nextevict = rt.rt_nextevict }
                | Reg_none -> raise (Failure ("Request made to free Reg_non"))

(*
 * string_of_reg
 *
 *      Based on a register input and a size, provide a string which
 *      represents the requested register.
 *
 * Output:  (register name)
 *)
let string_of_reg reg sz =
        let str = (if sz == 4 then "e" else "") in
        let str =
```

```
                (match reg with
                  Eax -> (str ^ "a")
                | Ebx -> (str ^ "b")
                | Ecx -> (str ^ "c")
                | Edx -> (str ^ "d")
                | Reg_none -> raise (Failure ("String_of_reg with Reg_none")))
        in
        let str = (if sz == 1 then (str ^ "l") else (str ^ "x")) in
        str

(*
 * reg_to_mem
 *
 *      Generate assembly to put the contents of a register into
 *      memory.
 *
 * Output:  (lasm)
 *)
let reg_to_mem symb lasm =
        if symb.sat_loc == Sa_loc_stack then
                let imm = symb.sat_soff in
                let imm_str = (if imm > 0 then
                        ("+" ^ (string_of_int imm))
                        else (string_of_int imm)) in
                let asm_str = ("\tmov\tss:[ebp" ^ imm_str ^
                        "],\t" ^ (string_of_reg symb.sat_reg symb.sat_size)) in
                let lasm = asm_str :: lasm in
                (lasm)
        else if symb.sat_loc == Sa_loc_data then
                let imm = symb.sat_soff in
                let imm_str = (if imm < 0 then
                       (raise (Failure ("Data segment immediate " ^
                        "is less than zero!  " ^ symb.sat_name)))
                        else (string_of_int symb.sat_soff)) in
                let asm_str = ("\tmov\tds:[" ^ imm_str ^
                        "],\t" ^ (string_of_reg symb.sat_reg symb.sat_size)) in
                let lasm = asm_str :: lasm in
                (lasm)
        else
                raise (Failure ("Location of reg_to_mem " ^
                        "is invalid: " ^ symb.sat_name))


(*
 * evict_reg
 *
 *      Evict a particular register, moving the contents from the
 *      register back into memory if necessary.
 *
 * Output:  (sat, rt, tt, lasm)
 *)
let evict_reg reg sat rt tt lasm =
        (* see if the register is already free - if so, return *)
        let reg_status =
                match reg with
                  Eax -> snd rt.rt_eax
                | Ebx -> snd rt.rt_ebx
                | Ecx -> snd rt.rt_ecx
                | Edx -> snd rt.rt_edx
                | Reg_none -> raise (Failure ("Reg_none passed into evict_reg"))
        in
        if reg_status == Rs_free then (sat, rt, tt, lasm)
        (* get the symbol name for the current reg entry *)
        else let pname =
                match reg with
```

```
                  Eax -> fst rt.rt_eax
                | Ebx -> fst rt.rt_ebx
                | Ecx -> fst rt.rt_ecx
                | Edx -> fst rt.rt_edx
                | Reg_none -> raise (Failure ("Reg_none passed into evict_reg"))
        in
        if ((String.compare pname "invld") == 0) then
                (* this is invalid - must be immediate, just evict *)
                let rt = update_rt reg invld_sat_symb rt true in
                (sat, rt, tt, lasm)
        else
        let psymb =
                (if NameMap.mem pname sat then (NameMap.find pname sat)
                else raise (Failure ("Could not find psymb: " ^ pname))) in
        (* Make sure reg in symbol matches the passed in reg *)
        let _ = (if psymb.sat_reg != reg then
                        let _ = debug_print_rt rt in
                        let _ = print_sat sat in
                        (raise (Failure ("Symbol reg (" ^ (string_of_reg
                        psymb.sat_reg 4) ^ ") does not equal reg used to obtain it! ("
^
                        (string_of_reg reg 4) ^ ") - " ^ psymb.sat_name)))
                else 0) in
        (* On eviction, we need to:
         *      1.  Move symbol from memory to its proper location
         *      2.  Set symbols stack offset properly
         *      3.  Set symbols register to invalid
         *      4.  Clear the Register Table setting
         *)
        if (psymb.sat_type == Sa_var ||
            psymb.sat_type == Sa_tmp) then
                (* Place variable back in memory *)
                let lasm = reg_to_mem psymb lasm in
                let nsymb = { sat_name = psymb.sat_name;
                              sat_type = psymb.sat_type;
                              sat_size = psymb.sat_size;
                              sat_reg = Reg_none;
                              sat_loc = psymb.sat_loc;
                              sat_soff = psymb.sat_soff;
                              sat_nextuse = 0 } in
                (* update the SAT *)
                let sat = NameMap.remove pname sat in
                let sat = NameMap.add pname nsymb sat in
                (* Update the RT to show the eviction *)
                let rt = update_rt reg invld_sat_symb rt true in
                (sat, rt, tt, lasm)
        else
                (* invld - we should never see this *)
                raise (Failure ("eviction of an invld resource"))

(*
 * evict_all_reg
 *
 *      Evict all registers.
 *
 * Output:  (sat, rt, tt, lasm)
 *)
let evict_all_reg sat rt tt lasm =
        let (sat, rt, tt, lasm) = evict_reg Eax sat rt tt lasm in
        let (sat, rt, tt, lasm) = evict_reg Ebx sat rt tt lasm in
        let (sat, rt, tt, lasm) = evict_reg Ecx sat rt tt lasm in
        let (sat, rt, tt, lasm) = evict_reg Edx sat rt tt lasm in
        (sat, rt, tt, lasm)
```

```
(*
 * get_reg_resource
 *
 *      Request a register resource.  Request can either be for a
 *      particular register, or may be just a generic request for
 *      any register.
 *
 * Output:  (sat, rt, tt, lasm, reg)
 *)
let get_reg_resource tac_symb tac_list sat rt tt lasm reg_req =
        let name =
                (if (tac_symb.tds_type == Dtmp ||
                     tac_symb.tds_type == Dvar) then
                        (sat_name_from_tac_ds tac_symb)
                else
                        let _ = print_tac_dest_src tac_symb in
                        raise (Failure ("non tmp/var requesting reg"))) in
        let symb =
                (if NameMap.mem name sat then (NameMap.find name sat)
                else
                        begin
                        let _ = print_endline "*** FAILURE:  COULD NOT FIND SYMB:" in
                        let _ = print_sat sat in
                        let _ = debug_print_rt rt in
                        let _ = print_tac_dest_src tac_symb in
                        let _ = print_endline "\tREMAINING TAC LIST" in
                        let _ = print_tac_list tac_list in
                        raise (Failure ("get_reg_resource: could not find symb: " ^
                                name))
                        end) in
        (* if there is a specific register request, we must oblige *)
        if reg_req != Reg_none then
                (* a specific register is requested *)
                (* if this symb is already in that reg, just return *)
                if symb.sat_reg == reg_req then
                        (* we already own the register we want, just return *)
                        (sat, rt, tt, lasm, reg_req)
                else
                (* now evict the register we need *)
                let (sat, rt, tt, lasm) = evict_reg reg_req sat rt tt lasm in
                let reg = reg_req in

                (* if this symb has a register, evict it first *)
                let (sat, rt, tt, lasm) = (if symb.sat_reg != Reg_none then
                        (* we have a register already, evict it *)
                        (evict_reg symb.sat_reg sat rt tt lasm)
                       else (sat, rt, tt, lasm)) in
                let symb = {
                        sat_name = symb.sat_name;
                        sat_type = symb.sat_type;
                        sat_size = symb.sat_size;
                        sat_reg = reg;
                        sat_loc = symb.sat_loc;
                        sat_soff = symb.sat_soff;
                        sat_nextuse = 0
                        } in
                (* remove the previous SAT entry, add the updated one *)
                let sat = NameMap.remove name sat in
                let sat = NameMap.add name symb sat in
                (* update rt to reflect register ownership *)
                let rt = update_rt reg symb rt false in
                (sat, rt, tt, lasm, reg)
        (* if it's already using a register, then great *)
        else if symb.sat_reg != Reg_none then (sat, rt, tt, lasm, symb.sat_reg)
```

```
        (* next, try to allocate a free register *)
        else let reg =
                (if snd rt.rt_eax == Rs_free then Eax
                else if snd rt.rt_ebx == Rs_free then Ebx
                else if snd rt.rt_ecx == Rs_free then Ecx
                else if snd rt.rt_edx == Rs_free then Edx
                else Reg_none) in
        if reg != Reg_none then
                (* there is a free register, use it *)
                let symb = {
                        sat_name = symb.sat_name;
                        sat_type = symb.sat_type;
                        sat_size = symb.sat_size;
                        sat_reg = reg;
                        sat_loc = symb.sat_loc;
                        sat_soff = symb.sat_soff;
                        sat_nextuse = 0
                        } in
                (* update the rt *)
                let rt = update_rt reg symb rt false in
                (* remove the previous SAT entry, add the updated one *)
                let sat = NameMap.remove name sat in
                let sat = NameMap.add name symb sat in
                (sat, rt, tt, lasm, reg)
        else
                (* there are no free registers, we need to take one *)
                let reg = rt.rt_nextevict in
                (* first evict the current contents of register *)
                let (sat, rt, tt, lasm) = evict_reg reg sat rt tt lasm in
                (* create updated symbol with register information *)
                let symb = {
                        sat_name = symb.sat_name;
                        sat_type = symb.sat_type;
                        sat_size = symb.sat_size;
                        sat_reg = reg;
                        sat_loc = symb.sat_loc;
                        sat_soff = symb.sat_soff;
                        sat_nextuse = 0
                        } in
                (* update the rt *)
                let rt = update_rt reg symb rt false in
                (* update the symbol alias table *)
                let sat = NameMap.remove name sat in
                let sat = NameMap.add name symb sat in
                (sat, rt, tt, lasm, reg)

(*
 * get_src_string
 *
 *      Get a string that represents a TAC dest/src type in assembly.
 *
 * Output: (source string)
 *)
let get_src_string src sat reg_size =
        if src.tds_type == Darray then
                begin
                (* the correct DS offset is in the sat sat_soff *)
                let src_n = sat_name_from_tac_ds src in
                let sate =
                        (if NameMap.mem src_n sat then NameMap.find src_n sat
                        else raise (Failure ("Name not found in sat: " ^
                                src_n))) in
                (string_of_int sate.sat_soff)
                end
        else if src.tds_type == Dimm then (string_of_int src.tds_value)
```

```
        else
        let src_n = sat_name_from_tac_ds src in
        let sate =
                (if NameMap.mem src_n sat then NameMap.find src_n sat
                else raise (Failure ("Name not found in sat: " ^ src_n)))
        in
        let sz = src.tds_size in
        (*
         * Three cases:
         *      1.  It's already in a register (just use it)
         *      2.  It's a tmp and it's on the stack
         *      3.  It's a var and it's on the stack
         *)
        if sate.sat_reg != Reg_none then
                (* it's already in a reg - perfect *)
                let sz = (if reg_size != 0 then reg_size
                        else sz) in
                (string_of_reg sate.sat_reg sz)
        else
                let ptr = (if sz == 1 then "byte ptr "
                        else if sz == 2 then "word ptr "
                        else if sz == 4 then "dword ptr "
                        else raise (Failure ("get_src_string: sz = " ^
                                        (string_of_int sz)))) in
                let imm = sate.sat_soff in
                let imm_str =
                        (if imm > 0 then ("+" ^ (string_of_int imm))
                        else (string_of_int imm)) in
                let imm_str_data = string_of_int imm in
                (* it's not in a register... *)
                if sate.sat_type == Sa_tmp || sate.sat_type == Sa_var then
                        if sate.sat_loc == Sa_loc_stack then
                                (* it's on the stack *)
                                (ptr ^ "ss:[ebp" ^ imm_str ^ "]")
                        else if sate.sat_loc == Sa_loc_data then
                                (* it's in the data segment *)
                                (ptr ^ "ds:[" ^ imm_str_data ^ "]")
                        else
                                raise (Failure ("unknown memory location!"))
                else raise (Failure ("non tmp/var asks for src string " ^
                                        sate.sat_name))

(*
 * place_in_reg
 *
 *      Place a TAC dest/src type into a register.  This is required
 *      for instructions that must be executed on at least one
 *      register operand.
 *
 * Output:  (reg, sat, rt, tt, lasm)
 *)
let place_in_reg tds ltac sat rt tt lasm =
        match tds.tds_type with
           Dvar | Dtmp -> let n = sat_name_from_tac_ds tds in
                let symb = (if NameMap.mem n sat then (NameMap.find n sat)
                        else (raise (Failure ("Name not found in sat " ^ n))))
                in
                if symb.sat_reg != Reg_none then
                        (symb.sat_reg, sat, rt, tt, lasm)
                else
                let src_str = get_src_string tds sat 0 in
                let (sat, rt, tt, lasm, reg) =
                        get_reg_resource tds ltac sat rt tt lasm Reg_none in
                (* get the string for the first source operand *)
                let dest_str = string_of_reg reg tds.tds_size in
```

```
                (* perform the assignment *)
                let asm = ("\tmov\t" ^ dest_str ^ ",\t" ^ src_str) in
                let lasm = asm :: lasm in
                (reg, sat, rt, tt, lasm)
          | Dimm -> (* an immediate, we must move it into a reg *)
                let reg = rt.rt_nextevict in
                let (sat, rt, tt, lasm) =
                       evict_reg reg sat rt tt lasm in
                let asm = ("\tmov\t" ^ (string_of_reg reg tds.tds_size) ^
                        ",\t" ^ (string_of_int tds.tds_value)) in
                let lasm = asm :: lasm in
                (reg, sat, rt, tt, lasm)
          | Darray -> raise (Failure ("source input is an array"))

(*
 * string_from_tacop
 *
 *      Get a string that represents a TAC operation in assembly.  Note
 *      that not all TAC operations have valid string representations
 *      within assembly.
 *
 * Output:  (string)
 *)
let string_from_tacop op =
       match op with
          Tadd -> "add"
        | Tsub -> "sub"
        | Tmul -> "mul"
        | Tdiv -> "div"
        | Trdiv -> "div"
        | Txor -> "xor"
        | Tor -> "or"
        | Tand -> "and"
        | Teq -> "je"
        | Tne -> "jne"
        | Tlt -> "jl"
        | Tgt -> "jg"
        | Tle -> "jle"
        | Tge -> "jge"
        | Tuminus -> "neg"
        | Tnot -> "not"
        | Tjmp -> "jmp"
        | Tjmpe -> "je"
        | Tjmpne -> "jne"
        | Tjmpgt -> "jg"
        | Tjmplt -> "jl"
        | Tjmpge -> "jge"
        | Tjmple -> "jle"
        | Tlabel -> "L"
        | Tarray -> "invld"
        | Tasn -> "mov"
        | Taasn -> "invld"
        | Tcmp -> "cmp"
        | Tret -> "ret"
        | Tparam -> "invld"
        | Tcall -> "call"
        | Tevictall -> "invld"
        | Tespadd -> "add"
        | Tevicteax -> "invld"
        | Tevictebx -> "invld"
        | Tevictecx -> "invld"
        | Tevictedx -> "invld"


(*
```

```
 * binary_op
 *
 *      Provide assembly for a binary TAC operation.
 *
 * The supported ops are:
 *
 *      add,    sub,    xor,    or,     and
 *
 *
 * Intended assembly for binary operations
 *
 *      mov     dest,   src1
 *      op      dest,   src2
 *
 * Output: (lasm, sat, rt, tt, env)
 *)
let binary_op ltac lasm sat rt tt env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let dest = tac_el.to_dest in
        let src1 = tac_el.to_src1 in
        let src2 = tac_el.to_src2 in
        (* first get the register for the temporary *)
        let (sat, rt, tt, lasm, reg) =
               get_reg_resource dest ltac sat rt tt lasm Reg_none in
        (* get the string for the first source operand *)
        let src1_str = get_src_string src1 sat 0 in
        (* get the string for the second source operand *)
        let src2_str = get_src_string src2 sat 0 in
        (* get the string for the destination register *)
        let dest_str = string_of_reg reg dest.tds_size in
        (* get the string for the operation *)
        let op_str = string_from_tacop op in
        (* finally, create the assembly string and add it to the list *)
        let asm = ("\tmov\t" ^ dest_str ^ ",\t" ^ src1_str) in
        let lasm = asm :: lasm in
        let asm = ("\t" ^ op_str ^ "\t" ^ dest_str ^ ",\t" ^ src2_str) in
        let lasm = asm :: lasm in
        (lasm, sat, rt, tt, env)


(*
 * comparison_op
 *
 *      Provide assembly for a TAC comparison operation.
 *
 * The supported comparison operations are
 *
 *      eq,     ne,     lt,     gt,     le,     ge
 *
 * The intended assembly for these operations is as follows
 *
 *              cmp     src1,   src2
 *              j_op    IL#1
 *              mov     dest,   0
 *              jmp     IL#2
 *      IL#1:   mov     dest,   1
 *      IL#2:
 *
 * NOTE:  The reason for this complexity is that some users may wish
 * to use the result of a comparison in an expression.  The x86 architecture
 * does comparisons through flags, so in order to move the result into a
 * temporary register, we need to add this additional complexity.
 *
 * Output:  (lasm, sat, rt, tt, env)
```

```
 *)
let comparison_op ltac lasm sat rt tt env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let dest = tac_el.to_dest in
        let src1 = tac_el.to_src1 in
        let src2 = tac_el.to_src2 in
        (* first get the register for the temporary *)
        let (sat, rt, tt, lasm, reg) =
               get_reg_resource dest ltac sat rt tt lasm Reg_none in
        (* get the string for the first source operand *)
        (* if the source is not in a reg, we need to move it there *)
        (* if the source is an immediate, we need to move it to a reg *)
        let (regs1, sat, rt, tt, lasm) =
                place_in_reg src1 ltac sat rt tt lasm in
        let _ = (if regs1 != reg then 1
                else raise (Failure("comparison_op: tmp reg and src1 reg " ^
                        "are the same!"))) in
        let src1_str = string_of_reg regs1 src1.tds_size in
        (* get the string for the second source operand *)
        let src2_str = get_src_string src2 sat 0 in
        (* get the string for the destination register *)
        let dest_str = string_of_reg reg dest.tds_size in
        (* get the string for the operation *)
        let jcmp_str = string_from_tacop op in
        (* create the implicit label numbers *)
        let il1 = env.age_implabel in
        let env = {
                age_implabel = env.age_implabel + 1;
                age_soff = env.age_soff;
                age_doff = env.age_doff;
                age_nlocals = env.age_nlocals } in
        let il2 = env.age_implabel in
        let env = {
                age_implabel = env.age_implabel + 1;
                age_soff = env.age_soff;
                age_doff = env.age_doff;
                age_nlocals = env.age_nlocals } in
        (* finally, create the assembly string and add it to the list *)
        let asm = ("\tcmp\t" ^ src1_str ^ ",\t" ^ src2_str) in
        let lasm = asm :: lasm in
        let asm = ("\t" ^ jcmp_str ^ "\t" ^ "IL" ^ (string_of_int il1)) in
        let lasm = asm :: lasm in
        let asm = ("\tmov\t" ^ dest_str ^ ",\t0") in
        let lasm = asm :: lasm in
        let asm = ("\tjmp\tIL" ^ (string_of_int il2)) in
        let lasm = asm :: lasm in
        let asm = ("IL" ^ (string_of_int il1) ^ ":\tmov\t" ^ dest_str ^ ",\t1")
        in
        let lasm = asm :: lasm in
        let asm = ("IL" ^ (string_of_int il2) ^ ":") in
        let lasm = asm :: lasm in
        (lasm, sat, rt, tt, env)


(*
 * jmp_op
 *
 *      Provide assembly for a jump operation.
 *
 * Jump operations supported are as follows:
 *
 *      jmp,    je,     jne,    jg,     jl      jge,    jle
 *
 * The intended assembly for this is as follows:
```

```
 *
 *      j_op    dest_label
 *
 * Output:  (lasm, sat, rt, tt, env) *)
let jmp_op ltac lasm sat rt tt env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let dest = tac_el.to_dest in
        (* get the name of the label we are jumping too *)
        let l_n = ("L" ^ (string_of_int (dest.tds_value))) in
        (* get the string of the jmp operation being performed *)
        let op_str = string_from_tacop op in
        (* generate the assembly *)
        let asm = ("\t" ^ op_str ^ "\t" ^ l_n) in
        let lasm = asm :: lasm in
        (lasm, sat, rt, tt, env)


(*
 * unary_op
 *
 *      Provide assembly for a unary operation.
 *
 * Supported unary operations are as follows:
 *
 *      not,    neg
 *
 * The intended assembly is:
 *
 *      mov     dest,   src
 *      op      dest
 *
 * Output: (lasm, sat, rt, tt, env)
 *)
let unary_op ltac lasm sat rt tt env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let dest = tac_el.to_dest in
        let src = tac_el.to_src1 in
        (* first get the register for the temporary *)
        let (sat, rt, tt, lasm, reg) =
                get_reg_resource dest ltac sat rt tt lasm Reg_none in
        (* get the string for the first source operand *)
        let src_str = get_src_string src sat 0 in
        (* get the string for the destination register *)
        let dest_str = string_of_reg reg dest.tds_size in
        (* get the string for the operation *)
        let op_str = string_from_tacop op in
        (* generate the assembly *)
        let asm = ("\tmov\t" ^ dest_str ^ ",\t" ^ src_str) in
        let lasm = asm :: lasm in
        let asm = ("\t" ^ op_str ^ "\t" ^ dest_str) in
        let lasm = asm :: lasm in
        (lasm, sat, rt, tt, env)

(*
 * div_op
 *
 *      Provide assembly for a divide operation.
 *
 * Supported divide operations are as follows:
 *
 *      div,    div (remainder)
 *
 *
```

```
 * The tac op is as follows:  < div / rdiv, dest, src1, src2 >
 * Intended assembly is as follows:
 *
 *      8-bit division:  AX / src = AL (rem AH)
 *      16-bit division:  DX:AX / src = AX (rem DX)
 *      32-bit division:  EDX:EAX / src = EAX (rem EDX)
 *
 *      < evict EAX >
 *      < evict EDX >
 *      mov     RAX,    src1
 *      div     src2
 *      mov     dest,   REGA ; if div
 *      mov     dest,   REGD ; if rdiv
 *
 * Output: (lasm, sat, rt, tt, env)
 *)
let div_op ltac lasm sat rt tt env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let dest = tac_el.to_dest in
        let src1 = tac_el.to_src1 in
        let src2 = tac_el.to_src2 in

        (* evict the contents of both EAX and EDX *)
        let (sat, rt, tt, lasm) = evict_reg Eax sat rt tt lasm in
        let (sat, rt, tt, lasm) = evict_reg Ebx sat rt tt lasm in
        let (sat, rt, tt, lasm) = evict_reg Edx sat rt tt lasm in

        (* need to take src1 argument before the evictions *)
        let src1_str = get_src_string src1 sat 0 in

        (* get the destination and its string - request ebx *)
        let (sat, rt, tt, lasm, reg) = get_reg_resource dest
                ltac sat rt tt lasm Ebx in

        (* get the source string for src2 *)
        let src2_str = get_src_string src2 sat 0 in

        (*
         * Virtual 8086 mode displays some goofy divide characteristics
         * when the extended register set is used.  Because of this, to be
         * safe, we will zero out all registers used before beginning.
         *)
        let asm = ("\txor\teax,\teax") in
        let lasm = asm :: lasm in
        let asm = ("\txor\tebx,\tebx") in
        let lasm = asm :: lasm in
        let asm = ("\txor\tedx,\tedx") in
        let lasm = asm :: lasm in
        (*
         * NOTE:  We do not change the sources to reflect where they are moved,
         * which means the registers will remain free.  These types of mul/div
         * improvements are a stretch goal if time allows.
         *)
        let sz = src1.tds_size in
        if sz == 1 then
                let asm = ("\tmov\tal,\t" ^ src1_str) in
                let lasm = asm :: lasm in
                let asm = ("\tmov\tbl,\t" ^ src2_str) in
                let lasm = asm :: lasm in
                let asm = ("\tdiv\tbl") in
                let lasm = asm :: lasm in
                if op == Tdiv then
                        let asm = ("\tmov\tbl,\tal") in
                        let lasm = asm :: lasm in
```

```
                        (lasm, sat, rt, tt, env)
                else (* op == Trdiv *)
                        let asm = ("\tmov\tbl,\tah") in
                        let lasm = asm :: lasm in
                        (lasm, sat, rt, tt, env)
        else if sz == 2 then
                let asm = ("\tmov\tax,\t" ^ src1_str) in
                let lasm = asm :: lasm in
                let asm = ("\tmov\tbx,\t" ^ src2_str) in
                let lasm = asm :: lasm in
                let asm = ("\tdiv\tbx") in
                let lasm = asm :: lasm in
                if op == Tdiv then
                        let asm = ("\tmov\tbx,\tax") in
                        let lasm = asm :: lasm in
                        (lasm, sat, rt, tt, env)
                else (* op == Trdiv *)
                        let asm = ("\tmov\tbx,\tdx") in
                        let lasm = asm :: lasm in
                        (lasm, sat, rt, tt, env)
        else (* sz == 4 *)
                let asm = ("\tmov\teax,\t" ^ src1_str) in
                let lasm = asm :: lasm in
                let asm = ("\tmov\tebx,\t" ^ src2_str) in
                let lasm = asm :: lasm in
                let asm = ("\tdiv\tebx") in
                let lasm = asm :: lasm in
                if op == Tdiv then
                        let asm = ("\tmov\tebx,\teax") in
                        let lasm = asm :: lasm in
                        (lasm, sat, rt, tt, env)
                else (* op == Trdiv *)
                        let asm = ("\tmov\tebx,\tedx") in
                        let lasm = asm :: lasm in
                        (lasm, sat, rt, tt, env)


(*
 * mul_op
 *
 *      Provide assembly for a multiply operation.
 *
 * Supported multiply operations:  mul
 *
 * Intended assembly:
 *
 *      < evict eax >
 *      < evict edx > ; ensure destination is routed to REGA
 *      mov     REGA,   src1
 *      mul     src2
 *      ; Perform work to get result into eax
 *
 * Output: (lasm, sat, rt, tt, env) *)
let mul_op ltac lasm sat rt tt env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let dest = tac_el.to_dest in
        let src1 = tac_el.to_src1 in
        let src2 = tac_el.to_src2 in
        (* evict the contents of both EAX and EDX *)
        let (sat, rt, tt, lasm) = evict_reg Eax sat rt tt lasm in
        let (sat, rt, tt, lasm) = evict_reg Edx sat rt tt lasm in
        (* get the destination *)
        let (sat, rt, tt, lasm, reg) = get_reg_resource dest
                ltac sat rt tt lasm Eax in
```

```
        (* get the source strings for src1 and src2 *)
        let src1_str = get_src_string src1 sat 0 in
        let src2_str = get_src_string src2 sat 0 in
        (* dest is REGA of some form - generate assembly *)
        let sz = src1.tds_size in
        if sz == 1 then
                let asm = ("\txor\teax,\teax") in
                let lasm = asm :: lasm in
                let asm = ("\tmov\tal,\t" ^ src1_str) in
                let lasm = asm :: lasm in
                let asm = ("\tmov\tdl,\t" ^ src2_str) in
                let lasm = asm :: lasm in
                let asm = ("\tmul\tdl") in
                let lasm = asm :: lasm in
                (lasm, sat, rt, tt, env)
        else if sz == 2 then
                let asm = ("\txor\teax,\teax") in
                let lasm = asm :: lasm in
                let asm = ("\txor\tedx,\tedx") in
                let lasm = asm :: lasm in
                let asm = ("\tmov\tax,\t" ^ src1_str) in
                let lasm = asm :: lasm in
                let asm = ("\tmov\tdx,\t" ^ src2_str) in
                let lasm = asm :: lasm in
                let asm = ("\tmul\tdx") in
                let lasm = asm :: lasm in
                (lasm, sat, rt, tt, env)
        else (* sz == 4 *)
                let asm = ("\txor\teax,\teax") in
                let lasm = asm :: lasm in
                let asm = ("\tmov\teax,\t" ^ src1_str) in
                let lasm = asm :: lasm in
                let asm = ("\tmov\tedx,\t" ^ src2_str) in
                let lasm = asm :: lasm in
                let asm = ("\tmul\tedx") in
                let lasm = asm :: lasm in
                (lasm, sat, rt, tt, env)


(*
 * assign_op
 *
 *      Provide assembly for an assignment operation.
 *
 * The supported ops are:
 *
 *      mov
 *
 *
 * Intended assembly for binary operations
 *
 *      mov     dest,   src1
 *
 * output: (lasm, sat, rt, tt, env)
 *)
let assign_op ltac lasm sat rt tt env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let dest = tac_el.to_dest in
        let src1 = tac_el.to_src1 in

        (* first get the register for the temporary *)
        let (sat, rt, tt, lasm, reg) =
               get_reg_resource dest ltac sat rt tt lasm Reg_none in
        let dest_str = string_of_reg reg dest.tds_size in
```

```
        (* get the string for the first source operand *)
        let src_str = get_src_string src1 sat 0 in
        (* perform the assignment *)
        let asm = ("\tmov\t" ^ dest_str ^ ",\t" ^ src_str) in
        let lasm = asm :: lasm in
        (lasm, sat, rt, tt, env)


(*
 * array_op
 *
 *      Provide assembly for an array operation.  This includes
 *      only array accesses, not assignments.
 *
 * The supported ops are:
 *
 *      < array access >
 *
 *
 * Intended assembly for binary operations
 *
 *      mov     edi,    < array index >
 *      add     edi,    < array base offset >
 *      mov     dest,   ds:[edi]
 *
 * output: (lasm, sat, rt, tt, env)
 *)
let array_op ltac lasm sat rt tt env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let dest = tac_el.to_dest in
        let src1 = tac_el.to_src1 in
        let src2 = tac_el.to_src2 in
        (* first get the register for the temporary *)
        let (sat, rt, tt, lasm, reg) =
               get_reg_resource dest ltac sat rt tt lasm Reg_none in
        let dest_str = string_of_reg reg dest.tds_size in
        (* get the string for the source operands *)
        let src1_str = get_src_string src1 sat 0 in
        (*
         * source 2 must be considered a 4-byte value if it
         * is in a register - in order to work with edi.
         *)
        let src2_str = get_src_string src2 sat 4 in
        (* generate the assembly *)
        let asm = ("\tmov\tedi,\t" ^ src2_str) in
        let lasm = asm :: lasm in
        let asm = ("\tadd\tedi,\t" ^ src1_str) in
        let lasm = asm :: lasm in
        let asm = ("\tmov\t" ^ dest_str ^ ",\tds:[edi]") in
        let lasm = asm :: lasm in
        (lasm, sat, rt, tt, env)

(*
 * array_asn_op
 *
 *      Provide assembly for an array assignment.
 *
 * The supported ops are:
 *
 *      < array assignment >
 *
 *
 * Intended assembly for binary operations
 *
```

```
 *      mov     edi,    < array index >
 *      add     edi,    < array base offset >
 *      mov     SIZE ptr ds:[edi],      src
 *
 * output: (lasm, sat, rt, tt, env)
 *)
let array_asn_op ltac lasm sat rt tt env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let arr = tac_el.to_dest in
        let index = tac_el.to_src1 in
        let src = tac_el.to_src2 in
        (* get the string for the source *)
        (* if the source is not in a reg, we need to move it there *)
        (* if the source is an immediate, we need to move it to a reg *)
        let (reg, sat, rt, tt, lasm) =
                place_in_reg src ltac sat rt tt lasm in
        let src_str = string_of_reg reg src.tds_size in
        (* get the ptr string *)
        let ptr_str =
                (if arr.tds_value == 1 then ("byte ptr ")
                else if arr.tds_value == 2 then ("word ptr ")
                else if arr.tds_value == 4 then ("dword ptr ")
                else raise (Failure ("invalid array data size: " ^
                        (string_of_int arr.tds_value)))) in
        (* get the string for the array operands *)
        let arr_str = get_src_string arr sat 0 in
        let index_str = get_src_string index sat 4 in
        (* generate the assembly *)
        let asm = ("\tmov\tedi,\t" ^ index_str) in
        let lasm = asm :: lasm in
        let asm = ("\tadd\tedi,\t" ^ arr_str) in
        let lasm = asm :: lasm in
        let asm = ("\tmov\t" ^ ptr_str ^ "ds:[edi],\t" ^ src_str) in
        let lasm = asm :: lasm in
        (lasm, sat, rt, tt, env)

(*
 * claim_tmp_stack_space
 *
 *      Claim temporary stack space for a TAC operation.  This
 *      will temporarily allocate space on the stack for any
 *      temporary values that need to be stored within the expression.
 *
 * Output: (sat, tt)
 *)
let claim_tmp_stack_space sat tt tac_dest =
        if tac_dest.tds_type != Dtmp then (sat, tt)
        else
        let sat_name = sat_name_from_tac_ds tac_dest in
        let sat_symb =
                (if NameMap.mem sat_name sat then (NameMap.find sat_name sat)
                else
                        begin
                        let _ = print_endline " *** FAILURE" in
                        let _ = print_sat sat in
                        let _ = print_tac_dest_src tac_dest in
                        raise (Failure ("claim_tmp_stack_space: name not found " ^
                                sat_name))
                        end) in
        (* if the offset is 0, then it is not using space *)
        if sat_symb.sat_soff != 0 then (sat, tt)
        else
        let off = tt.tt_base + (List.hd tt.tt_free) in
        (* update the temporaries table *)
```

```
        let tt = {
                tt_base = tt.tt_base;
                tt_free = (List.tl tt.tt_free) } in
        let new_symb = {
                sat_name = sat_symb.sat_name;
                sat_type = sat_symb.sat_type;
                sat_size = sat_symb.sat_size;
                sat_reg = sat_symb.sat_reg;
                sat_loc = sat_symb.sat_loc;
                sat_soff = off;
                sat_nextuse = sat_symb.sat_nextuse } in
        (* update the sat *)
        let sat = NameMap.remove sat_name sat in
        let sat = NameMap.add sat_name new_symb sat in
        (sat, tt)


(*
 * release_tmp_stack_space
 *
 *      Release temporary stack space that was previously claimed
 *      for a TAC expression's temporary variables.  This function
 *      checks whether a temporary value is used in the future, and
 *      if not, it releases its stack allocation.
 *
 * Output: (sat, tt, rt)
 *)
let release_tmp_stack_space sat tt rt ltac src =
        if src.tds_type != Dtmp then (sat, tt, rt)
        else
        let in_future = exists_in_future src.tds_value ltac in
        if in_future == true then (sat, tt, rt)
        else
        (* this source is a tmp that does not exist in future *)
        (* release it, give resources back *)
        let sat_name = sat_name_from_tac_ds src in
        let sat_symb =
               (if NameMap.mem sat_name sat then (NameMap.find sat_name sat)
               else raise (Failure ("release_tmp_stack_space:  Name not found "
               ^ sat_name))) in
        let my_free = sat_symb.sat_soff - tt.tt_base in
        (* give space back to tt *)
        let tt = {
                tt_base = tt.tt_base;
                tt_free = my_free :: tt.tt_free } in
        (* if I have a reg, release it in rt *)
        let my_reg = sat_symb.sat_reg in
        let rt = (if my_reg == Reg_none then (rt)
                 else (update_rt my_reg invld_sat_symb rt true)) in
        (* now update the symbol *)
        let new_symb = {
                sat_name = sat_symb.sat_name;
                sat_type = sat_symb.sat_type;
                sat_size = sat_symb.sat_size;
                sat_reg = Reg_none; (* release the register, regardless *)
                sat_loc = sat_symb.sat_loc;
                sat_soff = 0;
                sat_nextuse = sat_symb.sat_nextuse } in
        let sat = NameMap.remove sat_name sat in
        let sat = NameMap.add sat_name new_symb sat in
        (sat, tt, rt)


(*
```

```
 * generate_asm_for_op
 *
 *      Generate assembly for a given TAC operation.  This is
 *      the main assembly generating function for handling a list
 *      of TAC operations.
 *
 * Output: (lasm, sat, rt, tt, env)
 *)
let generate_asm_for_op ltac sat rt tt lasm env =
        let tac_el = List.hd ltac in
        let op = tac_el.to_op in
        let dest = tac_el.to_dest in
        let src1 = tac_el.to_src1 in
        let src2 = tac_el.to_src2 in

        (* we need to get stack space for new tmp variables *)
        let (sat, tt) = claim_tmp_stack_space sat tt dest in

        let (lasm, sat, rt, tt, env) =
        match op with
          Tadd -> (binary_op ltac lasm sat rt tt env)
        | Tsub -> (binary_op ltac lasm sat rt tt env)
        | Tmul -> (mul_op ltac lasm sat rt tt env)
        | Tdiv -> (div_op ltac lasm sat rt tt env)
        | Trdiv -> (div_op ltac lasm sat rt tt env)
        | Txor -> (binary_op ltac lasm sat rt tt env)
        | Tor -> (binary_op ltac lasm sat rt tt env)
        | Tand -> (binary_op ltac lasm sat rt tt env)
        | Teq -> (comparison_op ltac lasm sat rt tt env)
        | Tne -> (comparison_op ltac lasm sat rt tt env)
        | Tlt -> (comparison_op ltac lasm sat rt tt env)
        | Tgt -> (comparison_op ltac lasm sat rt tt env)
        | Tle -> (comparison_op ltac lasm sat rt tt env)
        | Tge -> (comparison_op ltac lasm sat rt tt env)
        | Tuminus -> (unary_op ltac lasm sat rt tt env)
        | Tnot -> (unary_op ltac lasm sat rt tt env)
        | Tjmp -> (jmp_op ltac lasm sat rt tt env)
        | Tjmpe -> (jmp_op ltac lasm sat rt tt env)
        | Tjmpne -> (jmp_op ltac lasm sat rt tt env)
        | Tjmpgt -> (jmp_op ltac lasm sat rt tt env)
        | Tjmplt -> (jmp_op ltac lasm sat rt tt env)
        | Tjmpge -> (jmp_op ltac lasm sat rt tt env)
        | Tjmple -> (jmp_op ltac lasm sat rt tt env)
        | Tlabel -> let asm = ("L" ^ (string_of_int dest.tds_value) ^
                                ":") in
                let lasm = asm :: lasm in
                (lasm, sat, rt, tt, env)
        | Tasn -> (assign_op ltac lasm sat rt tt env)
        | Tarray -> (array_op ltac lasm sat rt tt env)
        | Taasn -> (array_asn_op ltac lasm sat rt tt env)
        | Tcmp -> (* get the string for the first and second source operand *)
                let src2_str = get_src_string src2 sat 0 in
                (* load a register with the first source *)
                let (reg, sat, rt, tt, lasm) =
                        place_in_reg src1 ltac sat rt tt lasm in
                let src1_str = get_src_string src1 sat 0 in
                let asm = ("\tcmp\t" ^ src1_str ^ ",\t" ^ src2_str) in
                let lasm = asm :: lasm in
                (lasm, sat, rt, tt, env)
        | Tret -> (* Note: The source of the return value is actually in dest *)
                let src_str = get_src_string dest sat 0 in
                (* Result will be placed in EAX, regardless of size *)
                let (sat, rt, tt, lasm) = evict_reg Eax sat rt tt lasm in
                (* generate the assembly *)
                let asm = ("\tmov\teax,\t" ^ src_str) in
```

```
                let lasm = asm :: lasm in
                (* the "ret" instruction is handled by the function construct *)
                (lasm, sat, rt, tt, env)
        | Tparam -> (*let (sat, rt, tt, lasm) = evict_reg Eax sat rt tt lasm
                in*)
                (*
                 * params are only pushed before function calls - we need to
                 * evict all registers before calls, so evict here
                 *)
                let (sat, rt, tt, lasm) = evict_all_reg sat rt tt lasm in
                (* get the string for the parameter *)
                let p_str = get_src_string dest sat 0 in
                let asm = ("\tmov\teax,\t" ^ p_str) in
                let lasm = asm :: lasm in
                let asm = ("\tpush\teax") in
                let lasm = asm :: lasm in
                (lasm, sat, rt, tt, env)
        | Tcall -> let (sat, rt, tt, lasm) = evict_all_reg sat rt tt lasm
                in
                (* value is returned in eax *)
                let (sat, rt, tt, lasm, reg) = get_reg_resource dest
                        ltac sat rt tt lasm Eax
                in
                let f_str = src1.tds_name in
                let asm = ("\tcall\t" ^ f_str) in
                let lasm = asm :: lasm in
                (lasm, sat, rt, tt, env)
        | Tevictall -> let (sat, rt, tt, lasm) = evict_all_reg sat rt tt lasm
                in
                (lasm, sat, rt, tt, env)
        | Tespadd -> let imm_str = (string_of_int dest.tds_value) in
                let asm = ("\tadd\tesp,\t" ^ imm_str) in
                let lasm = asm :: lasm in
                (lasm, sat, rt, tt, env)
        | Tevicteax -> let (sat, rt, tt, lasm) =
                        evict_reg Eax sat rt tt lasm in
                (lasm, sat, rt, tt, env)
        | Tevictebx -> let (sat, rt, tt, lasm) =
                        evict_reg Ebx sat rt tt lasm in
                (lasm, sat, rt, tt, env)
        | Tevictecx -> let (sat, rt, tt, lasm) =
                        evict_reg Ecx sat rt tt lasm in
                (lasm, sat, rt, tt, env)
        | Tevictedx -> let (sat, rt, tt, lasm) =
                        evict_reg Edx sat rt tt lasm in
                (lasm, sat, rt, tt, env)
        in
        (* we need to release stack space for tmp variables *)
        let ltac_tl = List.tl ltac in
        let (sat, tt, rt) = release_tmp_stack_space sat tt rt ltac_tl dest in
        let (sat, tt, rt) = release_tmp_stack_space sat tt rt ltac_tl src1 in
        let (sat, tt, rt) = release_tmp_stack_space sat tt rt ltac_tl src2 in
        (lasm, sat, rt, tt, env)



(*
 * generate_function_assembly
 *
 *      Generate assembly for an entire function.
 *
 * Output: (lasm, env)
 *)
let generate_function_assembly func lasm gsat env =
```

```
        (* setup the portions of the environment which are local *)
        let env = {
                age_implabel = env.age_implabel;
                age_soff = -4; (* locals start at ebp-4 *)
                age_doff = env.age_doff;
                age_nlocals = 0 (* havent added locals yet *) } in
        let fname = func.func_symb.symb_name in
        (* first we need to add the local symbols to the sat *)
        let nmsymb = func.func_locals in
        let lsymb = NameMap.fold (fun k v l -> v :: l) nmsymb [] in
        let (sat, env) = add_local_sat_symbols lsymb gsat env in
        (* grap the list of tac ops *)
        let ltac = func.func_ltac in
        (* initialize rt and tt *)
        let rt = {
                rt_eax = "invld", Rs_free;
                rt_ebx = "invld", Rs_free;
                rt_ecx = "invld", Rs_free;
                rt_edx = "invld", Rs_free;
                rt_nextevict = Eax } in
        (* get the amount of space allocated for temporaries *)
        let tmp_num = get_stack_space_for_temps ltac 0 0 in
        let tmp_space = 4 * tmp_num in (* each tmp gets 4 bytes *)
        let rec get_tt_free numtmps nextoffset freelist =
                if numtmps == 0 then List.rev freelist
                else get_tt_free (numtmps - 1) (nextoffset - 4)
                        (nextoffset :: freelist)
        in
        let ttfree = get_tt_free tmp_num 0 [] in
        let local_space = 4 * env.age_nlocals in
        let ttbase = -4 - local_space in
        let tt = {
                tt_base = ttbase;
                tt_free = ttfree } in
        (* add the tmp variables to the SAT *)
        let sat = add_tmp_sat_symbols sat tt ltac in

        (* now we add the function name *)
        let asm = (fname ^ " PROC NEAR") in
        let lasm = asm :: lasm in
        let asm = ("\tpush\tebp") in
        let lasm = asm :: lasm in
        let asm = ("\tmov\tebp,\tesp") in
        let lasm = asm :: lasm in
        let asm = ("\tsub\tesp,\t" ^ (string_of_int (local_space + tmp_space)))
        in
        let lasm = asm :: lasm in
        (* now we initialize the locals *)
        let lasm = generate_local_init_assembly sat lsymb lasm in
        (* generate assembly for the Tac elements within the function *)
        let rec gen_all_asm ltac sat rt tt lasm env =
                if ltac == [] then (lasm, sat, rt, tt, env)
                else
                        begin
                        let (lasm, sat, rt, tt, env) = generate_asm_for_op
                                ltac sat rt tt lasm env in
                        (gen_all_asm (List.tl ltac) sat rt tt lasm env)
                        end
        in
        let (lasm, sat, rt, tt, env) = gen_all_asm ltac sat rt tt lasm env in
        (* we should now be complete with generating assembly for the ops *)
        let asm = ("\tadd\tesp,\t" ^ (string_of_int (local_space + tmp_space)))
        in
        let lasm = asm :: lasm in
        let asm = ("\tpop\tebp") in
```

```
        let lasm = asm :: lasm in
        let asm = ("\tret") in
        let lasm = asm :: lasm in
        (* Final addition - the end of the proc *)
        let asm = (fname ^ " ENDP") in
        let lasm = asm :: lasm in
        (lasm, env)


(*
 * generate_rcb_assembly
 *
 *      Generate assembly for an entire RCB.
 *
 * Output: (lasm, env)
 *)
let generate_rcb_assembly rcb lasm gsat env =
        let rname = rcb.rcb_symb.symb_name in
        (* first we need to add the local symbols to the sat *)
        let nmsymb = rcb.rcb_locals in
        let lsymb = NameMap.fold (fun k v l -> v :: l) nmsymb [] in
        let (sat, env) = add_local_sat_symbols lsymb gsat env in
        (* grap the list of tac ops *)
        let ltac = rcb.rcb_ltac in
        (* initialize rt and tt *)
        let rt = {
                rt_eax = "invld", Rs_free;
                rt_ebx = "invld", Rs_free;
                rt_ecx = "invld", Rs_free;
                rt_edx = "invld", Rs_free;
                rt_nextevict = Eax } in
        (* get the amount of space allocated for temporaries *)
        let tmpmap = IntMap.empty in
        let tmp_num = get_stack_space_for_temps ltac 0 0 in
        let tmp_space = 4 * tmp_num in (* each tmp gets 4 bytes *)
        let rec get_tt_free numtmps nextoffset freelist =
                if numtmps == 0 then List.rev freelist
                else get_tt_free (numtmps - 1) (nextoffset - 4)
                        (nextoffset :: freelist)
        in
        let ttfree = get_tt_free tmp_num 0 [] in
        let local_space = 4 * env.age_nlocals in
        let ttbase = -4 - local_space in
        let tt = {
                tt_base = ttbase;
                tt_free = ttfree } in
        (* add the tmp variables to the SAT *)
        let sat = add_tmp_sat_symbols sat tt ltac in
        (* now we add the RCB name *)
        let asm = (rname ^ " PROC NEAR") in
        let lasm = asm :: lasm in
        let asm = ("\tpush\tebp") in
        let lasm = asm :: lasm in
        let asm = ("\tmov\tebp,\tesp") in
        let lasm = asm :: lasm in
        let asm = ("\tsub\tesp,\t" ^ (string_of_int (local_space + tmp_space)))
        in
        let lasm = asm :: lasm in
        (* now we initialize the locals *)
        let lasm = generate_local_init_assembly sat lsymb lasm in
        (* generate assembly for the Tac elements within the RCB *)
        let rec gen_all_asm ltac sat rt tt lasm env =
                if ltac == [] then (lasm, sat, rt, tt, env)
                else
                        begin
```

```
                        let (lasm, sat, rt, tt, env) = generate_asm_for_op
                                ltac sat rt tt lasm env in
                        (gen_all_asm (List.tl ltac) sat rt tt lasm env)
                        end
        in
        let (lasm, sat, rt, tt, env) = gen_all_asm ltac sat rt tt lasm env in
        (* we should now be complete with generating assembly for the ops *)
        let asm = ("\tadd\tesp,\t" ^ (string_of_int (local_space + tmp_space)))
        in
        let lasm = asm :: lasm in
        let asm = ("\tpop\tebp") in
        let lasm = asm :: lasm in
        let asm = ("\tret") in
        let lasm = asm :: lasm in
        (* Final addition - the end of the proc *)
        let asm = (rname ^ " ENDP") in
        let lasm = asm :: lasm in
        (lasm, env)


(*
 * generate_assembly_program
 *
 *      Generate assembly for an entire program.
 *
 * Output:  (lasm)
 *)
let generate_assembly_program prog =
       (* first lets add the structure surrounding the assembly program *)
       (*
        * Intended assembly:
        *
        * ; Begin assembly
        * .386
        *
        * DSEG  SEGMENT PUBLIC
        *       BYTE    3FF0h   dup     (0)
        *       TABLE   DB      '0123456789ABCDEF'
        * DSEG  ENDS
        *
        * SSEG  SEGMENT STACK
        *       BYTE    1000h   dup     (0)
        * SSEG  ENDS
        *
        * CSEG  SEGMENT PUBLIC  USE16
        *       ASSUME  CS:CSEG, DS:DSEG, SS:SSEG
        *
        * START:
        *       mov     ax,     DSEG
        *       mov     ds,     ax
        *
        *       call    main    ; main must be defined
        *
        * EXIT: mov     ah,     4Ch
        *       int     21h
        *
        * < assembly from Functions and RCBs >
        *
        * < assembly for printing numbers >
        *
        * CSEG  ENDS
        *
        * END   START
        * ; End assembly
        *)
```

```
        let lasm = [] in
        let asm = ("; Begin Assembly") in
        let lasm = asm :: lasm in
        let asm = (".386") in
        let lasm = asm :: lasm in
        let asm = ("\n") in
        let lasm = asm :: lasm in
        let asm = ("DSEG\tSEGMENT\tPUBLIC") in
        let lasm = asm :: lasm in
        let asm = ("\tBYTE\t3FF0h\tdup\t(0)") in
        let lasm = asm :: lasm in
        let asm = ("\tTABLE\tDB\t'0123456789ABCDEF'") in
        let lasm = asm :: lasm in
        let asm = ("DSEG\tENDS") in
        let lasm = asm :: lasm in
        let asm = ("\n") in
        let lasm = asm :: lasm in
        let asm = ("SSEG\tSEGMENT\tSTACK") in
        let lasm = asm :: lasm in
        let asm = ("\tBYTE\t1000h\tdup\t(0)") in
        let lasm = asm :: lasm in
        let asm = ("SSEG\tENDS") in
        let lasm = asm :: lasm in
        let asm = ("\n") in
        let lasm = asm :: lasm in
        let asm = ("CSEG\tSEGMENT\tPUBLIC\tUSE16") in
        let lasm = asm :: lasm in
        let asm = ("\tASSUME\tCS:CSEG, DS:DSEG, SS:SSEG") in
        let lasm = asm :: lasm in
        let asm = ("\n") in
        let lasm = asm :: lasm in

        (* ENTRY POINT *)
        let asm = "START:" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tax,\tDSEG" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tds,\tax" in
        let lasm = asm :: lasm in
        let asm = "\tcall\tmain" in
        let lasm = asm :: lasm in
        let asm = "EXIT:\tmov\tah,\t4Ch" in
        let lasm = asm :: lasm in
        let asm = "\tint\t21h" in
        let lasm = asm :: lasm in
        let asm = "\n" in
        let lasm = asm :: lasm in


        (* setup the initial environment *)
        let env = {
                age_implabel = 0;
                age_soff = 0;
                age_doff = 0;
                age_nlocals = 0 } in

        (* create the global sat *)
        let gsat = Tac_types.NameMap.empty in
        let lsymb = NameMap.fold (fun k v l -> v :: l) prog.prog_globals [] in
        let (gsat, env) = add_global_sat_symbols lsymb gsat env in

        (* generate the assembly for functions *)
        (* Output: (lasm, env) *)
        let rec gen_all_funcs lfunc lasm gsat env =
                if lfunc == [] then (lasm, env)
```

```
                else
                        begin
                        let func = List.hd lfunc in
                        let lfunc = List.tl lfunc in
                        let (lasm, env) = generate_function_assembly
                                func lasm gsat env in
                        let asm = ("\n") in
                        let lasm = asm :: lasm in
                        (gen_all_funcs lfunc lasm gsat env)
                        end
        in
        let (lasm, env) = gen_all_funcs prog.prog_lfunc lasm gsat env in
        (* generate the assembly for RCBs *)
        let rec gen_all_rcbs lrcb lasm gsat env =
                if lrcb == [] then (lasm, env)
                else
                        begin
                        let rcb = List.hd lrcb in
                        let lrcb = List.tl lrcb in
                        let (lasm, env) = generate_rcb_assembly
                                rcb lasm gsat env in
                        let asm = ("\n") in
                        let lasm = asm :: lasm in
                        (gen_all_rcbs lrcb lasm gsat env)
                        end
        in
        let (lasm, env) = gen_all_rcbs prog.prog_lrcb lasm gsat env in

        (* add the printing functionality *)
        let lasm = hex_print lasm in

        let asm = ("CSEG\tENDS") in
        let lasm = asm :: lasm in
        let asm = ("\n") in
        let lasm = asm :: lasm in
        let asm = ("END\tSTART") in
        let lasm = asm :: lasm in
        let asm = ("; End Assembly") in
        let lasm = asm :: lasm in
        (lasm)
```

```
(* Output:  (lasm) *)
let hex_print lasm =
        (* hex_to_ascii *)
        let asm = "hex_to_ascii\tPROC" in
        let lasm = asm :: lasm in
        let asm = "\tpush\tdi" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tah,\t0" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tdi,\tax" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tal,\tTABLE[di]" in
        let lasm = asm :: lasm in
        let asm = "\tpop\tdi" in
        let lasm = asm :: lasm in
        let asm = "\tret" in
        let lasm = asm :: lasm in
        let asm = "hex_to_ascii\tendp" in
        let lasm = asm :: lasm in
        let asm = "\n" in
        let lasm = asm :: lasm in

        (* print_hex_al *)
        let asm = "print_hex_al\tproc" in
        let lasm = asm :: lasm in
        let asm = "\tpush\tcx" in
        let lasm = asm :: lasm in
        let asm = "\tpush\tdx" in
        let lasm = asm :: lasm in
        let asm = "\tpush\tsi" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tsi,\tax" in
        let lasm = asm :: lasm in
        let asm = "\tand\tal,\t0fh" in
        let lasm = asm :: lasm in
        let asm = "\tcall\thex_to_ascii" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tcx,\tax" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tax,\tsi" in
        let lasm = asm :: lasm in
        let asm = "\tshr\tal,\t4" in
        let lasm = asm :: lasm in
        let asm = "\tcall\thex_to_ascii" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tdx,\tax" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tah,\t2" in
        let lasm = asm :: lasm in
        let asm = "\tint\t21h" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tdx,\tcx" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tah,\t2" in
        let lasm = asm :: lasm in
        let asm = "\tint\t21h" in
        let lasm = asm :: lasm in
        let asm = "\tpop\tsi" in
        let lasm = asm :: lasm in
        let asm = "\tpop\tdx" in
        let lasm = asm :: lasm in
        let asm = "\tpop\tcx" in
        let lasm = asm :: lasm in
```

```
        let asm = "\tret" in
        let lasm = asm :: lasm in
        let asm = "print_hex_al\tendp" in
        let lasm = asm :: lasm in
        let asm = "\n" in
        let lasm = asm :: lasm in

        (* print_hex_ax *)
        let asm = "print_hex_ax\tproc" in
        let lasm = asm :: lasm in
        let asm = "\tpush\tax" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tal,\tah" in
        let lasm = asm :: lasm in
        let asm = "\tcall\tprint_hex_al" in
        let lasm = asm :: lasm in
        let asm = "\tpop\tax" in
        let lasm = asm :: lasm in
        let asm = "\tcall\tprint_hex_al" in
        let lasm = asm :: lasm in
        let asm = "\tret" in
        let lasm = asm :: lasm in
        let asm = "print_hex_ax\tendp" in
        let lasm = asm :: lasm in
        let asm = "\n" in
        let lasm = asm :: lasm in

        (* print_hex *)
        let asm = "print_hex\tproc" in
        let lasm = asm :: lasm in
        let asm = "\tpush\tebp" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tebp,\tesp" in
        let lasm = asm :: lasm in
        let asm = "\tmov\teax,\tss:[ebp+6]" in
        let lasm = asm :: lasm in
        let asm = "\tpush\tax" in
        let lasm = asm :: lasm in
        let asm = "\tshr\teax,\t16" in
        let lasm = asm :: lasm in
        let asm = "\tcall\tprint_hex_ax" in
        let lasm = asm :: lasm in
        let asm = "\tpop\tax" in
        let lasm = asm :: lasm in
        let asm = "\tcall\tprint_hex_ax" in
        let lasm = asm :: lasm in
        let asm = "\tpop\tebp" in
        let lasm = asm :: lasm in
        let asm = "\tret" in
        let lasm = asm :: lasm in
        let asm = "print_hex\tendp" in
        let lasm = asm :: lasm in
        let asm = "\n" in
        let lasm = asm :: lasm in

        (* print_crlf *)
        let asm = "print_crlf\tproc" in
        let lasm = asm :: lasm in
        let asm = "\tpush\tax" in
        let lasm = asm :: lasm in
        let asm = "\tpush\tdx" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tah,\t2" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tdl,\t13" in
```

```
        let lasm = asm :: lasm in
        let asm = "\tint\t21h" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tah,\t2" in
        let lasm = asm :: lasm in
        let asm = "\tmov\tdl,\t10" in
        let lasm = asm :: lasm in
        let asm = "\tint\t21h" in
        let lasm = asm :: lasm in
        let asm = "\tpop\tdx" in
        let lasm = asm :: lasm in
        let asm = "\tpop\tax" in
        let lasm = asm :: lasm in
        let asm = "\tret" in
        let lasm = asm :: lasm in
        let asm = "print_crlf\tendp" in
        let lasm = asm :: lasm in
        let asm = "\n" in
        let lasm = asm :: lasm in

        (lasm)
```

```
open Ast
open Tacgen
open Tac_display
open Asmgen
open Printf

let print = false

let _ =
  (* Make sure correct number of arguments is given *)
  if (Array.length Sys.argv) != 3 then
          raise (Failure("Num args: " ^ (string_of_int (Array.length Sys.argv)) ^
                " Expected 3."))
  else
  let filename = Sys.argv.(1) in
  let rcbseed = Sys.argv.(2) in
  let _ = print_endline ("Compiling File: " ^ filename) in
  let _ = print_endline ("RCB Seed Value: " ^ rcbseed) in

  (* Get the seed value and input channel *)
  let rcbseed_int = int_of_string rcbseed in
  let ic = open_in filename in

  (* Scan the file *)
  print_endline "Starting parsing...";
  let lexbuf = Lexing.from_channel ic in
  (* Parse the scanned tokens *)
  print_endline "Finished lexbuf definition...";
  let program = Parser.program Scanner.token lexbuf in
  print_endline "Finished parser program...";
  (* Create intermediate three-address code *)
  let tac_prog =
          Tacgen.tac_create_prog program rcbseed_int in
  let _ = Tac_display.print_tac_program tac_prog in
  (* Create the assembly listing *)
  let lasm = Asmgen.generate_assembly_program tac_prog in
  let lasm = List.rev lasm in
  let _ = List.map (fun x -> (print_endline x)) lasm in

  (* Write the results to an output assembly file *)
  let outfile = String.sub filename 0 ((String.length filename)-4) in
  let outfile = (outfile ^ ".asm") in
  let _ = print_endline ("Writing Assembly File: " ^ outfile) in

  let oc = open_out outfile in
  let _ = List.map (fun x -> (output_string oc x; output_char oc '\n')) lasm in
  let _ = close_out oc in
  1
```

```
/*
 * bsort.moc
 *
 *      This is a simple bubble sort algorithm.  The input
 *      list consists of an array of integers from 1 to
 *      20, and the algorithm arranges them sequentially
 *      from smallest to largest.
 */

void main();
ord4 bsort();
void print_the_array();
void print_hex(ord4 x);
void print_crlf();

/* 80 byte array - 20 digits */
array arr[80];

void main()
{
        ord4 tmp;

        /* initialize the array */
        arr[0]ord4 = 17;
        arr[4]ord4 = 8;
        arr[8]ord4 = 19;
        arr[12]ord4 = 20;
        arr[16]ord4 = 3;
        arr[20]ord4 = 9;
        arr[24]ord4 = 1;
        arr[28]ord4 = 18;
        arr[32]ord4 = 11;
        arr[36]ord4 = 7;
        arr[40]ord4 = 4;
        arr[44]ord4 = 13;
        arr[48]ord4 = 2;
        arr[52]ord4 = 12;
        arr[56]ord4 = 16;
        arr[60]ord4 = 6;
        arr[64]ord4 = 10;
        arr[68]ord4 = 14;
        arr[72]ord4 = 5;
        arr[76]ord4 = 15;

        /* print the starting state */
        print_the_array();
        print_crlf();
        print_crlf();

        /* sort the array */
        tmp = 1;
        while(tmp != 0)
                tmp = bsort();

        /* print the final state */
        print_the_array();
        print_crlf();
}

ord4 bsort()
{
        ord4 fst;
        ord4 snd;
        ord4 i;
        ord4 swapped;
```

```
        swapped = 0;
        for(i = 0; i < 19; i = i + 1)
        {
                fst = arr[4*i]ord4;
                snd = arr[4*(i+1)]ord4;
                if(fst > snd)
                {
                        arr[4*i]ord4 = snd;
                        arr[4*(i+1)]ord4 = fst;
                        swapped = swapped + 1;
                }
        }
        return swapped;
}

void print_the_array()
{
        ord4 i;
        ord4 x;
        for(i = 0; i < 20; i = i + 1)
        {
                x = arr[4*i]ord4;
                print_hex(x);
                print_crlf();
        }
}
```

```
/*
 * fact.moc
 *
 *      This source calculates the factorial of 8 (8!) and
 *      displays it to the screen.  This demonstrates
 *      recursion.
 */

void main();
ord4 fact(ord4 x);
void print_hex(ord4 x);
void print_crlf();


void main()
{
        ord4 x;

        x = fact(8);

        print_hex(x);
        print_crlf();
}


ord4 fact(ord4 x)
{
        ord4 tmp;
        if(x == 0)
        {
                return 0;
        }
        else if(x == 1)
        {
                return 1;
        }
        else
        {
                /*
                 * One restriction of the language: Parameters must
                 * be either immediates or variables, and may not be
                 * complex expressions
                 */
                tmp = x - 1;
                tmp = x * fact(tmp);
                return tmp;
        }
}
```

```
/*
 * spath.moc
 *
 *      This source file contains an algorithm to find
 *      the shortest path between two points.  This is done
 *      using three arrays: source, destination, and distance.
 *      Using these three arrays, the algorithm will find the
 *      shortest distance between two nodes.  The edges will be
 *      set up as follows.
 *
 *              Source  Dest    Distance
 *              1       2       3
 *              1       5       1
 *              1       7       5
 *              2       3       4
 *              3       6       2
 *              3       4       7
 *              5       4       23
 *
 *      There are two paths provided by these edges, but the
 *      shortest path has distance 14, so that is what we expect
 *      to be displayed.
 */

void main();
void print_hex(ord4 x);
void print_crlf();
ord4 calculate_distance(ord4 from, ord4 me, ord4 to, ord4 distance);

array source[28];
array dest[28];
array dist[28];

void main()
{
        ord4 shortest;

        /* initialize the arrays */
        source[0]ord4 = 1;
        dest[0]ord4 = 2;
        dist[0]ord4 = 3;

        source[4]ord4 = 1;
        dest[4]ord4 = 5;
        dist[4]ord4 = 1;

        source[8]ord4 = 1;
        dest[8]ord4 = 7;
        dist[8]ord4 = 5;

        source[12]ord4 = 2;
        dest[12]ord4 = 3;
        dist[12]ord4 = 4;

        source[16]ord4 = 3;
        dest[16]ord4 = 6;
        dist[16]ord4 = 2;

        source[20]ord4 = 3;
        dest[20]ord4 = 4;
        dist[20]ord4 = 7;

        source[24]ord4 = 5;
        dest[24]ord4 = 4;
        dist[24]ord4 = 23;
```

```
        /* calculate the shortest path */
        shortest = calculate_distance(0, 1, 4, 0);

        /* display the shortest path found */
        print_hex(shortest);
        print_crlf();

}


ord4 calculate_distance(ord4 from, ord4 me, ord4 to, ord4 distance)
{
        ord4 shortest = 1000;
        ord4 tmp;
        ord4 csrc;
        ord4 cdest;
        ord4 cdist;
        ord4 i;

        for(i = 0; i < 7; i = i + 1)
        {
                csrc = source[4*i]ord4;
                cdest = dest[4*i]ord4;
                cdist = dist[4*i]ord4;

                if(csrc == me)
                {
                        /* this is me, see if there are any paths */
                        if(cdest == to)
                        {
                                if((distance+cdist) < shortest)
                                {
                                        shortest = distance + cdist;
                                }
                        }
                        else if(cdest != from)
                        {
                                tmp = distance + cdist;
                                tmp = calculate_distance(me, cdest, to, tmp);
                                if(tmp < shortest)
                                {
                                        shortest = tmp;
                                }
                        }
                        else
                        {
                                /* do nothing */
                        }
                }
        }
        return shortest;
}
```

```
/*
 * rcb.moc
 *
 *      This source file is intended to test the functionality of
 *      Random Code Blocks within MOC-V.  This is intended to be
 *      done simply by diffing the results of compilations performed
 *      with various seed values.
 *
 *      Each RCB herein also modifies a global array, which will allow
 *      a user to run various compilations of this program and see the
 *      different outcomes.
 */

void main();
rcode rcb1;
rcode rcb2;
rcode rcb3;
rcode rcb4;
void print_hex(ord4 x);
void print_crlf();

array arr[4];

void main()
{
        ord4 x;

        /* initialize array to zero */
        arr[0]ord4 = 0;

        /* run a sequence of RCBs */
        [(25)rcb1, (25)rcb2, (25)rcb3, (25)rcb4];
        [(25)rcb1, (25)rcb2, (25)rcb3, (25)rcb4];
        [(25)rcb1, (25)rcb2, (25)rcb3, (25)rcb4];
        [(25)rcb1, (25)rcb2, (25)rcb3, (25)rcb4];


        x = arr[0]ord4;
        print_hex(x);
        print_crlf();
}

rcode rcb1
{
        arr[0]ord4 = arr[0]ord4 + 0x1;
}

rcode rcb2
{
        arr[0]ord4 = arr[0]ord4 + 0x10;
}

rcode rcb3
{
        arr[0]ord4 = arr[0]ord4 + 0x100;
}

rcode rcb4
{
        arr[0]ord4 = arr[0]ord4 + 0x1000;
}
```