Bachelor Project in Compiler Construction

Re-exam

August 23, 2019

Report from group GROUPNUMBER: 7

Sebastian Eklund Larsen, selar16
Sigurdur Smári Gunnarsson Waage, sigun16

Contents

1 Introduction

This project describes the process of writing a rudimentary compiler. For learning purposes it is aimed at a minimal programming language ”Kitty”. The compiler is written in C and compiles to assembly.

1.1 Extensions

There have been added increment and decrement statements as well for loops to the grammar.

1.2 Implementation Status

The compiler is able to generate correct code for most of the test files we have made. The known exceptions are files where functions at different scopes have the same name.Other than that programs fail if they allocate too much memory or need runtime checks.

2 Parsing and Abstract Syntax Trees

The objective, at this point in the project, is to generalize the input (kitty-code) into a tokenized tree structure (Abstract syntax tree). The nodes in the tree are designed to store and retain the features of the input program which are relevant. The relevant features of the code, in regards to compiling, is at this point restricted to the following parts: expressions, terms, variables, head, functions, lists of expressions, and lists of statements. The tokenization is achieved by passing the input through the text analyzing tool `flex`. And subsequently using `bison` to structure the tokens into an the abstract syntax tree according to the grammar described below.

2.1 The Grammar

The grammar can be seen in figure 1 and is continued in figure 2.
The words on the left are implemented as node structures, with a reference to the values represented by one of the following lines on the right side. The words on the right that are surrounded by angle brackets are the other grammar variables the left side can expand to. Everything written in bold is text, as it appears in the file.

2.2 Use of the `flex` Tool

Flex is used to scan text, and produces tokens. This is done by defining a language through regular expressions, and making corresponding tokens to match the expres-

⟨function⟩ : ⟨head⟩ ⟨body⟩ ⟨tail⟩
⟨head⟩ : **func id** (⟨par_decl_list⟩) **:** ⟨type⟩
⟨tail⟩ : **end id**
⟨type⟩ : **id**
| **int**
| **bool**
| **array of** ⟨type⟩
| **record of** { ⟨var_decl_list⟩ }
⟨par_decl_list⟩ : ⟨var_decl_list⟩
| ε
⟨var_decl_list⟩ : ⟨var_type⟩ **,** ⟨var_decl_list⟩
| ⟨var_type⟩
⟨var_type⟩ : **id :** ⟨type⟩
⟨body⟩ : ⟨decl_list⟩ ⟨statement_list⟩
⟨decl_list⟩ : ⟨declaration⟩ ⟨decl_list⟩
| ε
⟨declaration⟩ : **type id =** ⟨type⟩ **;**
| ⟨function⟩
| **var** ⟨var_decl_list⟩ **;**
⟨statement_list⟩ : ⟨statement⟩
| ⟨statement⟩ ⟨statement_list⟩
⟨statement⟩ : **return** ⟨expression⟩ **;**
| **write** ⟨expression⟩ **;**
| **allocate** ⟨variable⟩ **;**
| **allocate** ⟨variable⟩ **of length** ⟨expression⟩ **;**
| ⟨variable⟩ **=** ⟨expression⟩ **;**
| **if** ⟨expression⟩ **then** ⟨statement⟩
| **if** ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩
| **while** ⟨expression⟩ **do** ⟨statement⟩
| { ⟨statement_list⟩ }
⟨variable⟩ : **id**
| ⟨variable⟩ [⟨expression⟩]
| ⟨variable⟩ **. id**

Figure 1: grammar part 1 of 2

⟨expression⟩ : ⟨expression⟩ **op** ⟨expression⟩
| ⟨term⟩
⟨term⟩ : ⟨variable⟩
| **id (** ⟨act_list⟩ **)**
| **(** ⟨expression⟩ **)**
| **!** ⟨term⟩
| **|** ⟨expression⟩ **|**
| **num**
| **true**
| **false**
| **null**
⟨act_list⟩ : ⟨exp_list⟩
| ε
⟨exp_list⟩ : ⟨expression⟩
| ⟨expression⟩ **,** ⟨exp_list⟩

Figure 2: grammar part 2 of 2

sion with. When flex gets a text input it splits it up into tokens, where it prioritizes longest match, so if we have regular expressions `[a-zA-Z_][a-zA-Z0-9_]*` and `[while]`, and encounter "whiley", then it would match to the first expression because it has a longer match. When flex is done it has generated a sequence of tokens, which is used by bison for parsing. The tokens which flex can produce are the tokens representing the bold parts of the grammar, and will be the terminal tokens of our language.

We have a token for each binary operator, and we save the operator inside the token to make it easier to print later. Tokens which are equivalent to keywords like "while" and "type" simply return a token representing that keyword. Booleans are represented as 1's and 0's, and as such true and false return a boolean token with 1 or 0 saved in it respectably. null returns a NULL token. Numbers return an int token with the specified value saved in it. Words which start with an alphabet character followed by a combination of alphabet characters and numbers get matched to ids, and return id tokens. This tokens have a lower priority than the keywords, so if a keyword like "while" appears it will be matched to the while token. Comments can start with a `"#"` symbol, which makes the flex ignore the rest of the line. They can also appear starting with `"(*"` and ending with `"*)"`, when it increments a counter for each start comment it sees, and enters a new mode which only matches start and end comments. When it encounters an end comment it decrements the counter, and once the counter reaches 0 it goes back to its initial mode. If flex is unable to match symbols to a token it returns a token not in the grammar, indicating that the input file is not in our language. If the file ends in the comment mode it also returns an error token.

```
1       %nonassoc OR EQ LEQ GEQ NEQ
2       %right '<' '>'
3       %left '+' '-'
4       %left '*' '/' AND
5       %right '[' '{' '('
6       %left ']' '}' ')'
7       %nonassoc THEN
8       %nonassoc ELSE
```

Figure 3: Bison associations

2.3 Use of the `bison` Tool

In bison the grammar is implemented. The possible expansions for each left hand side are defined, where the expansions consist of a series of tokens. Bison constructs a transition table from this grammar, which defines state transitions from states to other states based on the tokens read. The tokens consist of the tokens produced by flex, and tokens for each left hand side of the grammar. The tokens from flex are terminal because they do not have an expansion. Tokens which do have an expansion must themselves, be able to be expanded into resulting in all terminals. Bison finishes parsing once it has read all the inputted tokens into the grammar, and it accepts if the finishing symbol results in a accept action from the transition table. While parsing the program, bison will save the tokens in a tree structure. Each node will have references to the children used in its production, which creates an abstract syntax tree with the start symbol as root. In order to make bison properly parse the program, we need to define the precedence and association of certain tokens, which can be seen in 3.

2.4 Grammar Extensions

These tokens were added to the compiler "++" returning INC, "–" returning DEC and "for" returning FOR.

Increment and decrement were added to the grammar of the compiler. They are represented by the same syntax they often use.
`i++;`
`i--;`
They are state as they resemble functionality of the state assign. Their grammar looks as follows:

(variable) INC ';'
(variable) DEC ';'

For loops were also added to the grammar in two types, incremental and decremental, they are of a bit less traditional style.

```
for (var i : int;) ++ [0,10]
for (var i : int;) -- [10,0]
```

With the grammar:
FOR '(' decllist ')' INC '[' term ',' term ']' statement
FOR '(' decllist ')' DEC '[' term ',' term ']' statement

The integer on the left inside the square brackets is then assigned to the variable created. Then it increments/decrements towards the integer on the right in the brackets.
While the incremental/decremental part works well, the whole for loop does not work completely as intended. The created variable can not be called in the statement that follows the for loop condidtions.
This could potentially be solved by making a new scope for the loop.

2.5 Abstract Syntax Trees

The abstract syntax tree allows for representing the program syntax. Each left hand side of the grammar has its own structure, with a kind enumerator to determine which expansion is used, and references to any of its expansions. When bison finds that the tokens on the stack correspond to some left hand side(implied by current state in transition table), it reduces by the rule of that left hand side, and calls the corresponding constructor for that rule with left hand sides tokens. This results in a structure with references to the important token values. We do this for every reduction, and eventually we have a tree which can be traversed to derive the original input.

We have added the ability to store type information in EXP nodes, so that we can type check its arguments and result in the type checking phase.
The BODY node gets a reference to a collection structure, which holds information on the current scope in the program, which we will go more in depth with in the type checking section. Collection is defined in the same file as the tree in order to avoid circular dependence, caused by BODY needing access to the COLLECTION structure, which needs access to NestScopeList, which needs Access to BODY. NestScopeList is used by COLLECTION to store a list of all the function nodes nested immediately in that scope, this is used to, postpone the type checking of functions, until we are done with the current scope..

2.6 Test

We test that association is implemented correctly with the tests `LargeExpTreeA`, `LargeExpTreeB`, `LargeExpTreeC` and `O_Assoc`. Since all of these files result in generated code, which writes the correct result out, we can assume that association is properly implemented.

3 The Symbol Table

Symbol tables are used to keep track of identifiers and their corresponding value, according to the scope at which they appear.

3.1 Scope Rules

Variables defined in a certain scope is inserted into the symbol table of that scope. When we utilize a variable of some name we will be using the variable of that name, which occurs earliest in the path of symbol tables, from the current to the outermost. When more than one variable of a certain name gets defined in the same scope, we keep the first one and ignore any subsequent.

3.2 Symbol Data

The symbol table consist of the three structures, Symbol, Symbol table and SymbolList.

3.2.1 Symbol

A symbol is a (String, Value) pair which is used to store values paired with their identifiers, for example variables, functions and objects. These symbols are then put in symbol tables. The symbols can be looked up in these tables for reasons such as for verifying if a variable has been declared. Symbol can also reference another symbol so that they can be used as a linked list.

3.2.2 Symbol table structure

The structure of the symbol table is that of a tree, each node contains a scope of symbols stored in a hash table. The program contains functions for getting and putting symbols into the table. A symbol table has a reference to its parents, which is used when a symbol which does not exist in the current symbol table is requested, as it will try to find it in the parent table instead. This is used to return the earliest matching symbol on the path from the current node to the root node.

3.2.3 SymbolList

SymbolList is used in the function `AllSymbolsInScope()` to return a list of all the symbols in the current scope.

3.3 The Algorithm

putSymbol(SymbolTable *t, char *name, void *value):
calculates hash on name and if the hashed value is an empty slot in the table, we create a new symbol at that slot with name and value equal to the arguments, and return the symbol. If the slot was not empty we look through the list of symbols at that slot, and if a symbol with the same name exists we return that, otherwise we create a new symbol at the end of the list, and return that symbol.

getSymbol(SymbolTable *t, char *name):
Calculates hash value from the name argument, and looks through that index of the given symbol table returning the symbol with the same name. If it does not find a match, it returns the result of getSymbol given its parent table, if there is no parent table returns null.

3.4 Test

The symbol table successfully passes the test in `O_FuncRedefinedType`, where the function `d()` correctly uses the type b defined inside its scope, instead of the one in the outer scope. It also properly inserts the variables `v1` and `v3` into the symbol table, and gets the correct ones out later. In the tests `C_ErrFuncTooMany` and `C_ErrFuncTooFew`, it is used to store information for functions, and is able successfully return the values afterwards. Allowing for the type checker to type check correctly.

4 Weeding

The weeder traverses the structure from the Abstract Syntax Tree section. The weeder checks if function id and end id matches for every function in the code. It also uses boolean functions to check if the functions are guaranteed to result in a return statement. If the id mismatches, the code is not fit for running, and the user gets a colorized message printed in the terminal and the program exits. The message includes the name of the function and the line number of the function start.
In the case of the return statements, the user gets a warning message but keeps running.

The id check compares the id strings from the head and tail in the abstract syntax tree. The return check traverses through the functions looking for return- and if/else statements. The same return check is performed for each nested if/else statement. In order for the return check to be successful it needs to satisfy either of these two conditions:
-If there is found an if/else statement within the functions statement list. Both ”then” and ”else” statements must result in a return statement.
-There is a return statement in the body statement list of the function itself.

If a function is guaranteed to reach a return state but still has code left in the function. A Note with the function line number is printed, it says that the function will never reach the end of the function code.

4.1 Test

Two test files were made for the weeder. The first test file has a function with an end ID that does not match. The testing should print an error and exit the program.
The latter test has two functions, one that is not guaranteed to end in return state and another that should. The second function has an if/else statement and more code that will not be reached. In the then statement is another if/else statement which should return true. The else statement returns true also. This way both if/else- and nested if/else statements can be tested. Running this should print a warning message for the first function and a note for the second function.

5 Type Checking

In the typechecking phase we inspect the program to see if the types in the program are used correctly. To do this we must define a representation for types in our compiler, decide in what context the types can be used, and check if the program fits those specifications.

5.1 Types

We have defined our type structure `Ty_ty` in the file types.h in figure 4, of the types defined in that structure only types of the kinds record, integer, array, name, nil and boolean are implemented in the compiler.

`Ty_int` is used to represent numbers, `Ty_bool` represents booleans and `Ty_nil` is the representation for null.

`Ty_Name` is used to reference user defined types through their name. This type has a reference to its name and the symbol table it is defined in. this allows us to get the actual type, by retrieving the symbol from the symbol table. The reason for saving a reference to name and table, instead of simply referencing the actual type, is to allow for forward declaration and recursive types, as they would not be able to get a reference until the type has been created. Once the type has been created one would be able to retrieve it with its name and symbol table.

We use `Ty_array` to represent arrays. Arrays are meant to be a list of a single type, so we save a reference to the type declared in the declaration of the array.

```
1  typedef struct Ty_ty_ {
2          enum {Ty_record, Ty_nil, Ty_int, Ty_string,
               Ty_array, Ty_name, Ty_void, Ty_bool} kind;
3          union {Ty_fieldList *record;
4                 Ty_ty *array;
5                 struct {char* name; SymbolTable *table;}
                      name;
6          } u;
7          Ty_tyList *equal;
8  } Ty_ty;
```

Figure 4: Type structure

```
1  typedef struct Ty_ty_func_{
2          Ty_tyList *formals;
3          Ty_ty *type;
4  } Ty_ty_func;
```

Figure 5: function

`Ty_record` represents records, which contains a set of identifiers, each belonging to some type. For each identifier the record contains a value, and by accessing the record with a specific identifier one can access that identifiers value. Record types have a reference to a list of (name, type) pairs, which will be used to typecheck the use of identifiers.

In order to typecheck functions we have introduced the `Ty_ty_func` which can be seen in figure 5. In this structure we have a list of types, representing the order of types appearing as arguments for the function. The structure also has a reference to the functions return type.

We have introduced the structure `COLLECTION` from tree.h (seen in figure 6), in order to save various information about the current scope. This structure has three different symbol tables, one for types, one for variables and one for functions, allowing there to be a function, variable and type with the same name. When we enter a new scope we create a new collection with a reference to the collection of the outer scope. For each symbol table in the outer collection we run scopeSymbolTable(), and save the result in the new collection. nestedlist is a pointer to the list of function nodes nested in the current scope. We build this list when going through declarations of the current scope, so that we can typecheck inside the functions afterwards. The reason we need to wait to enter these nodes until afterwards, is because the function might use variables or functions which are declared after the function, so we need to create any object which

```
1  typedef struct COLLECTION_{
2      SymbolTable *var;
3      SymbolTable *type;
4      SymbolTable *function;
5      NestScopeList *nestedlist;
6      Ty_ty *returns;
7      COLLECTION *next;
8  } COLLECTION;
```

Figure 6: Collection

could be reached by the function first. Collection also has a reference to its return type, to typecheck any return statements in this scope. The current collection is also saved in every BODY node so that the later phases can use the information from each scope.

5.2 Type Rules

5.2.1 Type Equivalence

Type equivalence defines which types are compatible with each other. Two of the same type are implicitly equivalent, but two different types could also be equivalent. our equivalence rules are as follows.
A `Ty_name` is equivalent to the first non `Ty_name` received by repeatedly getting the names actual type.
Two arrays are equivalent if the contained type is equivalent.
Records are equivalent if the order the identifiers were declared results in two list were each index's type is equivalent to the type at the same index in the other list.
`Ty_nil` is equivalent to both `Ty_array` and `Ty_record`.

5.2.2 Expressions

If an expression has a single term node as child it inherits the type of that term. If an expression has a binary operator one needs to check that the arguments are of the expected type according to the operator. The type of these expressions is the type the operand results in. The table for expected types and result type per operator can be seen in 7. As seen in the table, an expression can expect its arguments to be intergers, booleans or equivalent. The result can either be a boolean or integer.

5.2.3 Terms

Terms have a type depending on its kind, which was determined by the production chosen in the parsing phase. An overview of the types and kind of term can be seen in

Expression	Result	Argument
+	Int	Int
-	Int	Int
*	Int	Int
/	Int	Int
==	Bool	equiv
!=	Bool	equiv
>	Bool	Int
<	Bool	Int
>=	Bool	Int
<=	Bool	Int
&&	Bool	Bool
||	Bool	Bool

Figure 7: Expression types

the table in figure 8.

5.2.4 Assignment

The type of the variable on the left hand side must be equivalent to the expression on the right hand side.

5.2.5 Functions

The arguments used for calling a function must be equivalent to the types in the declaration. Inside the function we also need to check that the correct type is returned.

5.2.6 Circular name types

We do not allow for circular definitions of `Ty_name`. We enforce this rule because a circular name cannot be used, as it cannot hold any value.

5.2.7 Checks

We need to ensure that booleans are used for the checks, which happen in statements such as while, if and ifelse.

5.2.8 Allocate of Length

The requested length needs to be an integer.

Term	Term type
Variable	Variables type
Function()	Functions type
(Expression)	Expressions type
!Term	Boolean
|Expression|	Integer
Number	Integer
Boolean	Boolean
Null	Nil

Figure 8: Term types

5.3 The Algorithm

We do typechecking by traversing the abstract syntax tree. Whenever we enter a BODY node, we call scopeCollection on the current collection, because each BODY node is a scope. We then collect the variables, types and functions, which were defined in that scope. For function declarations we initially only collect type data from them, and store the FUNCTION node within the COLLECTION, so we can typecheck the functions scope later.

Once we have finished collecting information from the declarations, we check that we do not have any circular name types. If there are circular name types, we note that an error has occurred.

We then check that statements obey our type rules. We do this by first traversing any expression or variable node it has as children, so that we have access to those nodes' types. We then check the usage of those types dependent on the statements kind. Return statements are checked to see if the expression is of the same type as the function. Allocate of length checks if the expression is a number. In assignments we check that the variables type is equivalent to the type of the expression. For while and if we check that the expression is a boolean.

Once we have finished typechecking declarations and statements in a scope, we traverse each FUNCTION node stored in the collection. We call scopeCollection on the COLLECTION the function was contained in. We put symbols into this collection representing the argument variables of the function. This COLLECTION is then used to typecheck the BODY node inside the FUNCTION node.

When we enter EXPRESSION nodes we calculate the types of the children of the node

first, by traversing into them. Afterwards if the EXPRESSION node contains a binary operator, we check that the children are of the correct types according to the table in figure 7. The expression then gets assigned its type according to the same table. If the expression has a TERM node as a child, its type is that of the term.

A terms type is calculated by traversing any child node it has, and then decided based on the table in figure 8.

Type equivalence is tested with function compTy defined in `types.c`. It takes two types and checks if they are equivalent. If a type is of type `Ty_name`, we continuously fetch the actual type, until both types are not of type `Ty_name`. If the first type is either a record or array and the second is null, then we return true, because we should be able to assign those values of those types as null. If both types are records we check whether they are equivalent by returning the result of compRecords. If both types are arrays we return the result of compArray. In all other cases we check whether the types are of the same kind, and return true if they are, and false if they are not.

compArray decides that two arrays are equivalent if their inner type is equivalent.

In compRecords we compare the fields of the records as lists ordered by their definition, where the type of index I of type 1's list has to be equivalent to the type of index I of type 2's list, which is checked by calling compTy. If every index is equivalent the two records are equivalent.
In order to avoid infinite recursion on recursive records, we note which records are currently being compared, that way if the same two types should be checked for equivalence during the comparison, we can skip them. This is because being able to finish the comparison, would mean that the types are equivalent, and that the comparison should have returned true. As we finish a comparison, we also erase the notes on which types were being compared, in case they were not equivalent.

5.4 Tests

Structural equivalence is successfully tested in `F_SimpleStructuralEquiv`, which tries to assign the value of on variable to that of another variable. These two variables are structurally equivalent, so the type checker should allow for this. We also successfully test that two non equivalent types cannot be assigned to the others value in the test `NotStructuralEquiv`.

We check that if an identifier was used to define a type, then it does not also create a value which could be used as a variable. This is tested in `C_ErrAssignToType`, where we create a type, and then try to use that type as a variable. If the type checker behaves correctly it will throw an error, which our type checker correctly does.

We successfully test function calls in `C_ErrFuncTooMany`, `C_ErrFuncTooMany`, `C_ErrFuncParamsInvalidType`, to confirm that we do not allow function calls with the wrong number of arguments, or the wrong type of arguments.

Programs with circular type declarations, are rejected, as can be seen in the test `C_ErrTypeLoop`.

In `DiffArrayDim` we test that the type checker will see two arrays with different different dimensions as non equivalent, however it sees them as equivalent, so the test fails.

We are able to confirm that our compiler checks that the value in a return statement is of the correct type with the test `FuncWrongReturn`.

6 Code Generation

In the code generation phase, the compiler translate the syntax of the program into a representation of assembly code. We will be using a substitute for registers called temporaries, which will be replaced with registers in a later phase. We have decided that the registers r12, r13, r14 and r15 should be reserved for temporaries, as they are not volatile to system calls. As such the rest of the registers will be used freely in this phase.

6.1 Strategy

We are going to traverse the abstract syntax tree, and for each node we will have a template to translate that node into assembly code. We will be using temporaries to move values between nodes. Such that a node would call its child node with temporary `a`, and the child would produce its assembly code, ending with the result being in `a`. We pose no limit on how many temporaries can exist. In the code produced the temporaries are represented by the special character `@`, followed by a number to identify which temporary it is.

We choose to represent variables with an 8 byte header to hold type information, and 8 byte value to store the value assigned to the variable. The representation can be seen in the diagram in figure 9.

Allocated blocks of memory have a field to for marking, in order to facilitate mark and sweep garbage collection. Then we have one field indicating size of the block as the size of the block in bytes divided by 8. And another field indicating the amount of values is stored in the block. These two values are redundant as one could derive one from the other. The reason we have this redundancy, is so we can reuse the block once it is free. The block might have been reused for a structure with fewer values, so the number of values is reassigned. By keeping the size the same, we will be able to use the entire block if it is freed again. The values contained in the block have the same representation as variables, with header containing type information, followed by the value. The representation can be seen in the diagram in figure 9.

We need to design a structure for how we access variables according to the scope we are in. This structure is a stack frame, from which we will be able to get the address of variables in the scope, and variables reachable in the outer scope.

6.1.1 Stack Frame

Each stack frame has a stack pointer which points to the base of the stack. Every value and variable associated with the frames scope, is reachable from a set offset from the base.
When a function gets called the arguments will be pushed onto the stack in reverse order, these arguments will be represented the same way as variables, each variable first pushes the value then the header.
After the arguments have been pushed, we push the address of the base of the functions enclosing frame, we call this a static link. With this value we can retrieve values from enclosing scopes, by calculating the depth of a variable and an offset from its frame pointer.
When the function is actually called, it pushes a return address onto the stack. At the beginning of the function we push the previous frame pointer and set the frame pointer to be equal to the current top of the stack.
We then save both the number of arguments and local variables so that a garbage collector would know how many arguments and local variables to inspect.
We then push local variables onto the stack, in the same way we pushed arguments, except in increasing order. The full representation can be seen in the diagram in figure 9.

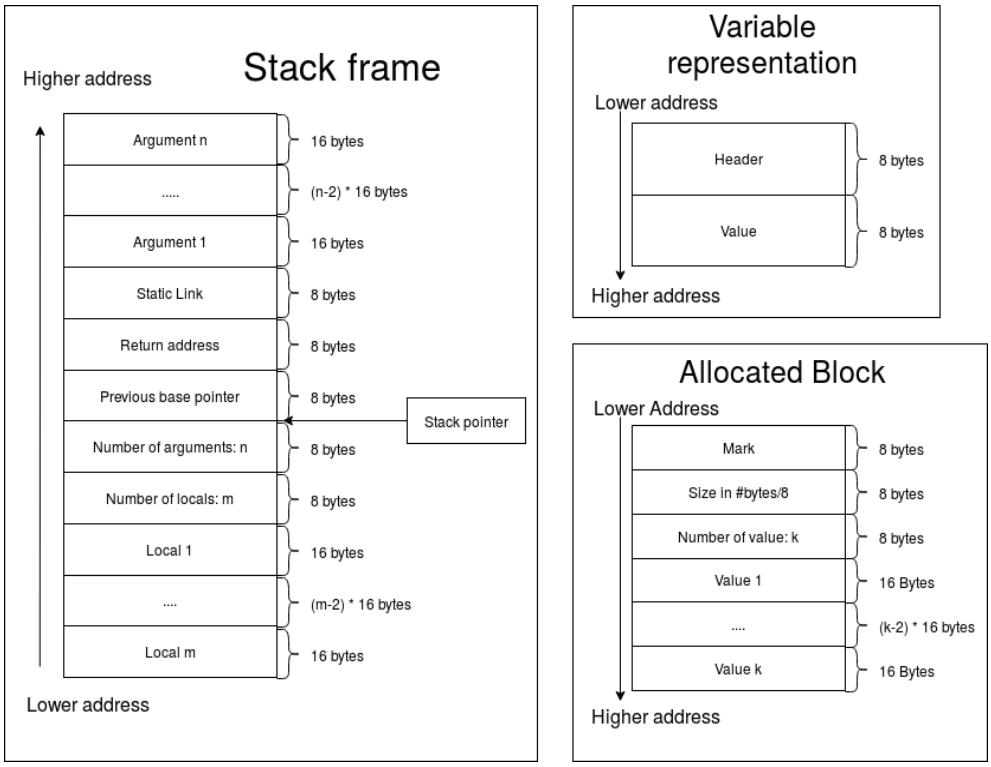

Figure 9: Stack frame, variable and memory block diagram

```
1 Expression(exp, temp){
2     Expression(exp->1,arg1)
3     if(condition(arg1)){
4         Expression(exp->2,arg2)
5     }
6     Do operation on arg1 and arg2, store result in temp.
7 }
```

Figure 10: Template for generating lazy binary expressions

6.2 Code Templates

6.2.1 Expression

The general template for expressions is to create a temporary for each child, the let those children generate code which results are stored in those temporaries. Then calculate a result from those temporaries, and return the result in a temporary.
We do however have a few exceptions where we have lazy operations, which will not calculate the second argument under certain condition. These operations are `AND` and `OR`, where we might know the result after the first argument. As such the effective template is similar to the pseudo code in figure 10, but for the other operands we do not generate the code for the if condition.
Expressions containing a term just takes the temporary with the result of the term node, and moves that into its own result temporary.

6.2.2 Term

Depending on the kind of the term, the code generation has to do very different things. The objective of term is to take its children and return some value based on it. If the term is a variable it has to get the variable and place the variables value in its result temporary.
When the term is a function call it has to create variables for each argument and push them onto the stack. Then calculate the depth of the scope the function was declared in to find and push the correct static link. Afterwards it generates the function call, after the function call the result is moved to the return temporary and the frame pointer is moved back to before the arguments on the stack.
If the term is an expression with parenthesis, it just moves the result of the expression into its return temporary. Terms which are a `NOT` of another term takes the resulting temporary and moves the opposite value into its result.
When the term expands to `|Expression|`, term generates the code to calculating the length of the expression if it is an array. If the type is integer, it generates the code for calculating the the absolute value.
For number, boolean and null it simply moves the associated value into the return temporary.

6.2.3 Variables

Variable nodes serve to return the address of a variable. If it contains a regular the compiler calculates at what depth of frame and offset the variable occurs. It then generates generates code for fetching the variables address. After which the address is stored in the return temporary.
If the variable node is a variable being accessed through an index. We must first generate code for getting the address of the variable, and then generate the code for the index expression. We then go to the address inside the variable and use the expression to calculate the offset at which the desired variable occurs. We then move the address of the desired variable into the return temporary.
If the node is a variable being accessed like a record, with an identifier. We first generate code to get the records address, then according to the type of the record we translate the identifier into an index, and fetch the identifiers variable in the same way we did with lists.

6.2.4 Functions

Functions start by saving the previous frame pointer, setting storing the new one. It then pushes the the amount of arguments and amount of local variables. Afterwards the local variables are pushed onto the stack. It then generates all the code for its statements, and if it returns it places the result of the function in rax, and setes the stack pointer to be what it was at function entry.

6.2.5 Statements

Statements behave based on their kind. Return takes its expression, stores the result in rax, and exits the scope. Write is implemented as a call, so the code generator generates code for calculating the specified expression, and places it inside rax. By calling write, the value in rax is written out.
Allocation works by calculating the amount of bytes the variables type fills divided by 8. We then pass that value in rax, and call getMem, which return an address to a block of the correct size. For allocation of a certain length we have another function call AllocL, which takes a the requested list length as argument in rax, and the header value for the indecis in rdi. This node starts by getting getting the address of the variable being allocated to, calculates the length to allocate then sets up the call to AllocL and sets the variable value to the returned address.
Assignments generates the code for getting the variable address into a temporary, and then generates the code to get the expression result into a temporary. We then set the value in the variable to the result of the expression.
If statements generates the result of its condition, and then skips the statements inside if the condition is false. The IfElse statements Jumps to the else statements if the condition is false. Whether the condition was true or false it should jump to the end of the if code after finishing the decided code block.
While statements creates a label in front of the condition calculation and check. It then

```
1  typedef struct frame_{
2      int level;
3      linkedlist *formal;
4      linkedlist *local;
5      linkedlist *func;
6      frame *enclosing;
7      COLLECTION *c;
8  } frame;
```

Figure 11: Frame structure

generates the calculation of the expression and checks it. If it is false, the program jumps to an end label outside the loop. Otherwise it runs the statements within the while loop, and jumps back to the start of the while loop at the end.

6.3 The Algorithm

The code generation traverses the abstract syntax tree and generates code according to the templates. At each entry into a scope we create a frame structure to keep track information about the scope and its stack frame. This structure is defined in frame.h as see in figure 11. We then set the level to be equal to the distance to the most outer scope. Formal is the list of argument names in the proper order, which is build by going through the parameters in the header node. Locals are the list of local variables names in their proper order, build by making a list out of variable declarations in the order they appear. Func is the the list of names of functions residing in the current scope, build by noting each function declaration in the scope.. We save a reference to the enclosing frame, and the collection of the scope we are in.
This structure is able to calculate the depth and offset a variable occurs, by checking if the variable name occurs in either its list of locals or formals, if it does it returns a structure indicating the correct offset. Otherwise it will increment the depth, and look in the enclosing frame. If it is found, we will have a depth indicating the amount of static links to move through, and an offset to apply afterwards to get the address of the variable.

The getMem call in our generated code makes use of the brk system call to allocate new memory. This would have been expanded to incorporate a free list maintained by garbage collection, if we had managed to implement garbage collection. Allocated memory also had a field to mark, which would be used in garbage collection to perform mark and sweep garbage collection. AllocL calculates the size to pass to getMem, by multiplying the length passed in rax by 2 and adding 3. After getMem returns the address of the memory block, we initialize all the values in the memory block to have the correct header.

Write takes a value in from rax and treats it like a signed integer. It then writes the

value of rax out by first pushing each digit onto the stack, and then writing them out afterwards.

6.4 Test

We do not perform tests on code generation until after register allocation, as the generated code cannot be readily run and tested the temporaries have been replaced.

7 Phases before Emit

7.0.1 Register Allocation

We implement register allocation, to replace the temporaries in the file with registers. They need to be allocated in such a way, that any pair of temporaries live at the same time, do not have the same register. We build a graph, with nodes representing all the temporaries, and edges between those pairs. If we color the graph, such that no neighboring nodes have the same color, we can decide those temporaries registers based on the color. This lets us replace the temporaries in the file with registers according to their color, which will run as the semantics of the program intends.

7.1 Analyses

7.1.1 Liveness Analysis

Liveness analysis is implemented to collect data from the temporary generated code for the register allocation. The analysis determines whether in each line of the code, the temporaries are being used or defined and for how long a register would have to be reserved for them.

In order to achieve this, The temporary code needs to be parsed to get all the necessary data.

```
1 typedef struct livenessNode{
2     int id;
3     char line[100];
4     struct parsedLine *pline;
5     struct stringBuffer *use;
6     struct stringBuffer *def;
7     struct stringBuffer *out;
8     struct stringBuffer *in;
9     struct nodeList *succ;
```

```
10      struct nodeList *pred;
11      char jumpto[100];
12      enum {movq, addq, subq, imulq, idivq, cmp, push, pop,
            jump, label, XOR, AND, OR, neg} op;
13  } livenessNode;
```

The data needed to perform the analysis is stored in this structure. It contains the line number, the code line, the line parsed in to words, used and defined temporaries, temporaries that are live between nodes, other nodes that this node is linked to, label to be jumped too and enums to determine the function of the line.

7.2 Algorithms

7.2.1 The Parser

Each line of the generated code is scanned and goes through four parsing steps.

1. Making the node Initially a node(livenessNode) is created for the line and that node will be linked to the predicessor and successor nodes(lines from the code). The predicessors are also lists of pointers to nodes since each node needs to be able to be linked with more than one node.

2. Parsing the line
During this step, all the words from the line are separated and put in lists of char buffers.

3. Get operand enum
The first parsed word is checked in order to determine whether the function of the corresponding line is relevant for the liveness analysis.

4. Operator handler
Now that the operand has been identified, the operators are being handled. If the operators are temporary registers, they are put into appropriate lists in the livenessNode. The temporaries that are defined are put into the defined list and those who are used are put into the used list. All labels in the code are put in a linked list of pointers that point to the nodes that contain them.

7.2.2 Liveness Analysis

The liveness analyser starts by traversing through the node list looking for nodes with jumps and links them to the corresponding label node so that the liveness analysis can analyse calls and loops correctly.

The liveness analysis algorithm traverses through the list of nodes backwards. Two functions are applied to each node.

```
1  stringBuffer *in(livenessNode *node){
2      stringBuffer *ret = NEW(stringBuffer);
3      stringBuffer *right = NEW(stringBuffer);
4      right = getSub(node->out, node->def);
5      ret = getUnion(node->use, right);
6      return ret;
7  }
```

To generate the list for *in*, the list of defined temporaries are subtracted from the list of out temporaries. There is made a union from that list and the use list.

```
1  stringBuffer *out(livenessNode *node){
2      stringBuffer *ret = NEW(stringBuffer);
3      stringBuffer *tmp = NEW(stringBuffer);
4      nodeList *succ = node->succ;
5      livenessNode *succNode = succ->node;
6      ret = succNode->in;
7      succ = succ->next;
8      if(succ != NULL && succ->node != NULL){
9          succNode = succ->node;
10         tmp = succNode->in;
11         ret = getUnion(ret, tmp);
12     }
13     return ret;
14 }
```

To generate the list for *out*, all *in* lists are put in a union.

Since both functions in() and out() need to call each other to get the correct results, this process takes a few iterations through all the nodes to get it right. This is also because of the way that the code can loop or jump to other positions in code.

During an iteration, when the *in*- and *out* lists are updated, they are compared to the lists from the previous iteration. If all comparisons are equal in an iteration, the algorithm is complete and the nodes are ready for the register allocation.

7.2.3 Register Allocation

Register allocation starts by building the graph, by going through the liveness node for each program point. Each node contains the set of currently live temporaries, and for each temporary we create a node, unless the node already exists. Afterwards we create edges between each pair of temporaries occurring in the liveness node. Once their are no more liveness nodes the graph has been built.

Before we color the graph we simplify it. This is done by making a stack of nodes, and pushing a node which has fewer neighbors than the amount of allowed colors. We use the registers r12, r13, 14 and r15 for temporaries and as such we have 4 colors to work with, as such we have 4 colors to color our graph with. Once a node is pushed on the stack all its neighbors has their amount of neighbors decremented by one. If we at any point only have nodes on the graph, where the amount of neighbors is not less than amount of colors, we choose a node not on the stack and mark it for spilling, and perform the same actions on the node as when we normally simplify. The simplification ends when all nodes are on the stack. When a node is marked for spilling, it indicates that it might not be able to have a color assigned to it.

All the nodes marked for spilling might not be able to be colored, but every other node can be colored no matter their neighbors color. This means that if we can find a coloring for the sub-graph which only contains marked nodes, we can color the entire graph. We have not implemented an algorithm for finding the solution to this sub-graph, and instead choose to represent each of those nodes as unique addresses in memory.

We color the graph by popping nodes off of the stack, and assigning them a color their neighbors have not been assigned to. Spilled nodes get colors, which represents their addresses instead. Once the stack is empty all temporaries have an assigned color.

We then print the program, replacing each temporary with the register or memory address its color represents. This output is the finished code which gets printed at the emit phase.

7.3 Test

We test the emit of code by register under the emit section.

8 Emit

The emit of the code happens at the end of register allocation, where the generated code without temporaries is written to standard output. During this section we have to test that the code generated and the temporaries allocated result in correct assembly code.

8.1 Example Code

As an example we show the code generated from the `O_Factorial.src` test file.

8.1.1 Code generated from the test O Factorial.src

```
1  .section .data
2      spilling: .space 0
3      initialBrk: .quad 0
4      currentBrk: .quad 0
5      newBrk: .quad 0
6      freelist: .quad 0
7      Callstack: .space 400
8      top: .quad 0
9  .section .text
10 .globl main
11
12 main:
13     movq $12, %rax
14     movq $0, %rdi
15     syscall
16     movq %rax, (initialBrk)
17     movq %rax, (currentBrk)
18     movq %rax, (newBrk)
19     movq $Callstack, (top)
20     push %rbp
21     movq %rsp, %rbp
22     call pushframe
23     push $0
24     push $0
25     push %r12
26     push %r13
27     push %r14
28     push %r15
29     movq $5, %r15
30     push %r15
31     push $0
32     movq %rbp, %rdi
33     push %rdi
34     call factorial
35     movq %rax, %r15
36     addq $8, %rsp
37     addq $16, %rsp
38     push %rbp
39     movq %rsp, %rbp
40     push %rax
41     push %rcx
42     push %rdx
43     push %rdi
44     push %rsi
45     push %r15
46     movq %r15, %rax
```

```
47      call write
48      pop %r15
49      pop %rsi
50      pop %rdi
51      pop %rdx
52      pop %rcx
53      pop %rax
54      movq %rbp, %rsp
55      pop %rbp
56      pop %r15
57      pop %r14
58      pop %r13
59      pop %r12
60      pop %rax
61      pop %rax
62      call popframe
63      movq %rbp, %rsp
64      pop %rbp
65      movq $60, %rax
66      movq $0, %rdi
67      syscall
68
69  getMem:
70      push %r12
71      push %r13
72      push %r14
73      push %r15
74      push %r11
75      push %r10
76      push %r9
77      push %r8
78      push %rdi
79      push %rsi
80      push %rdx
81      push %rcx
82      push %rbx
83      movq (currentBrk), %rdi
84      movq $8, %rdx
85      imulq %rdx
86      addq %rax, %rdi
87      movq $12, %rax
88      syscall
89      movq %rax, (newBrk)
90      movq %rax, (currentBrk);
91      pop %rbx
92      pop %rcx
```

```
 93      pop %rdx
 94      pop %rsi
 95      pop %rdi
 96      pop %r8
 97      pop %r9
 98      pop %r10
 99      pop %r11
100      pop %r15
101      pop %r14
102      pop %r13
103      pop %r12
104      ret
105  allocL:
106      movq %rax, %rsi
107      movq $2, %rcx
108      imulq %rcx
109      addq $3, %rax
110      movq %rax, %rcx
111      push %rdi
112      call getMem
113      pop %rdi
114      movq $0, (%rax)
115      movq %rcx, 8(%rax)
116      movq %rsi, 16(%rax)
117      movq %rax, %rcx
118      addq $24, %rcx
119  allocLloop:
120      cmp $0, %rsi
121      je allocLloopEnd
122      movq %rdi, (%rcx)
123      movq $0, 8(%rcx)
124      addq $16, %rcx
125      subq $1, %rsi
126      jmp allocLloop
127  allocLloopEnd:
128      ret
129  popframe:
130      subq $8, (top)
131      movq (top), %rax
132      movq $0, (%rax)
133      ret
134  pushframe:
135      movq (top), %rax
136      movq %rbp, (%rax)
137      addq $8, (top)
138      ret
```

```
139 write:
140     push %rbp
141     movq %rsp, %rbp
142     push %rax
143     push %r11
144     push %r10
145     push %r9
146     push %r8
147     push %rdi
148     push %rsi
149     push %rdx
150     push %rcx
151     push %rbx
152     push %r12
153     push %r13
154     push %r14
155     push %r15
156     movq $0, %r15
157     movq %rax, %r14
158     push $10
159     addq $1, %r15
160     cmp $0, %r14
161     jge positive
162     push $45
163     movq $1, %rax
164     movq $1, %rdi
165     movq %rsp, %rsi
166     movq $1, %rdx
167     syscall
168     addq $8, %rsp
169     movq %r14, %rsi
170     neg %rsi
171     movq %rsi, %r14
172     movq %r14, %rax
173     jmp writeloop
174 positive:
175     movq %r14, %rax
176 writeloop:
177     movq $0, %rdx
178     movq $10, %rcx
179     idivq %rcx
180     addq $48, %rdx
181     push %rdx
182     addq $1, %r15
183     cmp $0, %rax
184     jne writeloop
```

```
185 printloop:
186     movq $1, %rax
187     movq $1, %rdi
188     movq %rsp, %rsi
189     movq $1, %rdx
190     syscall
191     addq $8, %rsp
192     addq $-1, %r15
193     cmp $0, %r15
194     jne printloop
195     pop %r15
196     pop %r14
197     pop %r13
198     pop %r12
199     pop %rbx
200     pop %rcx
201     pop %rdx
202     pop %rsi
203     pop %rdi
204     pop %r8
205     pop %r9
206     pop %r10
207     pop %r11
208     pop %rax
209     movq %rbp, %rsp
210     pop %rbp
211     ret
212 factorial:
213     push %rbp
214     movq %rsp, %rbp
215     push $1
216     push $0
217     push %r12
218     push %r13
219     push %r14
220     push %r15
221 if0:
222     movq $0, %r15
223     movq %rbp, %rdi
224     addq $24, %rdi
225     movq %rdi, %r15
226     movq %r15, %rdx
227     movq %rdx, %r15
228     movq %r15, %rcx
229     movq 8(%rcx), %rdi
230     movq %rdi, %r15
```

```
231     movq $0, %r14
232     cmp %r15, %r14
233     jne explabel0
234     movq $1, %r15
235     jmp expend0
236 explabel0:
237     movq $0, %r15
238 expend0:
239     movq %r15, %rax
240     movq %rax, %r15
241     movq $0, %r14
242     cmp $1, %r15
243     je lazyOR0
244     movq %rbp, %rdi
245     addq $24, %rdi
246     movq %rdi, %r14
247     movq %r14, %rdx
248     movq %rdx, %r14
249     movq %r14, %rcx
250     movq 8(%rcx), %rdi
251     movq %rdi, %r14
252     movq $1, %r13
253     cmp %r14, %r13
254     jne explabel1
255     movq $1, %r14
256     jmp expend1
257 explabel1:
258     movq $0, %r14
259 expend1:
260     movq %r14, %rax
261     movq %rax, %r14
262 lazyOR0:
263     OR %r15, %r14
264     movq %r14, %r15
265     cmp $1, %r15
266     jne ifelse0
267     movq $1, %r15
268     movq %r15, %rax
269     pop %r15
270     pop %r14
271     pop %r13
272     pop %r12
273     movq %rbp, %rsp
274     pop %rbp
275     ret
276     jmp ifend0
```

```
277 ifelse0:
278     movq %rbp, %rdi
279     addq $24, %rdi
280     movq %rdi, %r15
281     movq %r15, %rdx
282     movq %rdx, %r15
283     movq %r15, %rcx
284     movq 8(%rcx), %rdi
285     movq %rdi, %r15
286     movq %rbp, %rdi
287     addq $24, %rdi
288     movq %rdi, %r14
289     movq %r14, %rdx
290     movq %rdx, %r14
291     movq %r14, %rcx
292     movq 8(%rcx), %rdi
293     movq %rdi, %r14
294     movq $1, %r13
295     movq %r14, %rax
296     subq %r13, %rax
297     movq %rax, %r14
298     push %r14
299     push $0
300     movq %rbp, %rdi
301     movq 16(%rdi), %rdi
302     push %rdi
303     call factorial
304     movq %rax, %r14
305     addq $8, %rsp
306     addq $16, %rsp
307     movq %r15, %rax
308     imulq %r14, %rax
309     movq %rax, %r15
310     movq %r15, %rax
311     pop %r15
312     pop %r14
313     pop %r13
314     pop %r12
315     movq %rbp, %rsp
316     pop %rbp
317     ret
318 ifend0:
319     pop %r15
320     pop %r14
321     pop %r13
322     pop %r12
```

```
323     addq $16, %rsp
324     movq %rbp, %rsp
325     pop %rbp
326     ret
```

8.2 Test

The generated code is able to successfully run `O_ArrayComparisonsA` but not `O_ArrayComparisonsB`. This is because the compiler does not have garbage collection implemented, so it cannot reuse memory, and therefore runs out of memory. The first test works fine, and shows that array comparisons work as intended in our program.

The tests `F_ShortCircuitAND` and `F_ShortCircuitAND` successfully tests that we skip the second argument, if the first argument already determines the result.

`O_AbsTest` and `O_AbsoluteValueTest` complete successfully, in testing calculating the absolute value of integer expressions. If the expression was an array we would have recieved the length of the array, as successfully tested in `O_ArrayLength`.

`O_Factorial`, `O_BinarySearchTree`, `O_FuncReturnRecord` and `O_Function` all complete successfully, and tests that functions are able to properly get the values passed as arguments. If the return value is a record, the allocated address should still exist after exiting the scope.

We test the scope behavior with the tests `O_StaticLink`, `O_StaticLinkA` and `O_StaticLinkB` to see if the stack frame layout works as intended. They all complete correctly, so the stack frame works correctly.

`O_RecordsWithArray` successfully tests if the compiler correctly retrieves the correct value from the variable by addresses in the correct order.

The tests `O_ WhileDo` and `O_IfThen` checks if these statements are correctly generated. Both tests passes.

`O_FuncRedefinedInItself` fails because the code generator just makes a label according to the name. So even though type checking allows for multiple functions with the same name at different scopes, the code generator does not.

The tests identified with `R_` all fail because there are no run time checks in the generated code.

9 Conclusion

The compiler is able to correctly generate code, for a large set of programs in its language. It lacks garbage collection and memory recycling, which makes programs with many or big allocations crash. A facility for generating unique labels is not in place, so functions with the same name collide, and assembly cannot compile it. Our compiler has a big overhead, in that code generation writes the generated code into a file, and then liveness analysis reads that file into a structure. When we could have had code generation create the structure instead.
We also do not try to color the spilled nodes, which might have been able to be colored, which means our emited code is slower as it has to access memory for those temporaries. Another improvement would have been to make peephole optimization, where we could shave some operations off of our assembly code. This could for example be to combine moves which pivot a single value between registers into one, that only moves from the source to the final destination.

A Source Code

A.1 main.c

```
1  #include "tree.h"
2  #include "pretty.h"
3  #include "weed.h"
4  #include "typecheck.h"
5  #include "codegen.h"
6  #include <stdio.h>
7  #include "regallocation.h"
8  #include "graphcolor.h"
9  /*
10 Compiler with finished parsing, weeding and typechecking
11 Code generation currently writes to file output.asm,
12 this was done with the intention of replacing temporaries
       using
13 liveness analysis and register allocation, which were not
       completed.
14 It is possible to run some programs by manually replacing
       temporaries with registers
15 */
16
17 int lineno;
18
19 int yyparse();
20
21 BODY *theexpression;
22
23 void errComp(){
24     //fprintf(stdout, ".globl _start\n\n");
25     fprintf(stdout, ".globl main\n\n");
26
27     //main program
28     //fprintf(stdout, "_start:\n");
29     fprintf(stdout, "main:\n");
30     fprintf(stdout, "    push %%rbp\n");
31     fprintf(stdout, "    movq %%rsp, %%rbp\n");
32     fprintf(stdout, "    push $10\n");
33     fprintf(stdout, "    push $114\n");
34     fprintf(stdout, "    push $111\n");
35     fprintf(stdout, "    push $114\n");
36     fprintf(stdout, "    push $114\n");
37     fprintf(stdout, "    push $101\n");
38     fprintf(stdout, "    movq $6, %%r12\n");
```

```
39
40      fprintf(stdout, "printloop:\n");
41      fprintf(stdout, "    movq $1, %%rax\n");
42      fprintf(stdout, "    movq $1, %%rdi\n");
43      fprintf(stdout, "    movq %%rsp, %%rsi\n");
44      fprintf(stdout, "    movq $1, %%rdx\n");
45      fprintf(stdout, "    syscall\n");
46      fprintf(stdout, "    addq $8, %%rsp\n");
47      fprintf(stdout, "    addq $-1, %%r12\n");
48      fprintf(stdout, "    cmp $0, %%r12\n");
49      fprintf(stdout, "    jne printloop\n");
50
51
52      fprintf(stdout, "    movq %%rbp, %%rsp\n");
53      fprintf(stdout, "    pop %%rbp\n");
54      fprintf(stdout, "    movq $60, %%rax\n");
55      fprintf(stdout, "    movq $0, %%rdi\n");
56      fprintf(stdout, "    syscall\n");
57  }
58
59  int main()
60  { lineno = 1;
61    fprintf(stderr, "Parsing\n");
62    if(yyparse()){
63        errComp();
64        return 0;
65    }
66    fprintf(stderr, "Weeding\n");
67    weedBODY(theexpression);
68    fprintf(stderr, "Typechecking\n");
69    if(checkTREE(theexpression)){
70        errComp();
71        return 0;
72    }
73    prettyBODY(theexpression);
74    fprintf(stderr, "Generating code\n");
75    codeGEN(theexpression);
76    //regAllocation();
77    registerAllocation();
78    return 0;
79  }
```

A.2 compiler.l

```
1  %{
2  #include "y.tab.h"
```

```
3  #include <string.h>
4  extern int comment = 0;
5  extern int lineno;
6  extern int fileno();
7  %}
8
9  %option noyywrap nounput noinput
10 %x COMMENT
11
12 %%
13 [ \t]+         /* ignore */;
14 \n               lineno++;
15
16 "*"              { yylval.stringconst = (char *)malloc(
       strlen(yytext)+1);
17                   sprintf(yylval.stringconst,"%s",
                         yytext);
18                   return '*'; };
19 "/"              { yylval.stringconst = (char *)malloc(
       strlen(yytext)+1);
20                   sprintf(yylval.stringconst,"%s",
                         yytext);
21                   return '/'; }
22 "+"              { yylval.stringconst = (char *)malloc(
       strlen(yytext)+1);
23                   sprintf(yylval.stringconst,"%s",
                         yytext);
24                   return '+'; }
25 "-"              { yylval.stringconst = (char *)malloc(
       strlen(yytext)+1);
26                   sprintf(yylval.stringconst,"%s",
                         yytext);
27                   return '-'; };
28 "=="             { yylval.stringconst = (char *)malloc(
       strlen(yytext)+1);
29                   sprintf(yylval.stringconst,"%s",
                         yytext);
30                   return EQ; };
31 "!="             { yylval.stringconst = (char *)malloc(
       strlen(yytext)+1);
32                   sprintf(yylval.stringconst,"%s",
                         yytext);
33                   return NEQ; };
34 ">"              { yylval.stringconst = (char *)malloc(
       strlen(yytext)+1);
35                   sprintf(yylval.stringconst,"%s",
```

```
                                yytext);
36                              return '>'; };
37  "<"             { yylval.stringconst = (char *)malloc(
        strlen(yytext)+1);
38                              sprintf(yylval.stringconst,"%s",
                                yytext);
39                              return '<'; };
40  ">="            { yylval.stringconst = (char *)malloc(
        strlen(yytext)+1);
41                              sprintf(yylval.stringconst,"%s",
                                yytext);
42                              return GEQ; };
43  "<="            { yylval.stringconst = (char *)malloc(
        strlen(yytext)+1);
44                              sprintf(yylval.stringconst,"%s",
                                yytext);
45                              return LEQ; };
46  "&&"            { yylval.stringconst = (char *)malloc(
        strlen(yytext)+1);
47                              sprintf(yylval.stringconst,"%s",
                                yytext);
48                              return AND; };
49  " || "            { yylval.stringconst = (char *)malloc(
        strlen(yytext)+1);
50                              sprintf(yylval.stringconst,"%s",
                                yytext);
51                              return OR; };
52  "++"            { yylval.stringconst = (char *)malloc(
        strlen(yytext)+1);
53                              sprintf(yylval.stringconst,"%s",
                                yytext);
54                              return INC; };
55  "--"            { yylval.stringconst = (char *)malloc(
        strlen(yytext)+1);
56                              sprintf(yylval.stringconst,"%s",
                                yytext);
57                              return DEC; };
58  "("             return '(';
59  ")"             return ')';
60  "["             return '[';
61  "]"             return ']';
62  "{"             return '{';
63  "}"             return '}';
64  "|"             return '|';
65  "!"             return '!';
66  ","             return ',';
```

```
67  "."                 return '.';
68  "="                 return '=';
69  ":"                 return ':';
70  ";"                 return ';';
71  "of length"         return OFLENGTH;
72  "allocate"          return ALLOCATE;
73  "write"             return WRITE;
74  "while"             return WHILE;
75  "do"                return DO;
76  "for"               return FOR;
77  "return"            return RETURN;
78  "if"                return IF;
79  "else"              return ELSE;
80  "then"              return THEN;
81  "var"               return VARI;
82  "record of"         return RECORDOF;
83  "array of"          return ARRAYOF;
84  "func"              return FUNC;
85  "int"               return INT;
86  "bool"              return BOOL;
87  "end"               return END;
88  "type"              return TYPEY;
89
90  "true"                { yylval.boolconst = 1;
91                          return tBOOL; }
92
93  "false"               { yylval.boolconst = 0;
94                          return tBOOL; }
95
96  "null"                { yylval.nullconst = NULL;
97                          return tNULL; }
98
99  0|([1-9][0-9]*)         { yylval.intconst = atoi(yytext);
100                         return tINTCONST; }
101 [a-zA-Z_][a-zA-Z0-9_]* { yylval.stringconst = (char *)
        malloc(strlen(yytext)+1);
102                           sprintf(yylval.stringconst,"%s",
                                yytext);
103                           return tIDENTIFIER; }
104 "#".*
105
106 "(*"                    {BEGIN(COMMENT);
107                           comment++;}
108 <COMMENT>"*)"           { comment--;
109                           if(comment == 0){BEGIN(INITIAL);}
110                           }
```

```
111 <COMMENT>"(*"              {comment++;}
112 <COMMENT>(.|\n)          ;
113 <COMMENT><<EOF>>         return ERROR;
114
115
116 .                         return ERROR;/* ignore */;
117
118 %%
```

A.3 compiler.y

```
 1 %{
 2 #include <stdio.h>
 3 #include "tree.h"
 4
 5 extern char *yytext;
 6 extern BODY *theexpression;
 7 extern int lineno;
 8
 9 void yyerror() {
10     fprintf(stderr, "syntax error before %s, at line %d \n
          ", yytext, lineno);
11 }
12 %}
13
14 %union {
15     int intconst;
16     char *stringconst;
17     int boolconst;
18     void *nullconst;
19     struct EXP *exp;
20     struct TERM *term;
21     struct EXPLIST *exp_list;
22     struct VAR *variable;
23     struct ACTLIST *act_list;
24     struct FUNCTION *function;
25     struct HEAD *head;
26     struct TAIL *tail;
27     struct TYPE *type;
28     struct PAR_DECL_LIST *par_decl_list;
29     struct VAR_DECL_LIST *var_decl_list;
30     struct VAR_TYPE *var_type;
31     struct BODY *body;
32     struct DECLARATION *declaration;
33     struct DECL_LIST *decl_list;
34     struct STATE *statement;
```

```
35      struct STATE_LIST *statement_list;
36  }
37
38  %token <intconst> tINTCONST
39  %token <stringconst> tIDENTIFIER '+' '-' '*' '/' AND OR
        EQ LEQ GEQ NEQ INC DEC '<' '>'
40  %token <boolconst> tBOOL
41  %token <nullconst>  tNULL
42  %token <error> ERROR
43
44  %token RETURN WHILE ALLOCATE WRITE OFLENGTH IF ELSE THEN
        DO FOR TYPEY ARRAYOF RECORDOF VARI FUNC INT BOOL END
45  %type <exp> expression
46
47  %type <variable> variable
48  %type <term> term
49  %type <act_list> act_list
50  %type <exp_list> exp_list
51  %type <function> function
52  %type <head> head
53  %type <tail> tail
54  %type <type> type
55  %type <par_decl_list> par_decl_list
56  %type <var_decl_list> var_decl_list
57  %type <var_type> var_type
58  %type <body> body program
59  %type <declaration> declaration
60  %type <decl_list> decl_list
61  %type <statement> statement
62  %type <statement_list> statement_list
63
64  %start program
65
66  %nonassoc OR EQ LEQ GEQ NEQ
67  %right '<' '>'
68  %left '+' '-' INC DEC
69  %left '*' '/' AND
70  %right '[' '{' '('
71  %left ']' '}' ')'
72  %nonassoc THEN
73  %nonassoc ELSE
74
75
76  %%
77   program: body
78            { theexpression = $1;}
```

```
79  ;
80
81
82  variable : tIDENTIFIER
83               {$$ = makeVARid($1);}
84           | variable '[' expression ']'
85               {$$ = makeVARlist($1, $3);}
86           | variable '.' tIDENTIFIER
87               {$$ = makeVARvarid($1, $3);}
88  ;
89
90  expression  : expression '+' expression
91                  {$$ = makeEXPbinop($1,$2,$3);}
92              | expression '-' expression
93                  {$$ = makeEXPbinop($1,$2,$3);}
94              | expression '*' expression
95                  {$$ = makeEXPbinop($1,$2,$3);}
96              | expression '/' expression
97                  {$$ = makeEXPbinop($1,$2,$3);}
98              | expression '<' expression
99                  {$$ = makeEXPbinop($1,$2,$3);}
100             | expression '>' expression
101                 {$$ = makeEXPbinop($1,$2,$3);}
102             | expression AND expression
103                 {$$ = makeEXPbinop($1,$2,$3);}
104             | expression OR expression
105                 {$$ = makeEXPbinop($1,$2,$3);}
106             | expression GEQ expression
107                 {$$ = makeEXPbinop($1,$2,$3);}
108             | expression LEQ expression
109                 {$$ = makeEXPbinop($1,$2,$3);}
110             | expression NEQ expression
111                 {$$ = makeEXPbinop($1,$2,$3);}
112             | expression EQ expression
113                 {$$ = makeEXPbinop($1,$2,$3);}
114             | term
115                 {$$ = makeEXPterm($1);}
116 ;
117
118 exp_list : expression
119              {$$ = makeEXPLISTexp($1);}
120          | expression ',' exp_list
121              {$$ = makeEXPLISTlist($1, $3);}
122 ;
123
124 term   : variable
```

```
125                 {$$ = makeTERMvar($1);}
126         | tINTCONST
127                 {$$ = makeTERMnum($1);}
128         | tIDENTIFIER '(' act_list ')'
129                 {$$ = makeTERMidfunc($1, $3);}
130         | '(' expression ')'
131                 {$$ = makeTERMexp($2);}
132         | '!' term
133                 {$$ = makeTERMnot($2);}
134         | '|' expression '|'
135                 {$$ = makeTERMlenexp($2);}
136         | tBOOL
137                 {$$ = makeTERMbool($1);}
138         | tNULL
139                 {$$ = makeTERMnull($1);}
140 ;
141
142 act_list : exp_list
143                 {$$ = makeACTLISTlist($1);}
144           | %empty
145                 {$$ = makeACTLISTnil();}
146 ;
147
148 statement : RETURN expression ';'
149                     {$$ = makeSTATEreturn($2);}
150               | WRITE expression ';'
151                     {$$ = makeSTATEwrite($2);}
152               | ALLOCATE variable ';'
153                     {$$ = makeSTATEallocate($2);}
154               | ALLOCATE variable OFLENGTH expression ';'
155                     {$$ = makeSTATEallocOfLength($2,$4);}
156               | variable INC ';'
157                     {$$ = makeSTATEinc($1);}
158               | variable DEC ';'
159                     {$$ = makeSTATEdec($1);}
160               | variable '=' expression ';'
161                     {$$ = makeSTATEassign($1,$3);}
162               | IF expression THEN statement
163                     {$$ = makeSTATEif($2,$4);}
164               | IF expression THEN statement ELSE statement
165                     {$$ = makeSTATEifElse($2,$4,$6);}
166               | WHILE expression DO statement
167                     {$$ = makeSTATEwhile($2,$4);}
168               | FOR '(' decl_list ')' INC '[' term ',' term
                      ']' statement
169                     {$$ = makeSTATEforinc($3,$7,$9,$11);}
```

```
170             | FOR '(' decl_list ')' DEC '[' term ',' term
                    ']' statement
171                 {$$ = makeSTATEfordec($3,$7,$9,$11);}
172             | '{' statement_list '}'
173                 {$$ = makeSTATEstateList($2);}
174 ;
175
176 statement_list : statement
177                 {$$ = makeSTATE_LISTstatement($1);}
178               | statement statement_list
179                 {$$ = makeSTATE_LISTstatementList($1,$2
                        );}
180 ;
181
182 declaration : TYPEY tIDENTIFIER '=' type ';'
183                 {$$ = makeDECLARATIONtypeId($2, $4);}
184           | function
185                 {$$ = makeDECLARATIONdeclFuncD($1);}
186           | VARI var_decl_list ';'
187                 {$$ = makeDECLARATIONdeclVarD($2);}
188 ;
189
190 decl_list : declaration decl_list
191                 {$$ = makeDECL_LISTdeclList($1, $2);}
192             | %empty
193                 {$$ = makeDECL_LISTnil();}
194 ;
195
196 body : decl_list statement_list
197       {$$ = makeBODY($1, $2);}
198 ;
199
200 var_type : tIDENTIFIER ':' type
201             {$$ = makeVAR_TYPEid($1, $3);}
202 ;
203
204 var_decl_list : var_type ',' var_decl_list
205                 {$$ = makeVAR_DECL_LISTlist($1, $3);}
206               | var_type
207                 {$$ = makeVAR_DECL_LISTvt($1);}
208 ;
209
210 par_decl_list : var_decl_list
211                 {$$ = makePAR_DECL_LISTvdl($1);}
212               | %empty
213                 {$$ = makePAR_DECL_LISTnil();}
```

```
214 ;
215
216 type : tIDENTIFIER
217              {$$ = makeTYPEid($1);}
218       | INT
219              {$$ = makeTYPEint();}
220       | BOOL
221              {$$ = makeTYPEbool();}
222       | ARRAYOF type
223              {$$ = makeTYPEarrayof($2);}
224       | RECORDOF '{' var_decl_list '}'
225              {$$ = makeTYPErecordof($3);}
226 ;
227
228 function : head body tail
229              {$$ = makeFUNCTION($1, $2, $3);}
230 ;
231
232 head : FUNC tIDENTIFIER '(' par_decl_list ')' ':' type
233       {$$ = makeHEAD($2, $4, $7);}
234 ;
235
236 tail : END tIDENTIFIER
237              {$$ = makeTAIL($2);}
238 ;
239 %%
```

A.4 memory.h

```
1  void *Malloc(unsigned n);
2
3  #define NEW(type) (type *)Malloc(sizeof(type))
```

A.5 memory.c

```
1  #include <stdio.h>
2  #include <malloc.h>
3  #include <stdlib.h>
4
5  void *Malloc(unsigned n)
6  {
7    void *p;
8    if(!(p = malloc(n)))
9    {
10     fprintf(stderr,"Malloc(%d) failed.\n",n);
11     fflush(stderr);
```

```
12      abort();
13    }
14    return p;
15  }
```

A.6 tree.h

```
1   #ifndef __tree_h
2   #define __tree_h
3
4   #include "types.h"
5
6
7   typedef struct NestScopeList_ NestScopeList;
8   typedef struct COLLECTION_ COLLECTION;
9
10
11  /*included for saving type collection in body node for
        codegen*/
12
13  /*typedef struct EXP {
14    int lineno;
15    enum {timesK, divK, plusK, minusK, termK} kind;
16    union {
17      struct {struct EXP *left; struct EXP *right;} timesE;
18      struct {struct EXP *left; struct EXP *right;} divE;
19      struct {struct EXP *left; struct EXP *right;} plusE;
20      struct {struct EXP *left; struct EXP *right;} minusE;
21      struct {struct TERM *left;} termE;
22    } val;
23  } EXP; */
24
25  typedef struct EXP {
26    int lineno;
27    enum {binopK, termK} kind;
28    struct {Ty_ty *type; Ty_tyList *args;} types;
29    union {
30      struct {struct EXP *left; char *op; struct EXP *right
          ;} binopE;
31      struct {struct TERM *left;} termE;
32    } val;
33  } EXP;
34
35  typedef struct ACTLIST{
36    int lineno;
37    enum{explistK, nilK} kind;
```

```
38    union{
39      struct{struct EXPLIST *left;} explistA;
40      char *nullA;
41    } val;
42  } ACTLIST;
43
44
45  typedef struct TERM{
46    int lineno;
47    enum{varK, idfuncK, expK, notTermK, lenExpK, numK,
          boolK, nullK} kind;
48    union {
49    int intconstT;
50    int boolconstT;
51    char *null;
52    struct {struct VAR *left;} varT;
53    struct {struct TERM *left;} nTermT;
54    struct {struct EXP *left;} lenExpT;
55    struct {struct EXP *left;} expT;
56    struct {char *id; struct ACTLIST *left;} idfuncT;
57    } val;
58  } TERM;
59
60  typedef struct VAR{
61    int lineno;
62    enum{idVK, listVK, varidVK} kind;
63    union{
64      char *idV;
65      struct {struct VAR *left; struct EXP *right;} listV;
66      struct {struct VAR *left; char *id;} varidV;
67    } val;
68  } VAR;
69
70  typedef struct EXPLIST{
71    int lineno;
72    enum{expLK, explistLK} kind;
73    union{
74      struct {struct EXP *left;} expL;
75      struct {struct EXP *left; struct EXPLIST *right;}
            explistL;
76    } val;
77  } EXPLIST;
78
79  typedef struct FUNCTION{
80    int lineno;
81    union{
```

```
82      struct {struct HEAD *left; struct BODY *middle;
            struct TAIL *right;} function;
83    } val;
84  } FUNCTION;
85
86  typedef struct HEAD {
87    int lineno;
88    union{
89      struct {char *id; struct PAR_DECL_LIST *left; struct
            TYPE *right;} head;
90    } val;
91  } HEAD;
92
93  typedef struct TAIL {
94    int lineno;
95    union{
96      struct {char *id;} tail;
97    }val;
98  } TAIL;
99
100 typedef struct TYPE {
101   int lineno;
102   enum {idTyK, intTyK, boolTyK, arrayTyK, recTyK} kind;
103   union{
104     char *id;
105     int intconst;
106     int boolconst;
107     struct {struct TYPE *left;} array;
108     struct {struct VAR_DECL_LIST *left;} record;
109   } val;
110 } TYPE;
111
112 typedef struct PAR_DECL_LIST{
113   int lineno;
114   enum{vdlPK, nilPK} kind;
115   union{
116     struct {struct VAR_DECL_LIST *left;} vdecl_list;
117     char *nullP;
118   } val;
119 } PAR_DECL_LIST;
120
121 typedef struct VAR_DECL_LIST{
122   int lineno;
123   enum{listVDLK, vtK} kind;
124   union{
125     struct {struct VAR_TYPE *left; struct VAR_DECL_LIST *
```

```
        right;} listV;
126       struct {struct VAR_TYPE *left;} vtype;
127     } val;
128   } VAR_DECL_LIST;
129
130   typedef struct VAR_TYPE{
131     int lineno;
132     union{
133       struct {char *id; struct TYPE *left;} typeV;
134     } val;
135   } VAR_TYPE;
136
137   typedef struct BODY{
138     int lineno;
139     union{
140       struct{struct DECL_LIST *left; struct STATE_LIST *
            right;} body;
141     } val;
142     COLLECTION *c;
143   } BODY;
144
145   typedef struct DECLARATION {
146     int lineno;
147     enum {typeIdK, declFuncK, declVarK} kind;
148     union {
149       struct{char *id; struct TYPE *left;} typeIdD;
150       FUNCTION *declFuncD;
151       VAR_DECL_LIST *declVarD;
152     } val;
153   } DECLARATION;
154
155   typedef struct DECL_LIST {
156     int lineno;
157     enum {declListK, nilDK} kind;
158     union {
159       struct {struct DECLARATION *left; struct DECL_LIST *
            right;} declListDL;
160       char* nilDL;
161
162     } val;
163   } DECL_LIST;
164
165   typedef struct STATE {
166     int lineno;
167     enum {returnK, writeK, allocateK, allocOfLengthK,
          assignK, ifK, ifElseK, whileK, stateListK, incK, decK
```

```
      , forincK, fordecK} kind;
168   union {
169     //struct {struct EXP *left;} returnS;
170     EXP *returnS;
171     //struct {struct EXP *left;} writeS;
172     EXP *writeS;
173     //struct {struct VAR *left;} allocateS;
174     VAR *allocateS;
175
176     VAR *incS;
177
178     VAR *decS;
179
180     struct {struct VAR *left; struct EXP *right;}
          allocOfLengthS;
181     struct {struct VAR *left; struct EXP *right;} assignS
          ;
182     struct {struct EXP *left; struct STATE *right;} ifS;
183     struct {struct EXP *left; struct STATE *middle;
          struct STATE *right;} ifElseS;
184     struct {struct EXP *left; struct STATE *right;}
          whileS;
185     struct {struct STATE_LIST *left;} stateListS;
186     struct {struct TERM *first; struct TERM *second;
          struct TERM *third; struct STATE *fourth;} forincS
          ;
187     struct {struct TERM *first; struct TERM *second;
          struct TERM *third; struct STATE *fourth;} fordecS
          ;
188   } val;
189 } STATE;
190
191 typedef struct STATE_LIST {
192   int lineno;
193   enum {statementK, statementListK} kind;
194   union {
195     STATE *statementSL;
196     struct {struct STATE *left; struct STATE_LIST *right
          ;} statementListSL;
197   } val;
198 } STATE_LIST;
199
200 typedef struct NestScopeList_{
201     FUNCTION *func;
202     NestScopeList *next;
203 } NestScopeList;
```

```
204
205 typedef struct COLLECTION_{
206     SymbolTable *var;
207     SymbolTable *type;
208     SymbolTable *function;
209     NestScopeList *nestedlist;
210     Ty_ty *returns;
211     COLLECTION *next;
212 } COLLECTION;
213
214 /*EXP *makeEXPtimes(EXP *left, EXP *right);
215
216 EXP *makeEXPdiv(EXP *left, EXP *right);
217
218 EXP *makeEXPplus(EXP *left, EXP *right);
219
220 EXP *makeEXPminus(EXP *left, EXP *right);*/
221
222 EXP *makeEXPbinop(EXP *left, char *op, EXP *right);
223
224 EXP *makeEXPterm(TERM *left);
225
226
227
228 TERM *makeTERMnum(int intconst);
229
230 TERM *makeTERMidfunc(char *id, ACTLIST *left);
231
232 TERM *makeTERMvar(VAR *left);
233
234 TERM *makeTERMnot(TERM *left);
235
236 TERM *makeTERMexp(EXP *left);
237
238 TERM *makeTERMlenexp(EXP *left);
239
240 TERM *makeTERMbool(int boolconst);
241
242 TERM *makeTERMnull();
243
244
245
246 VAR *makeVARid(char *id);
247
248 VAR *makeVARlist(VAR *left, EXP *right);
249
```

```
250 VAR *makeVARvarid(VAR *left, char *id);
251
252
253
254 ACTLIST *makeACTLISTlist(EXPLIST *left);
255
256 ACTLIST *makeACTLISTnil();
257
258
259
260 EXPLIST *makeEXPLISTexp(EXP *left);
261
262 EXPLIST *makeEXPLISTlist(EXP *left, EXPLIST *right);
263
264
265 FUNCTION *makeFUNCTION(HEAD *h, BODY *b, TAIL *t);
266
267
268 HEAD *makeHEAD(char *id, PAR_DECL_LIST *left, TYPE *right
        );
269
270
271 TAIL *makeTAIL(char *id);
272
273
274 TYPE *makeTYPEid(char *id);
275
276 TYPE *makeTYPEint();
277
278 TYPE *makeTYPEbool();
279
280 TYPE *makeTYPEarrayof(TYPE *left);
281
282 TYPE *makeTYPErecordof(VAR_DECL_LIST *left);
283
284
285 PAR_DECL_LIST *makePAR_DECL_LISTvdl(VAR_DECL_LIST *left);
286
287 PAR_DECL_LIST *makePAR_DECL_LISTnil();
288
289
290 VAR_DECL_LIST *makeVAR_DECL_LISTlist(VAR_TYPE *left,
        VAR_DECL_LIST *right);
291
292 VAR_DECL_LIST *makeVAR_DECL_LISTvt(VAR_TYPE *left);
293
```

```
294
295 VAR_TYPE *makeVAR_TYPEid(char *id, TYPE *left);
296
297
298 BODY *makeBODY(DECL_LIST *left, STATE_LIST *right);
299
300
301 DECLARATION *makeDECLARATIONtypeId(char *id, TYPE *left);
302
303 DECLARATION *makeDECLARATIONdeclFuncD(FUNCTION *left);
304
305 DECLARATION *makeDECLARATIONdeclVarD(VAR_DECL_LIST *left)
        ;
306
307
308 DECL_LIST *makeDECL_LISTdeclList(DECLARATION *left,
        DECL_LIST *right);
309
310 DECL_LIST *makeDECL_LISTnil();
311
312
313 STATE *makeSTATEreturn(EXP *left);
314
315 STATE *makeSTATEwrite(EXP *left);
316
317 STATE *makeSTATEallocate(VAR *left);
318
319 STATE *makeSTATEallocOfLength(VAR *left, EXP *right);
320
321 STATE *makeSTATEassign(VAR *left, EXP *right);
322
323 STATE *makeSTATEif(EXP *left, STATE *right);
324
325 STATE *makeSTATEifElse(EXP *left, STATE *middle, STATE *
        right);
326
327 STATE *makeSTATEwhile(EXP *left, STATE *right);
328
329 STATE *makeSTATEstateList(STATE_LIST *left);
330
331 STATE *makeSTATEinc(VAR *left);
332
333 STATE *makeSTATEdec(VAR *left);
334
335 STATE *makeSTATEforinc(DECL_LIST *first, TERM *second,
        TERM *third, STATE *fourth);
```

```
336
337 STATE *makeSTATEfordec(DECL_LIST *first, TERM *second,
        TERM *third, STATE *fourth);
338
339
340 STATE_LIST *makeSTATE_LISTstatement(STATE *left);
341
342 STATE_LIST *makeSTATE_LISTstatementList(STATE *left,
        STATE_LIST *right);
343 #endif
```

A.7 tree.c

```
 1 #include "memory.h"
 2 #include "tree.h"
 3 extern int lineno;
 4 /*
 5 EXP *makeEXPid(char *id)
 6 { EXP *e;
 7   e = NEW(EXP);
 8   e->lineno = lineno;
 9   e->kind = idK;
10   e->val.idE = id;
11   return e;
12 }
13
14 EXP *makeEXPintconst(int intconst)
15 { EXP *e;
16   e = NEW(EXP);
17   e->lineno = lineno;
18   e->kind = intconstK;
19   e->val.intconstE = intconst;
20   return e;
21 }
22
23 EXP *makeEXPtimes(EXP *left, EXP *right)
24 { EXP *e;
25   e = NEW(EXP);
26   e->lineno = lineno;
27   e->kind = timesK;
28   e->val.timesE.left = left;
29   e->val.timesE.right = right;
30   return e;
31 }
32
33 EXP *makeEXPdiv(EXP *left, EXP *right)
```

```
34 { EXP *e;
35   e = NEW(EXP);
36   e->lineno = lineno;
37   e->kind = divK;
38   e->val.divE.left = left;
39   e->val.divE.right = right;
40   return e;
41 }
42
43 EXP *makeEXPplus(EXP *left, EXP *right)
44 { EXP *e;
45   e = NEW(EXP);
46   e->lineno = lineno;
47   e->kind = plusK;
48   e->val.plusE.left = left;
49   e->val.plusE.right = right;
50   return e;
51 }
52
53 EXP *makeEXPminus(EXP *left, EXP *right)
54 { EXP *e;
55   e = NEW(EXP);
56   e->lineno = lineno;
57   e->kind = minusK;
58   e->val.minusE.left = left;
59   e->val.minusE.right = right;
60   return e;
61 }*/
62
63 EXP *makeEXPbinop(EXP *left, char *op, EXP *right){
64     EXP *e;
65     e=NEW(EXP);
66     e->kind=binopK;
67     e->lineno=lineno;
68     e->val.binopE.left=left;
69     e->val.binopE.op=op;
70     e->val.binopE.right=right;
71     return e;
72 }
73
74 EXP *makeEXPterm(TERM *t){
75     EXP *e;
76     e = NEW(EXP);
77     e->lineno=lineno;
78     e->kind=termK;
79     e->val.termE.left=t;
```

```
80      return e;
81  }
82
83  //e.g. 9
84  TERM *makeTERMnum(int intconst){
85      TERM *t;
86      t = NEW(TERM);
87      t->lineno = lineno;
88      t->kind=numK;
89      t->val.intconstT = intconst;
90      return t;
91  }
92
93  //"id(ACTLIST)"
94  TERM *makeTERMidfunc(char *id, ACTLIST *left){
95      TERM *t;
96      t = NEW(TERM);
97      t->lineno = lineno;
98      t->kind=idfuncK;
99      t->val.idfuncT.id=id;
100     t->val.idfuncT.left = left;
101     return t;
102 }
103
104 //VARIABLE
105 TERM *makeTERMvar(VAR *left){
106     TERM *t;
107     t = NEW(TERM);
108     t->lineno = lineno;
109     t->kind = varK;
110     t->val.varT.left=left;
111     return t;
112 }
113
114 //"!TERM"
115 TERM *makeTERMnot(TERM *left){
116     TERM *t;
117     t = NEW(TERM);
118     t->lineno = lineno;
119     t->kind = notTermK;
120     t->val.nTermT.left=left;
121     return t;
122 }
123
124 //"(EXPRESSION)"
125 TERM *makeTERMexp(EXP *left){
```

```
126       TERM *t;
127       t = NEW(TERM);
128       t->lineno = lineno;
129       t->kind = expK;
130       t->val.expT.left=left;
131       return t;
132   }
133
134   //"|expression|"
135   TERM *makeTERMlenexp(EXP *left){
136       TERM *t;
137       t = NEW(TERM);
138       t->lineno = lineno;
139       t->kind = lenExpK;
140       t->val.lenExpT.left=left;
141       return t;
142   }
143
144   //"true" or "false"
145   TERM *makeTERMbool(int boolconst){
146       TERM *t;
147       t = NEW(TERM);
148       t->lineno = lineno;
149       t->kind = boolK;
150       t->val.boolconstT=boolconst;
151       return t;
152   }
153
154   //"null"
155   TERM *makeTERMnull(){
156       TERM *t;
157       t = NEW(TERM);
158       t->lineno = lineno;
159       t->kind = nullK;
160       t->val.null=0;
161       return t;
162   }
163
164   VAR *makeVARid(char *id){
165       VAR *v;
166       v=NEW(VAR);
167       v->lineno=lineno;
168       v->kind=idVK;
169       v->val.idV=id;
170       return v;
171   }
```

```
172
173 VAR *makeVARlist(VAR *left, EXP *right){
174     VAR *v;
175     v=NEW(VAR);
176     v->lineno=lineno;
177     v->kind=listVK;
178     v->val.listV.left=left;
179     v->val.listV.right=right;
180     return v;
181 }
182
183 VAR *makeVARvarid(VAR *left, char *id){
184     VAR *v;
185     v=NEW(VAR);
186     v->lineno=lineno;
187     v->kind=varidVK;
188     v->val.varidV.left=left;
189     v->val.varidV.id=id;
190     return v;
191 }
192
193 ACTLIST *makeACTLISTlist(EXPLIST *left){
194     ACTLIST *a;
195     a=NEW(ACTLIST);
196     a->lineno=lineno;
197     a->kind=explistK;
198     a->val.explistA.left=left;
199     return a;
200 }
201
202 ACTLIST *makeACTLISTnil(){
203     ACTLIST *a;
204     a=NEW(ACTLIST);
205     a->lineno=lineno;
206     a->kind=nilK;
207     a->val.nullA = 0;
208     return a;
209 }
210
211 EXPLIST *makeEXPLISTexp(EXP *left){
212     EXPLIST *e;
213     e=NEW(EXPLIST);
214     e->lineno=lineno;
215     e->kind=expLK;
216     e->val.expL.left=left;
217     return e;
```

```
218 }
219
220 EXPLIST *makeEXPLISTlist(EXP *left, EXPLIST *right){
221     EXPLIST *e;
222     e=NEW(EXPLIST);
223     e->lineno=lineno;
224     e->kind=explistLK;
225     e->val.explistL.left=left;
226     e->val.explistL.right=right;
227     return e;
228 }
229
230 FUNCTION *makeFUNCTION(HEAD *h, BODY *b, TAIL*t){
231     FUNCTION *f;
232     f=NEW(FUNCTION);
233     f->lineno=lineno;
234     f->val.function.left=h;
235     f->val.function.middle=b;
236     f->val.function.right=t;
237     return f;
238 }
239
240 HEAD *makeHEAD(char *id, PAR_DECL_LIST *left, TYPE *right
        ){
241     HEAD *h;
242     h=NEW(HEAD);
243     h->lineno=lineno;
244     h->val.head.id=id;
245     h->val.head.left=left;
246     h->val.head.right=right;
247     return h;
248 }
249
250 TAIL *makeTAIL(char *id){
251     TAIL *t;
252     t=NEW(TAIL);
253     t->lineno=lineno;
254     t->val.tail.id=id;
255     return t;
256 }
257
258 TYPE *makeTYPEid(char *id){
259     TYPE *t;
260     t=NEW(TYPE);
261     t->lineno=lineno;
262     t->kind=idTyK;
```

```
263         t->val.id=id;
264         return t;
265     }
266
267     TYPE *makeTYPEint(){
268         TYPE *t;
269         t=NEW(TYPE);
270         t->lineno=lineno;
271         t->kind=intTyK;
272         t->val.intconst=0;
273         return t;
274     }
275
276     TYPE *makeTYPEbool(){
277         TYPE *t;
278         t=NEW(TYPE);
279         t->lineno=lineno;
280         t->kind=boolTyK;
281         t->val.boolconst=0;
282         return t;
283     }
284
285     TYPE *makeTYPEarrayof(TYPE *left){
286         TYPE *t;
287         t=NEW(TYPE);
288         t->lineno=lineno;
289         t->kind=arrayTyK;
290         t->val.array.left=left;
291         return t;
292     }
293
294     TYPE *makeTYPErecordof(VAR_DECL_LIST *left){
295         TYPE *t;
296         t=NEW(TYPE);
297         t->lineno=lineno;
298         t->kind=recTyK;
299         t->val.record.left=left;
300         return t;
301     }
302
303     PAR_DECL_LIST *makePAR_DECL_LISTnil(){
304         PAR_DECL_LIST *p;
305         p=NEW(PAR_DECL_LIST);
306         p->lineno=lineno;
307         p->kind=nilK;
308         p->val.nullP=0;
```

```
309     return p;
310 }
311
312 PAR_DECL_LIST *makePAR_DECL_LISTvdl(VAR_DECL_LIST *left){
313     PAR_DECL_LIST *p;
314     p=NEW(PAR_DECL_LIST);
315     p->lineno=lineno;
316     p->kind=vdlPK;
317     p->val.vdecl_list.left=left;
318     return p;
319 }
320
321 VAR_DECL_LIST *makeVAR_DECL_LISTlist(VAR_TYPE *left,
        VAR_DECL_LIST *right){
322     VAR_DECL_LIST *v;
323     v=NEW(VAR_DECL_LIST);
324     v->lineno=lineno;
325     v->kind=listVDLK;
326     v->val.listV.left=left;
327     v->val.listV.right=right;
328     return v;
329 }
330
331 VAR_DECL_LIST *makeVAR_DECL_LISTvt(VAR_TYPE *left){
332     VAR_DECL_LIST *v;
333     v=NEW(VAR_DECL_LIST);
334     v->lineno=lineno;
335     v->kind=vtK;
336     v->val.vtype.left=left;
337     return v;
338 }
339
340 VAR_TYPE *makeVAR_TYPEid(char *id, TYPE *left){
341     VAR_TYPE *v;
342     v=NEW(VAR_TYPE);
343     v->lineno=lineno;
344     v->val.typeV.id=id;
345     v->val.typeV.left=left;
346     return v;
347 }
348
349 BODY *makeBODY(DECL_LIST *left, STATE_LIST *right){
350     BODY *b;
351     b=NEW(BODY);
352     b->lineno=lineno;
353     b->val.body.left=left;
```

```
354     b->val.body.right=right;
355     return b;
356 }
357
358 DECLARATION *makeDECLARATIONtypeId(char *id, TYPE *left){
359     DECLARATION *d;
360     d = NEW(DECLARATION);
361     d->lineno = lineno;
362     d->kind = typeIdK;
363     d->val.typeIdD.left = left;
364     d->val.typeIdD.id=id;
365     return d;
366 }
367
368 DECLARATION *makeDECLARATIONdeclFuncD(FUNCTION *left){
369     DECLARATION *d;
370     d = NEW(DECLARATION);
371     d->lineno = lineno;
372     d->kind = declFuncK;
373     d->val.declFuncD = left;
374     return d;
375 }
376
377 DECLARATION *makeDECLARATIONdeclVarD(VAR_DECL_LIST *left)
    {
378     DECLARATION *d;
379     d = NEW(DECLARATION);
380     d->lineno = lineno;
381     d->kind = declVarK;
382     d->val.declVarD = left;
383     return d;
384 }
385
386 DECL_LIST *makeDECL_LISTdeclList(DECLARATION *left,
    DECL_LIST *right){
387     DECL_LIST *dl;
388     dl = NEW(DECL_LIST);
389     dl->lineno = lineno;
390     dl->kind = declListK;
391     dl->val.declListDL.left = left;
392     dl->val.declListDL.right = right;
393     return dl;
394 }
395
396 DECL_LIST *makeDECL_LISTnil(){
397     DECL_LIST *dl;
```

```
398       dl = NEW(DECL_LIST);
399       dl->lineno = lineno;
400       dl->kind = nilDK;
401       dl->val.declListDL.left = 0;
402       return dl;
403   }
404
405   STATE *makeSTATEreturn(EXP *left){
406       STATE *s;
407       s = NEW(STATE);
408       s->lineno = lineno;
409       s->kind = returnK;
410       s->val.returnS = left;
411       return s;
412   }
413   STATE *makeSTATEwrite(EXP *left){
414       STATE *s;
415       s = NEW(STATE);
416       s->lineno = lineno;
417       s->kind = writeK;
418       s->val.writeS = left;
419       return s;
420   }
421   STATE *makeSTATEallocate(VAR *left){
422       STATE *s;
423       s = NEW(STATE);
424       s->lineno = lineno;
425       s->kind = allocateK;
426       s->val.allocateS = left;
427       return s;
428   }
429
430   STATE *makeSTATEallocOfLength(VAR *left, EXP *right){
431       STATE *s;
432       s = NEW(STATE);
433       s->lineno = lineno;
434       s->kind = allocOfLengthK;
435       s->val.allocOfLengthS.left = left;
436       s->val.allocOfLengthS.right = right;
437       return s;
438   }
439   STATE *makeSTATEassign(VAR *left, EXP *right){
440       STATE *s;
441       s = NEW(STATE);
442       s->lineno = lineno;
443       s->kind = assignK;
```

```
444     s->val.assignS.left = left;
445     s->val.assignS.right = right;
446     return s;
447 }
448
449 STATE *makeSTATEif(EXP *left, STATE *right){
450     STATE *s;
451     s = NEW(STATE);
452     s->lineno = lineno;
453     s->kind = ifK;
454     s->val.ifS.left = left;
455     s->val.ifS.right = right;
456     return s;
457 }
458 STATE *makeSTATEifElse(EXP *left, STATE *middle, STATE *
        right){
459     STATE *s;
460     s = NEW(STATE);
461     s->lineno = lineno;
462     s->kind = ifElseK;
463     s->val.ifElseS.left = left;
464     s->val.ifElseS.middle = middle;
465     s->val.ifElseS.right = right;
466     return s;
467 }
468
469 STATE *makeSTATEwhile(EXP *left, STATE *right){
470     STATE *s;
471     s = NEW(STATE);
472     s->lineno = lineno;
473     s->kind = whileK;
474     s->val.whileS.left = left;
475     s->val.whileS.right = right;
476     return s;
477 }
478
479 STATE *makeSTATEstateList(STATE_LIST *left){
480     STATE *s;
481     s = NEW(STATE);
482     s->lineno = lineno;
483     s->kind = stateListK;
484     s->val.stateListS.left = left;
485     return s;
486 }
487
488 STATE *makeSTATEinc(VAR *left){
```

```
489     STATE *s;
490     s = NEW(STATE);
491     s->lineno = lineno;
492     s->kind = incK;
493     s->val.incS = left;
494     return s;
495 }
496
497 STATE *makeSTATEdec(VAR *left){
498     STATE *s;
499     s = NEW(STATE);
500     s->lineno = lineno;
501     s->kind = decK;
502     s->val.decS = left;
503     return s;
504 }
505
506 STATE *makeSTATEforinc(DECL_LIST *first, TERM *second,
        TERM *third, STATE *fourth){
507     STATE *s;
508     s = NEW(STATE);
509     s->lineno = lineno;
510     s->kind = forincK;
511     s->val.forincS.first = first;
512     s->val.forincS.second = second;
513     s->val.forincS.third = third;
514     s->val.forincS.fourth = fourth;
515     return s;
516 }
517
518 STATE *makeSTATEfordec(DECL_LIST *first, TERM *second,
        TERM *third, STATE *fourth){
519     STATE *s;
520     s = NEW(STATE);
521     s->lineno = lineno;
522     s->kind = fordecK;
523     s->val.fordecS.first = first;
524     s->val.fordecS.second = second;
525     s->val.fordecS.third = third;
526     s->val.fordecS.fourth = fourth;
527     return s;
528 }
529
530 STATE_LIST *makeSTATE_LISTstatement(STATE *left){
531     STATE_LIST *s;
532     s = NEW(STATE_LIST);
```

```
533     s->lineno = lineno;
534     s->kind = statementK;
535     s->val.statementSL = left;
536     return s;
537 }
538
539 STATE_LIST *makeSTATE_LISTstatementList(STATE *left,
        STATE_LIST *right){
540     STATE_LIST *s;
541     s = NEW(STATE_LIST);
542     s->lineno = lineno;
543     s->kind = statementListK;
544     s->val.statementListSL.left = left;
545     s->val.statementListSL.right = right;
546     return s;
547 }
```

A.8 pretty.h

```
1  #include "tree.h"
2
3  void prettyEXP(EXP *e);
4  void prettyTERM(TERM *t);
5  void prettyVAR(VAR *v);
6  void prettyACTLIST(ACTLIST *act);
7  void prettyEXPLIST(EXPLIST *expl);
8  void prettySTATE(STATE *s);
9  void prettySTATE_LIST(STATE_LIST *sl);
10 void prettyDECLARATION(DECLARATION *d);
11 void prettyDECL_LIST(DECL_LIST *dl);
12 void prettyFunction(FUNCTION *f);
13 void prettyHEAD(HEAD *h);
14 void prettyTAIL(TAIL *t);
15 void prettyTYPE(TYPE *t);
16 void prettyPAR_DECL_LIST(PAR_DECL_LIST *p);
17 void prettyVAR_DECL_LIST(VAR_DECL_LIST *vdl);
18 void prettyVAR_TYPE(VAR_TYPE *v);
19 void prettyBODY(BODY *b);
```

A.9 pretty.c

```
1  #include <stdio.h>
2  #include "pretty.h"
3  int indendation= 0;
4  int spaces = 2;
5  #ifndef debug
```

```
 6 #define debug
 7 int db = 1;
 8 #endif
 9
10 void prettyEXP(EXP *e)
11 { switch (e->kind) {
12       case binopK:
13             prettyEXP(e->val.binopE.left);
14             fprintf(stderr, " %s ", e->val.binopE.op);
15             prettyEXP(e->val.binopE.right);
16             break;
17       case termK:
18             prettyTERM(e->val.termE.left);
19             break;
20   }
21 }
22
23 void prettyTERM(TERM *t){
24   switch(t->kind){
25     case varK:
26             prettyVAR(t->val.varT.left);
27             break;
28     case idfuncK:
29             fprintf(stderr, "%*s", spaces*indendation,"");
30             fprintf(stderr, "%s (", t->val.idfuncT.id);
31             prettyACTLIST(t->val.idfuncT.left);
32             fprintf(stderr, ")");
33             break;
34     case expK:
35             fprintf(stderr, "(");
36             prettyEXP(t->val.expT.left);
37             fprintf(stderr, ")");
38             break;
39     case notTermK:
40             fprintf(stderr, "!");
41             prettyTERM(t->val.nTermT.left);
42             break;
43     case lenExpK:
44             fprintf(stderr, "|");
45             prettyEXP(t->val.lenExpT.left);
46             fprintf(stderr, "|");
47             break;
48     case numK:
49             fprintf(stderr, "%d", t->val.intconstT);
50             break;
51     case boolK:
```

```
52              if(t->val.boolconstT == 0){
53                      fprintf(stderr, "false");
54              } else if (t->val.boolconstT == 1)
55              {
56                      fprintf(stderr, "true");
57              }
58              break;
59      case nullK:
60              fprintf(stderr, "null");
61              break;
62    }
63  }
64
65  void prettyVAR(VAR *v){
66      switch(v->kind){
67              case idVK:
68                      fprintf(stderr, "%s", v->val.idV);
69                      break;
70              case listVK:
71                      prettyVAR(v->val.listV.left);
72                      fprintf(stderr, "[");
73                      prettyEXP(v->val.listV.right);
74                      fprintf(stderr, "]");
75                      break;
76              case varidVK:
77                      prettyVAR(v->val.varidV.left);
78                      fprintf(stderr, ".%s", v->val.varidV.id);
79                      break;
80      }
81  }
82
83  void prettyACTLIST(ACTLIST *act){
84      switch(act->kind){
85              case explistK:
86                      prettyEXPLIST(act->val.explistA.left);
87                      break;
88              case nilK:
89                      break;
90      }
91  }
92
93  void prettyEXPLIST(EXPLIST *expl){
94      switch(expl->kind){
95              case expLK:
96                      prettyEXP(expl->val.expL.left);
97                      break;
```

```
 98              case explistLK:
 99                   prettyEXP(expl->val.explistL.left);
100                   fprintf(stderr, ", ");
101                   prettyEXPLIST(expl->val.explistL.right);
102                   break;
103         }
104    }
105
106    void prettyFunction(FUNCTION *f){
107         prettyHEAD(f->val.function.left);
108         prettyBODY(f->val.function.middle);
109         prettyTAIL(f->val.function.right);
110    }
111
112    void prettyHEAD(HEAD *h){
113         fprintf(stderr, "func %s(", h->val.head.id);
114         indendation++;
115         prettyPAR_DECL_LIST(h->val.head.left);
116         fprintf(stderr, ") : ");
117         prettyTYPE(h->val.head.right);
118         indendation--;
119         fprintf(stderr, "\n");
120    }
121
122    void prettyTAIL(TAIL *t){
123         fprintf(stderr, "end %s\n", t->val.tail.id);
124    }
125
126    void prettyTYPE(TYPE *t){
127         switch(t->kind){
128              case idTyK:
129                   fprintf(stderr, "%*s", spaces*indendation,
                          "");
130                   fprintf(stderr, "%s", t->val.id);
131                   break;
132              case intTyK:
133                   fprintf(stderr, "int");
134                   break;
135              case boolTyK:
136                   fprintf(stderr, "bool");
137                   break;
138              case arrayTyK:
139                   fprintf(stderr, "array of ");
140                   prettyTYPE(t->val.array.left);
141                   break;
142              case recTyK:
```

```
143                 fprintf(stderr, "record of {\n");
144                 prettyVAR_DECL_LIST(t->val.record.left);
145                 fprintf(stderr, " }\n");
146                 break;
147         }
148 }
149
150 void prettyPAR_DECL_LIST(PAR_DECL_LIST *p){
151         switch(p->kind){
152                 case nilPK:
153                         break;
154                 case vdlPK:
155                         prettyVAR_DECL_LIST(p->val.vdecl_list.left
                                );
156                         break;
157         }
158 }
159
160 void prettyVAR_DECL_LIST(VAR_DECL_LIST *vdl){
161         switch(vdl->kind){
162                 case listVDLK:
163                         prettyVAR_TYPE(vdl->val.listV.left);
164                         fprintf(stderr, ", ");
165                         prettyVAR_DECL_LIST(vdl->val.listV.right);
166                         break;
167                 case vtK:
168                         prettyVAR_TYPE(vdl->val.vtype.left);
169                         break;
170         }
171 }
172
173 void prettyVAR_TYPE(VAR_TYPE *v){
174         fprintf(stderr, "%s : ", v->val.typeV.id);
175         prettyTYPE(v->val.typeV.left);
176 }
177
178 void prettyBODY(BODY *b){
179         prettyDECL_LIST(b->val.body.left);
180         prettySTATE_LIST(b->val.body.right);
181 }
182
183 void prettyDECLARATION(DECLARATION *d){
184         switch(d->kind){
185                 case typeIdK:
186                         fprintf(stderr, "%*s", (spaces*indendation
                                ),"");
```

```
187                 fprintf(stderr, "type %s = ", d->val.
                        typeIdD.id);
188                 prettyTYPE(d->val.typeIdD.left);
189                 // fprintf(";\n");
190                 break;
191           case declFuncK:
192                 prettyFunction(d->val.declFuncD);
193                 break;
194           case declVarK:
195                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
196                 fprintf(stderr, "var ");
197                 prettyVAR_DECL_LIST(d->val.declVarD);
198                 fprintf(stderr, ";\n");
199                 break;
200     }
201 }
202
203 void prettyDECL_LIST(DECL_LIST *dl){
204     switch(dl->kind){
205           case declListK:
206                 prettyDECLARATION(dl->val.declListDL.left)
                        ;
207                 prettyDECL_LIST(dl->val.declListDL.right);
208                 break;
209           case nilDK:
210                 break;
211     }
212 }
213
214 void prettySTATE_LIST(STATE_LIST *sl){
215   switch(sl->kind){
216     case statementK:
217           prettySTATE(sl->val.statementSL);
218           break;
219     case statementListK:
220           prettySTATE(sl->val.statementListSL.left);
221           prettySTATE_LIST(sl->val.statementListSL.right)
                  ;
222           break;
223   }
224
225 }
226
227 void prettySTATE(STATE *s){
228     switch(s->kind){
```

```
229             case returnK:
230                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
231                 fprintf(stderr, "return ");
232                 prettyEXP(s->val.returnS);
233                 fprintf(stderr, ";\n");
234                 break;
235             case writeK:
236                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
237                 fprintf(stderr, "write ");
238                 prettyEXP(s->val.writeS);
239                 fprintf(stderr, ";\n");
240                 break;
241             case allocateK:
242                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
243                 fprintf(stderr, "allocate ");
244                 prettyVAR(s->val.allocateS);
245                 fprintf(stderr, ";\n");
246                 break;
247             case allocOfLengthK:
248                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
249                 fprintf(stderr, "allocate ");
250                 prettyVAR(s->val.allocOfLengthS.left);
251                 fprintf(stderr, " of length ");
252                 prettyEXP(s->val.allocOfLengthS.right);
253                 fprintf(stderr, ";\n");
254                 break;
255             case assignK:
256                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
257                 prettyVAR(s->val.assignS.left);
258                 fprintf(stderr, " = ");
259                 prettyEXP(s->val.assignS.right);
260                 fprintf(stderr, ";\n");
261                 break;
262             case ifK:
263                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
264                 fprintf(stderr, "if ");
265                 prettyEXP(s->val.ifS.left);
266                 indendation++;
267                 fprintf(stderr, "\n");
268                 fprintf(stderr, "%*s", spaces*indendation,
```

```
                ");
269             fprintf(stderr, "then\n");
270             prettySTATE(s->val.ifS.right);
271             indendation--;
272             break;
273         case ifElseK:
274             fprintf(stderr, "%*s", spaces*indendation,
                ");
275             fprintf(stderr, "if ");
276             prettyEXP(s->val.ifElseS.left);
277             indendation++;
278             fprintf(stderr, "\n");
279             fprintf(stderr, "%*s", spaces*indendation,
                ");
280             fprintf(stderr, "then\n");
281             prettySTATE(s->val.ifElseS.middle);
282             fprintf(stderr, "\n");
283             fprintf(stderr, "%*s", spaces*indendation,
                ");
284             fprintf(stderr, "else\n");
285             prettySTATE(s->val.ifElseS.right);
286             indendation--;
287             break;
288         case whileK:
289             fprintf(stderr, "%*s", spaces*indendation,
                ");
290             fprintf(stderr, "while ");
291             prettyEXP(s->val.whileS.left);
292             indendation++;
293             fprintf(stderr, "\n");
294             fprintf(stderr, "%*s", spaces*indendation,
                ");
295             fprintf(stderr, "do\n");
296             prettySTATE(s->val.whileS.right);
297             indendation--;
298             break;
299         case stateListK:
300             indendation++;
301             fprintf(stderr, "%*s", spaces*indendation,
                ");
302             fprintf(stderr, "{\n");
303             prettySTATE_LIST(s->val.stateListS.left);
304             fprintf(stderr, "%*s", spaces*indendation,
                ");
305             fprintf(stderr, "}\n");
306             indendation--;
```

```
307                 break;
308             case incK:
309                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
310                 prettyVAR(s->val.incS);
311                 fprintf(stderr, "++;\n");
312                 break;
313             case decK:
314                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
315                 prettyVAR(s->val.decS);
316                 fprintf(stderr, "--;\n");
317                 break;
318             case forincK:
319                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
320                 fprintf(stderr, "for ");
321                 prettyVAR_TYPE(s->val.forincS.first);
322                 fprintf(stderr, " ++ [");
323                 prettyTERM(s->val.forincS.second);
324                 fprintf(stderr, ",");
325                 prettyTERM(s->val.forincS.third);
326                 fprintf(stderr, "]\n");
327                 indendation++;
328                 prettySTATE(s->val.forincS.fourth);
329                 indendation--;
330                 break;
331             case fordecK:
332                 fprintf(stderr, "%*s", spaces*indendation,
                        "");
333                 fprintf(stderr, "for ");
334                 prettyVAR_TYPE(s->val.fordecS.first);
335                 fprintf(stderr, " -- [");
336                 prettyTERM(s->val.fordecS.second);
337                 fprintf(stderr, ",");
338                 prettyTERM(s->val.fordecS.third);
339                 fprintf(stderr, "]\n");
340                 indendation++;
341                 prettySTATE(s->val.fordecS.fourth);
342                 indendation--;
343                 break;
344         }
345 }
```

A.10 symbol.h

```
1  #ifndef __symbol_h
2  #define __symbol_h
3
4  #define HashSize 317
5
6  /* SYMBOL will be extended later.
7     Function calls will take more parameters later.
8  */
9
10 typedef struct SYMBOL {
11   char *name;
12   void *value;
13   struct SYMBOL *next;
14 } SYMBOL;
15
16 typedef struct SymbolTable {
17     SYMBOL *table[HashSize];
18     struct SymbolTable *next;
19 } SymbolTable;
20
21 typedef struct SymbolList {
22   SYMBOL *head;
23   struct SymbolList *tail;
24 } SymbolList;
25
26 int Hash(char *str);
27
28 SymbolTable *initSymbolTable();
29
30 SymbolTable *scopeSymbolTable(SymbolTable *t);
31
32 SYMBOL *putSymbol(SymbolTable *t, char *name, void *value
       );
33
34 SYMBOL *getSymbol(SymbolTable *t, char *name);
35
36 SymbolList *createSymbolList(SYMBOL *head, SymbolList *
       tail);
37
38 SymbolList *AllSymbolsInScope(SymbolTable *t);
39
40 void dumpSymbolTable(SymbolTable *t);
41
42 #endif
```

A.11 symbol.c

```
1  #include "symbol.h"
2  #include "memory.h"
3  #include <stdio.h>
4  #include <string.h>
5
6  /*
7  OBS
8  passed names assumed to not be changed
9  duplicate names in put get dropped and returns original
       symbol
10 names assumed to be strings terminating on "\0"
11 */
12
13
14 int Hash(char *str){
15     int i = 0;
16     int hash = 0;
17     while((char) str[i] != '\0'){
18         hash = hash + (int) str[i];
19         hash = hash*2;
20         i++;
21     }
22     return hash % HashSize;
23 }
24
25 SymbolTable *initSymbolTable(){
26     SymbolTable *table = NEW(SymbolTable);
27     return table;
28 }
29
30 SymbolTable *scopeSymbolTable(SymbolTable *t){
31     SymbolTable *table = NEW(SymbolTable);
32     table->next = t;
33     return table;
34 }
35
36 /*
37 beware of contents of pointer to name changing
38 accounting for duplicate name, currently done by dropping
       the new symbol and returning the already existing
       symbol
39 name assumed to be string ending in "\0"
40 */
41 SYMBOL *putSymbol(SymbolTable *t, char *name, void *value
```

```
    ){
42      SYMBOL *s;
43      int i = Hash(name);
44      if(t->table[i] == NULL){
45          t->table[i] = NEW(SYMBOL);
46          t->table[i]->name = name;
47          t->table[i]->value = value;
48          s = t->table[i];
49      } else {
50          SYMBOL *next = t->table[i];
51
52          if(strcmp(name, next->name) == 0){
53              return next;
54          }
55
56          while(next->next != NULL){
57              next = next->next;
58              if(strcmp(name, next->name) == 0){
59                  return next;
60              }
61          }
62
63          next->next = NEW(SYMBOL);
64          next->next->name = name;
65          next->next->value = value;
66          s = next->next;
67      }
68      return s;
69  }
70
71
72  //checks list of symbols with same hash value, returning
        pointer of symbol if match found.
73  //If not found, recursively calls next symboltable until
        match or root reached.
74  SYMBOL *getSymbol(SymbolTable *t, char *name){
75      int i = Hash(name);
76      SYMBOL *s = t->table[i];
77      while(s != NULL){
78          if(strcmp(name, s->name) == 0){
79              return s;
80          }
81          s = s->next;
82      }
83
84
```

```
85      if(t->next != NULL){
86          s = getSymbol(t->next, name);
87          return s;
88      } else {
89          return NULL;
90      }
91  }
92
93  void dumpSymbolTable(SymbolTable *t){
94      fprintf(stderr,"New table \n");
95      for(int i = 0; i<HashSize; i++){
96          SYMBOL *s = t->table[i];
97          while(s != NULL){
98              fprintf(stderr,"Name: %s", s->name);
99              //fprintf(stderr,"Value is: %d \n", s->value)
                    ;
100             s = s->next;
101         }
102     }
103     if(t->next != NULL){
104         dumpSymbolTable(t->next);
105     }
106 }
107
108 SymbolList *createSymbolList(SYMBOL *head, SymbolList *
        tail){
109     SymbolList *sl = NEW(SymbolList);
110     sl->head = head;
111     sl->tail = tail;
112     return sl;
113 }
114
115 SymbolList *AllSymbolsInScope(SymbolTable *t){
116     SymbolList *sl = NULL;
117     for(int i = 0; i<HashSize; i++){
118         SYMBOL *s = t->table[i];
119         while(s != NULL){
120             sl = createSymbolList(s, sl);
121             s = s->next;
122         }
123     }
124     return sl;
125 }
```

A.12 weed.h

```
1  #include "tree.h"
2  #include <stdbool.h>
3
4
5  void weedEXP(EXP *e);
6  void weedTERM(TERM *t);
7  void weedVAR(VAR *v);
8  void weedACTLIST(ACTLIST *act);
9  void weedEXPLIST(EXPLIST *expl);
10 void weedSTATE(STATE *s);
11 void weedSTATE_LIST(STATE_LIST *sl);
12 void weedDECLARATION(DECLARATION *d);
13 void weedDECL_LIST(DECL_LIST *dl);
14 void weedFunction(FUNCTION *f);
15 void weedHEAD(HEAD *h);
16 void weedTAIL(TAIL *t);
17 void weedTYPE(TYPE *t);
18 void weedPAR_DECL_LIST(PAR_DECL_LIST *p);
19 void weedVAR_DECL_LIST(VAR_DECL_LIST *vdl);
20 void weedVAR_TYPE(VAR_TYPE *v);
21 void weedBODY(BODY *b);
22
23 // Next three are for looking for return statements.
24 bool statementListChecker(STATE_LIST *sl);
25 bool ifChecker(STATE *s);
26 bool thenElseChecker(STATE *s);
```

A.13 weed.c

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include "weed.h"
6
7  // Color palette
8  #define RED   "\x1B[31m"
9  #define GRN   "\x1B[32m"
10 #define YEL   "\x1B[33m"
11 #define BLU   "\x1B[34m"
12 #define MAG   "\x1B[35m"
13 #define CYN   "\x1B[36m"
14 #define WHT   "\x1B[37m"
15 #define RESET "\x1B[0m"
16
17 void weedEXP(EXP *e)
```

```
18 { switch (e->kind) {
19       case binopK:
20             weedEXP(e->val.binopE.left);
21             weedEXP(e->val.binopE.right);
22             break;
23       case termK:
24             weedTERM(e->val.termE.left);
25             break;
26   }
27 }
28
29 void weedTERM(TERM *t){
30   switch(t->kind){
31     case varK:
32             weedVAR(t->val.varT.left);
33             break;
34     case idfuncK:
35             weedACTLIST(t->val.idfuncT.left);
36             break;
37     case expK:
38             weedEXP(t->val.expT.left);
39             break;
40     case notTermK:
41             weedTERM(t->val.nTermT.left);
42             break;
43     case lenExpK:
44             weedEXP(t->val.lenExpT.left);
45             break;
46     case numK:
47             break;
48     case boolK:
49             break;
50     case nullK:
51             break;
52   }
53 }
54
55 void weedVAR(VAR *v){
56       switch(v->kind){
57             case idVK:
58                   break;
59             case listVK:
60                   weedVAR(v->val.listV.left);
61                   weedEXP(v->val.listV.right);
62                   break;
63             case varidVK:
```

```
64                  weedVAR(v->val.varidV.left);
65                  break;
66        }
67  }
68
69  void weedACTLIST(ACTLIST *act){
70        switch(act->kind){
71              case explistK:
72                  weedEXPLIST(act->val.explistA.left);
73                  break;
74              case nilK:
75                  break;
76        }
77  }
78
79  void weedEXPLIST(EXPLIST *expl){
80        switch(expl->kind){
81              case expLK:
82                  weedEXP(expl->val.expL.left);
83                  break;
84              case explistLK:
85                  weedEXP(expl->val.explistL.left);
86                  weedEXPLIST(expl->val.explistL.right);
87                  break;
88        }
89  }
90
91  //
      ////////////////////////////////////////////////////////////////////////////

92  // This function iterates through a statement list and
      looks for if/else- and return statements.//
93  //
      ////////////////////////////////////////////////////////////////////////////

94  bool statementListChecker(STATE_LIST *sl){
95        STATE *slLeft = sl->val.statementListSL.left;
96        STATE_LIST *slRight = sl->val.statementListSL.right;
97
98        if(slLeft->kind == ifElseK){
99              // Passes both then- and else statement to
                    ifChecker().
100             if(ifChecker(slLeft) == true){
101                 if(slRight != NULL){
102                 fprintf(stderr, CYN "Note: Return
                        statement on line %d does not allow the
```

```
                    end of code to be reached.\n" RESET,
                    slLeft->lineno);
103             }
104             return true;
105         }
106     }
107     if(slLeft->kind == returnK){
108         if(slRight != NULL){
109             fprintf(stderr, CYN "Note: Return
                    statement on line %d does not allow the
                    end of code to be reached.\n" RESET,
                    slLeft->lineno);
110         }
111         return true;
112     }
113     if(slRight != NULL){
114         statementListChecker(slRight);
115     }else{
116         return false;
117     }
118 }
119
120 //
    ////////////////////////////////////////////////////////////

121 // This function checks both then and else.
                    //
122 // It returns true only if they both end with a return
    stament//
123 //
    ////////////////////////////////////////////////////////////

124 bool ifChecker(STATE *s){
125     STATE *middle = s->val.ifElseS.middle;
126     STATE *right = s->val.ifElseS.right;
127
128     if(thenElseChecker(middle) == true){
129         if(thenElseChecker(right) == true){
130             return true;
131         }
132     }
133     return false;
134 }
135
136 //
    ////////////////////////////////////////////////////////////////////////
```

```
137 // This function checks the current state for a return
    statement or a statement list. //
138 //
    ////////////////////////////////////////////////////////////////////////////

139 bool thenElseChecker(STATE *s){
140      if(s->kind == returnK){
141           return true;
142      }
143      else if(s->kind == ifElseK){
144           if(ifChecker(s) == true){
145                return true;
146           }
147      }
148      else if(s->kind == stateListK){
149           if(statementListChecker(s->val.stateListS.left)
                 == true){
150                return true;
151           }
152      }
153      return false;
154 }
155
156
157 void weedFunction(FUNCTION *f){
158     char *headID = (f->val.function.left)->val.head.id;
159     char *tailID = (f->val.function.right)->val.tail.id;
160     if(strcmp(headID,tailID) != 0){
161         fprintf(stderr, RED "Error: Id of function:"
                RESET);
162         fprintf(stderr, BLU " \"%s\" ", headID);
163         fprintf(stderr, RED "does not match end id. line
               %d\n" RESET, (f->val.function.left)->lineno);
164         fprintf(stderr, "exiting...\n");
165         exit(0);
166     }
167      weedHEAD(f->val.function.left);
168      if(statementListChecker((f->val.function.middle)->
           val.body.right) == false){
169           fprintf(stderr, MAG "Warning: Function: " RESET
                );
170           fprintf(stderr, BLU "\"%s\" ", headID);
171           fprintf(stderr, MAG "might not end with a
                return statement. line %d\n" RESET, (f->val.
                function.left)->lineno);
```

```
172         }
173         weedBODY(f->val.function.middle);
174         weedTAIL(f->val.function.right);
175   }
176
177   void weedHEAD(HEAD *h){
178         weedPAR_DECL_LIST(h->val.head.left);
179         weedTYPE(h->val.head.right);
180   }
181
182   void weedTAIL(TAIL *t){
183   }
184
185   void weedTYPE(TYPE *t){
186         switch(t->kind){
187               case idTyK:
188                     break;
189               case intTyK:
190                     break;
191               case boolTyK:
192                     break;
193               case arrayTyK:
194                     weedTYPE(t->val.array.left);
195                     break;
196               case recTyK:
197                     weedVAR_DECL_LIST(t->val.record.left);
198                     break;
199         }
200   }
201
202   void weedPAR_DECL_LIST(PAR_DECL_LIST *p){
203         switch(p->kind){
204               case nilPK:
205                     break;
206               case vdlPK:
207                     weedVAR_DECL_LIST(p->val.vdecl_list.left);
208                     break;
209         }
210   }
211
212   void weedVAR_DECL_LIST(VAR_DECL_LIST *vdl){
213         switch(vdl->kind){
214               case listVDLK:
215                     weedVAR_TYPE(vdl->val.listV.left);
216                     weedVAR_DECL_LIST(vdl->val.listV.right);
217                     break;
```

```
218             case vtK:
219                 weedVAR_TYPE(vdl->val.vtype.left);
220                 break;
221         }
222 }
223
224 void weedVAR_TYPE(VAR_TYPE *v){
225         weedTYPE(v->val.typeV.left);
226 }
227
228 void weedBODY(BODY *b){
229         fprintf(stderr, "HITT\n");
230         weedDECL_LIST(b->val.body.left);
231         fprintf(stderr, "HITT2\n");
232         weedSTATE_LIST(b->val.body.right);
233 }
234
235 void weedDECLARATION(DECLARATION *d){
236         switch(d->kind){
237             case typeIdK:
238                 weedTYPE(d->val.typeIdD.left);
239                 break;
240             case declFuncK:
241                 weedFunction(d->val.declFuncD);
242                 break;
243             case declVarK:
244                 weedVAR_DECL_LIST(d->val.declVarD);
245                 break;
246         }
247 }
248
249 void weedDECL_LIST(DECL_LIST *dl){
250         fprintf(stderr, "INNI\n");
251         switch(dl->kind){
252             case declListK:
253                 weedDECLARATION(dl->val.declListDL.left);
254                 weedDECL_LIST(dl->val.declListDL.right);
255                 break;
256             case nilDK:
257                 break;
258         }
259 }
260
261 void weedSTATE_LIST(STATE_LIST *sl){
262         fprintf(stderr, "INNI2\n");
263     switch(sl->kind){
```

```
264       case statementK:
265             weedSTATE(sl->val.statementSL);
266             break;
267       case statementListK:
268             weedSTATE(sl->val.statementListSL.left);
269             weedSTATE_LIST(sl->val.statementListSL.right);
270             break;
271    }
272
273 }
274
275 void weedSTATE(STATE *s){
276       switch(s->kind){
277             case returnK:
278                   weedEXP(s->val.returnS);
279                   break;
280             case writeK:
281                   weedEXP(s->val.writeS);
282                   break;
283             case allocateK:
284                   weedVAR(s->val.allocateS);
285                   break;
286             case allocOfLengthK:
287                   weedVAR(s->val.allocOfLengthS.left);
288                   weedEXP(s->val.allocOfLengthS.right);
289                   break;
290             case assignK:
291                   weedVAR(s->val.assignS.left);
292                   weedEXP(s->val.assignS.right);
293                   break;
294             case ifK:
295                   weedEXP(s->val.ifS.left);
296                   weedSTATE(s->val.ifS.right);
297                   break;
298             case ifElseK:
299                   weedEXP(s->val.ifElseS.left);
300                   weedSTATE(s->val.ifElseS.middle);
301                   weedSTATE(s->val.ifElseS.right);
302                   break;
303             case whileK:
304                   weedEXP(s->val.whileS.left);
305                   weedSTATE(s->val.whileS.right);
306                   break;
307             case stateListK:
308                   weedSTATE_LIST(s->val.stateListS.left);
309                   break;
```

```
310             case incK:
311                  weedVAR(s->val.incS);
312                  break;
313             case decK:
314                  weedVAR(s->val.decS);
315                  break;
316             case forincK:
317                  fprintf(stderr, "bara ga\n");
318                  weedDECL_LIST(s->val.forincS.first);
319                  weedTERM(s->val.forincS.second);
320                  weedTERM(s->val.forincS.third);
321                  weedSTATE(s->val.forincS.fourth);
322                  break;
323             case fordecK:
324                  weedDECL_LIST(s->val.fordecS.first);
325                  weedTERM(s->val.fordecS.second);
326                  weedTERM(s->val.fordecS.third);
327                  weedSTATE(s->val.fordecS.fourth);
328                  break;
329        }
330   }
```

A.14 types.h

```
1  #ifndef __types_h
2  #define __types_h
3
4  #include "symbol.h"
5  #include "memory.h"
6  #include <stdio.h>
7  typedef struct Ty_ty_ Ty_ty;
8  typedef struct Ty_tyList_ Ty_tyList;
9  typedef struct Ty_field_ Ty_field;
10 typedef struct Ty_fieldList_ Ty_fieldList;
11 typedef struct Ty_ty_func_ Ty_ty_func;
12
13
14 typedef struct Ty_ty_ {
15         enum {Ty_record, Ty_nil, Ty_int, Ty_string,
               Ty_array, Ty_name, Ty_void, Ty_bool} kind;
16         union {Ty_fieldList *record;
17                Ty_ty *array;
18                struct {char* name; SymbolTable *table;}
                      name;
19         } u;
20         Ty_tyList *equal;
```

```
21 } Ty_ty;
22
23 typedef struct Ty_ty_func_{
24         Ty_tyList *formals;
25         Ty_ty *type;
26 } Ty_ty_func;
27
28 Ty_ty *Ty_Nil(void);
29 Ty_ty *Ty_Int(void);
30 Ty_ty *Ty_String(void);
31 Ty_ty *Ty_Void(void);
32 Ty_ty *Ty_Bool(void);
33
34 Ty_ty *Ty_Record(Ty_fieldList *fields);
35 Ty_ty *Ty_Array(Ty_ty *ty);
36 Ty_ty *Ty_Name(char* name, SymbolTable *table);
37
38 Ty_ty_func *Ty_Func(Ty_tyList *formals, Ty_ty *type);
39
40 typedef struct Ty_tyList_ {Ty_ty *head; Ty_tyList *tail;}
       Ty_tyList;
41 Ty_tyList *Ty_TyList(Ty_ty *head, Ty_tyList *tail);
42
43 typedef struct Ty_field_ {char* name; Ty_ty *ty;}
     Ty_field;
44 Ty_field *Ty_Field(char* name, Ty_ty *ty);
45
46 typedef struct Ty_fieldList_ {Ty_field *head;
     Ty_fieldList *tail;} Ty_fieldList;
47 Ty_fieldList *Ty_FieldList(Ty_field *head, Ty_fieldList *
     tail);
48
49 Ty_tyList *FListToList(Ty_fieldList *list);
50
51 int tyComp(Ty_ty *t1, Ty_ty *t2);
52 int compRecords(Ty_ty *t1, Ty_ty *t2);
53 int compArray(Ty_ty *t1, Ty_ty *t2);
54
55 #endif
```

A.15 types.c

```
1 #include "types.h"
2
3
4 Ty_ty *Ty_Nil(void){
```

```
5       Ty_ty *type;
6       type = NEW(Ty_ty);
7       type->kind=Ty_nil;
8       return type;
9   }
10
11  Ty_ty *Ty_Int(void){
12      Ty_ty *type;
13      type = NEW(Ty_ty);
14      type->kind=Ty_int;
15      return type;
16  }
17
18  Ty_ty *Ty_String(void){
19      Ty_ty *type;
20      type = NEW(Ty_ty);
21      type->kind=Ty_string;
22      return type;
23  }
24
25  Ty_ty *Ty_Void(void){
26      Ty_ty *type;
27      type = NEW(Ty_ty);
28      type->kind=Ty_void;
29      return type;
30  }
31
32  Ty_ty *Ty_Bool(void){
33      Ty_ty *type;
34      type = NEW(Ty_ty);
35      type->kind=Ty_bool;
36      return type;
37  }
38
39  Ty_ty *Ty_Record(Ty_fieldList *fields){
40      Ty_ty *type;
41      type = NEW(Ty_ty);
42      type->kind=Ty_record;
43      type->u.record=fields;
44      return type;
45  }
46
47  Ty_ty *Ty_Array(Ty_ty *ty){
48      Ty_ty *type;
49      type = NEW(Ty_ty);
50      type->kind=Ty_array;
```

```
51      type->u.array=ty;
52      type->equal=NULL;
53      return type;
54  }
55
56  Ty_ty *Ty_Name(char* name, SymbolTable *table){
57      Ty_ty *type;
58      type = NEW(Ty_ty);
59      type->kind=Ty_name;
60      type->u.name.name=name;
61      type->u.name.table=table;
62      return type;
63  }
64
65  Ty_tyList *Ty_TyList(Ty_ty *head, Ty_tyList *tail){
66      Ty_tyList *list;
67      list=NEW(Ty_tyList);
68      list->head=head;
69      list->tail = tail;
70      return list;
71  }
72
73  Ty_field *Ty_Field(char* name, Ty_ty *ty){
74      Ty_field *field;
75      field = NEW(Ty_field);
76      field->name=name;
77      field->ty=ty;
78      return field;
79  }
80
81  Ty_fieldList *Ty_FieldList(Ty_field *head, Ty_fieldList *
        tail){
82      Ty_fieldList *flist;
83      flist = NEW(Ty_fieldList);
84      flist->head=head;
85      flist->tail=tail;
86      return flist;
87  }
88
89  Ty_ty_func *Ty_Func(Ty_tyList *formals, Ty_ty *type){
90      Ty_ty_func *func;
91      func = NEW(Ty_ty_func);
92      func->formals=formals;
93      func->type=type;
94  }
95
```

```
 96  Ty_tyList *FListToList(Ty_fieldList *flist){
 97      Ty_tyList *list = NULL;
 98      if(flist != NULL){
 99          list=NEW(Ty_tyList);
100          list->head=flist->head->ty;
101          list->tail=FListToList(flist->tail);
102      }
103      return list;
104  }
105
106  /*
107  1 means types equivalent
108  */
109  int tyComp(Ty_ty *t1, Ty_ty *t2){
110      if((t1 == NULL) || (t2 == NULL)){
111          return 0;
112      }
113
114      while(t1->kind == Ty_name){
115          t1=getSymbol(t1->u.name.table, t1->u.name.name)->
                value;
116          if(t1 == NULL){
117              return 0;
118          }
119      }
120
121      while(t2->kind == Ty_name){
122          t2=getSymbol(t2->u.name.table, t2->u.name.name)->
                value;
123          if(t2 == NULL){
124              return 0;
125          }
126      }
127
128      if(t1->kind == Ty_record && t2->kind ==Ty_nil){
129          return 1;
130      }
131
132      if(t1->kind == Ty_array && t2->kind ==Ty_nil){
133          return 1;
134      }
135
136
137      if(t1->kind == Ty_record && t2->kind == Ty_record){
138          if(compRecords(t1, t2) == 1){
139              return 1;
```

```
140         }
141         return 0;
142     }
143
144     if(t1->kind == Ty_array || t2->kind == Ty_array){
145         if(compArray(t1, t2) == 1){
146             return 1;
147         }
148         return 0;
149     }
150
151     if(t1->kind ==  t2->kind){
152         return 1;
153     }
154
155     return 0;
156 }
157 /*
158 Return 1 if equivalent
159 return 0 when not equivalent
160 !!problem, might endlessly recurse on recursive records
161 */
162 int compRecords(Ty_ty *t1, Ty_ty *t2){
163     Ty_fieldList *flist1 = t1->u.record;
164     Ty_fieldList *flist2 = t2->u.record;
165
166     Ty_tyList *list1 = t1->equal;
167     Ty_tyList *list2 = t2->equal;
168
169     while(list1 != NULL){
170         if(list1->head == t2){
171             return 1;
172         }
173         list1=list1->tail;
174     }
175
176     while(list2 != NULL){
177         if(list2->head == t1){
178             return 1;
179         }
180         list2=list2->tail;
181     }
182
183
184     t1->equal=Ty_TyList(t2, t1->equal);
185     t2->equal=Ty_TyList(t1, t2->equal);
```

```
186
187         while(flist1 != NULL && flist2 != NULL){
188             if(tyComp(flist1->head->ty, flist2->head->ty) !=
                    1){
189                 t1->equal=t1->equal->tail;
190                 t2->equal=t2->equal->tail;
191                 return 0;
192             }
193             flist1=flist1->tail;
194             flist2=flist2->tail;
195         }
196
197         if(flist1 == NULL && flist2 == NULL){
198             t1->equal=t1->equal->tail;
199             t2->equal=t2->equal->tail;
200             return 1;
201         }
202
203
204
205
206         t1->equal=t1->equal->tail;
207         t2->equal=t2->equal->tail;
208         return 0;
209     }
210
211     int compArray(Ty_ty *t1, Ty_ty *t2){
212         while(t1->kind == Ty_array){
213             t1 = t1->u.array;
214         }
215
216         while(t2->kind == Ty_array){
217             t2=t2->u.array;
218         }
219
220         if(tyComp(t1, t2) == 1){
221             return 1;
222         }
223
224         return 0;
225     }
226
227     /*int main()
228     {
229         Ty_ty *t = Ty_Array(Ty_Void());
230         if(t->u.array->kind == Ty_void){
```

```
231             printf("array of void");
232         }
233       return 0;
234     }*/
```

A.16 typecheck.h

```
1  #ifndef __typecheck_h
2  #define __typecheck_h
3  #include "tree.h"
4  #include "symbol.h"
5  #include "memory.h"
6  #include "types.h"
7  #include <string.h>
8
9  COLLECTION *collect;
10
11 COLLECTION *initCOLLECTION();
12 COLLECTION *scopeCOLLECTION(COLLECTION *c, Ty_ty *
       returntype);
13
14 int checkTREE(BODY *b);
15 void checkEXP(EXP *e, COLLECTION *c);
16 void checkTERM(TERM *t, COLLECTION *c);
17 void checkVAR(VAR *v, COLLECTION *c);
18 Ty_tyList *checkACTLIST(ACTLIST *act, COLLECTION *c);
19 Ty_tyList *checkEXPLIST(EXPLIST *expl, COLLECTION *c);
20 void checkSTATE(STATE *s, COLLECTION *c);
21 void checkSTATE_LIST(STATE_LIST *sl, COLLECTION *c);
22 void checkDECLARATION(DECLARATION *d, COLLECTION *c);
23 void checkDECL_LIST(DECL_LIST *dl, COLLECTION *c);
24 void checkFunction(FUNCTION *f, COLLECTION *c);
25 Ty_fieldList *checkHEAD(HEAD *h, COLLECTION *c);
26 void checkTAIL(TAIL *t, COLLECTION *c);
27 Ty_ty *checkTYPE(TYPE *t, COLLECTION *c);
28 Ty_fieldList *checkPAR_DECL_LIST(PAR_DECL_LIST *p,
       COLLECTION *c);
29 Ty_fieldList *checkVAR_DECL_LIST(VAR_DECL_LIST *vdl,
       COLLECTION *c);
30 Ty_field *checkVAR_TYPE(VAR_TYPE *v, COLLECTION *c);
31 void checkBODY(BODY *b, COLLECTION *c);
32 Ty_ty *getVarType(VAR *v, COLLECTION *c);
33 Ty_ty *getTermType(TERM *t, COLLECTION *c);
34 int EXPtype(EXP *e, COLLECTION *c);
35 void appendNestList(FUNCTION *head, COLLECTION *c);
36
```

```
37 #endif
```

A.17 typecheck.c

```
 1 #include <stdio.h>
 2 #include "typecheck.h"
 3
 4 int error = 0;
 5
 6 COLLECTION *initCOLLECTION(){
 7       COLLECTION *c;
 8       c = NEW(COLLECTION);
 9       c->function=initSymbolTable();
10       c->type=initSymbolTable();
11       c->var=initSymbolTable();
12       c->nestedlist=NULL;
13       c->returns=Ty_Nil();
14       c->next=NULL;
15       return c;
16 }
17
18 COLLECTION *scopeCOLLECTION(COLLECTION *c, Ty_ty *
      returntype){
19       COLLECTION *col;
20       col=NEW(COLLECTION);
21       col->function=scopeSymbolTable(c->function);
22       col->type=scopeSymbolTable(c->type);
23       col->var=scopeSymbolTable(c->var);
24       col->nestedlist=NULL;
25       col->returns=returntype;
26       col->next=c;
27       return col;
28 }
29
30 void appendNestList(FUNCTION *head, COLLECTION *c){
31       NestScopeList *list;
32       list = NEW(NestScopeList);
33       list->func=head;
34       list->next=c->nestedlist;
35       c->nestedlist = list;
36 }
37
38 int checkTREE(BODY *b){
39       collect = initCOLLECTION();
40       checkBODY(b, collect);
41       return error;
```

```
42 }
43
44 void checkEXP(EXP *e, COLLECTION *c)
45 { switch (e->kind) {
46       case binopK:
47            checkEXP(e->val.binopE.left, c);
48            checkEXP(e->val.binopE.right, c);
49            if (EXPtype(e, c)){
50                 fprintf(stderr, "Error unexpected types in
                         binary expression, in line %d\n", e->
                        lineno);
51                 error=1;
52            }
53            break;
54       case termK:
55            //printf("Here\n");
56            checkTERM(e->val.termE.left, c);
57            e->types.type = getTermType(e->val.termE.left,
                  c);
58            //printf("%d\n", e->types.type->kind);
59            break;
60   }
61 }
62
63
64 //Determines the type of an expression and it's children,
       then verifies that the children are of correct type,
     returning 1 if they are not.
65
66 int EXPtype(EXP *e, COLLECTION *c){
67       if (strcmp(e->val.binopE.op, "==")==0){
68            e->types.type=Ty_Bool();
69            e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                  Ty_Int(),NULL));
70            goto checksametype;
71       }
72       if (strcmp(e->val.binopE.op, ">=")==0){
73            e->types.type=Ty_Bool();
74            e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                  Ty_Int(),NULL));
75            goto checkexpect;
76       }
77       if (strcmp(e->val.binopE.op, "<=")==0){
78            e->types.type=Ty_Bool();
79            e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                  Ty_Int(),NULL));
```

```
80              goto checkexpect;
81          }
82          if (strcmp(e->val.binopE.op, "!=")==0){
83              e->types.type=Ty_Bool();
84              e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                    Ty_Int(),NULL));
85              goto checksametype;
86          }
87          if (strcmp(e->val.binopE.op, "<")==0){
88              e->types.type=Ty_Bool();
89              e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                    Ty_Int(),NULL));
90              goto checkexpect;
91          }
92          if (strcmp(e->val.binopE.op, ">")==0){
93              e->types.type=Ty_Bool();
94              e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                    Ty_Int(),NULL));
95              goto checkexpect;
96          }
97          if (strcmp(e->val.binopE.op, "-")==0){
98              e->types.type=Ty_Int();
99              e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                    Ty_Int(),NULL));
100             goto checkexpect;
101         }
102         if (strcmp(e->val.binopE.op, "+")==0){
103             e->types.type=Ty_Int();
104             e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                    Ty_Int(),NULL));
105             goto checkexpect;
106         }
107         if (strcmp(e->val.binopE.op, "*")==0){
108             e->types.type=Ty_Int();
109             e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                    Ty_Int(),NULL));
110             goto checkexpect;
111         }
112         if (strcmp(e->val.binopE.op, "/")==0){
113             e->types.type=Ty_Int();
114             e->types.args=Ty_TyList(Ty_Int(), Ty_TyList(
                    Ty_Int(),NULL));
115             goto checkexpect;
116         }
117         if (strcmp(e->val.binopE.op, " || ")==0){
118             e->types.type=Ty_Bool();
```

```
119             e->types.args=Ty_TyList(Ty_Bool(), Ty_TyList(
                    Ty_Bool(),NULL));
120             goto checkexpect;
121         }
122         if (strcmp(e->val.binopE.op, "&&")==0){
123             e->types.type=Ty_Bool();
124             e->types.args=Ty_TyList(Ty_Bool(), Ty_TyList(
                    Ty_Bool(),NULL));
125             goto checkexpect;
126         }
127
128         checkexpect:
129         //printf("%d, %d, %s\n", e->val.binopE.left->types.
                type->kind, e->val.binopE.right->types.type->kind
                , e->val.binopE.op);
130         if (tyComp(e->val.binopE.left->types.type, e->types.
                args->head) != 1 || tyComp(e->val.binopE.right->
                types.type, e->types.args->tail->head) != 1){
131             fprintf(stderr, "expression has unexpected
                    types in line %d\n", e->lineno);
132             error = 1;
133             return 1;
134         }
135         goto end;
136
137         checksametype:
138         if(tyComp(e->val.binopE.left->types.type, e->val.
                binopE.right->types.type) != 1){
139             fprintf(stderr, "Expected same type in line %d\
                    n", e->lineno);
140             error = 1;
141             return 1;
142         }
143         goto end;
144
145         end:
146         return 0;
147 }
148
149
150 Ty_ty *getTermType(TERM *t, COLLECTION *c){
151         switch(t->kind){
152             case varK:
153                 return getVarType(t->val.varT.left, c);
154             case idfuncK:
155                 return ((Ty_ty_func*)getSymbol(c->function
```

```
                    , t->val.idfuncT.id)->value)->type;
156             case expK:
157                 return t->val.expT.left->types.type;
158             case notTermK:
159                 return Ty_Bool(); //getTermType(t->val.
                        nTermT.left, c);
160             case lenExpK:
161                 return Ty_Int();
162             case numK:
163                 return Ty_Int();
164             case boolK:
165                 return Ty_Bool();
166             case nullK:
167                 return Ty_Nil();
168         }
169 }
170
171 // Returns the type of a variable. Returns NULL if no
        type found.
172 Ty_ty *getVarType(VAR *v, COLLECTION *c){
173         Ty_ty *ty;
174         SYMBOL *s;
175         switch(v->kind){
176             case idVK:
177                 s = getSymbol(c->var, v->val.idV);
178                 if (s == NULL){
179                     fprintf(stderr, "No variable %s, in
                            line %d\n", v->val.idV, v->lineno)
                            ;
180                     ty = NULL;
181                     error = 1;
182                 } else {
183                     ty = s->value;
184                 }
185                 return ty;
186                 break;
187             case listVK:
188                 return Ty_Array(getVarType(v->val.listV.
                        left, c));
189                 break;
190             case varidVK:
191                 ty = getVarType(v->val.varidV.left, c);
192                 while (ty->kind == Ty_name || ty->kind ==
                        Ty_array){
193                     if(ty->kind == Ty_name){
194                         ty = (Ty_ty*)getSymbol(ty->u.
```

```
                                name.table, ty->u.name.name)
                                ->value;
195                         }
196
197                         if(ty->kind == Ty_array){
198                              ty= ty->u.array;
199                         }
200                   }
201
202                   Ty_fieldList *fl;
203                   fl = ty->u.record;
204                   while(fl != NULL){
205                         if (strcmp(fl->head->name, v->val.
                                varidV.id) == 0){
206                           return fl->head->ty;
207                         }
208                         fl = fl->tail;
209                   }
210       }
211       return NULL;
212  }
213
214  void checkTERM(TERM *t, COLLECTION *c){
215    switch(t->kind){
216       Ty_tyList *list;
217       Ty_tyList *formals;
218      case varK:
219            checkVAR(t->val.varT.left, c);
220            break;
221      case idfuncK:
222            list = checkACTLIST(t->val.idfuncT.left, c);
223            formals = ((Ty_ty_func*) getSymbol(c->function,
                    t->val.idfuncT.id)->value)->formals;
224            while((list != NULL) && (formals != NULL)){
225                  if (list->head->kind != formals->head->
                        kind){
226                        fprintf(stderr, "wrong parameters for
                              function in %d kind %d and %d\n",
                              t->lineno, list->head->kind,
                             formals->head->kind);
227                        error = 1;
228                  }
229                  list=list->tail;
230                  formals=formals->tail;
231            }
232            if(list != NULL || formals !=NULL){
```

```
233                 fprintf(stderr, "Wrong amount of variables
                        in %d \n", t->lineno);
234                 error = 1;
235           }
236           break;
237     case expK:
238           checkEXP(t->val.expT.left, c);
239           break;
240     case notTermK:
241           checkTERM(t->val.nTermT.left, c);
242           break;
243     case lenExpK:
244           checkEXP(t->val.lenExpT.left, c);
245           break;
246     case numK:
247           break;
248     case boolK:
249           break;
250     case nullK:
251           break;
252   }
253 }
254
255 void checkVAR(VAR *v, COLLECTION *c){
256     switch(v->kind){
257           case idVK:
258                 break;
259           case listVK:
260                 checkVAR(v->val.listV.left, c);
261                 checkEXP(v->val.listV.right, c);
262                 break;
263           case varidVK:
264                 checkVAR(v->val.varidV.left, c);
265                 break;
266     }
267 }
268
269 Ty_tyList *checkACTLIST(ACTLIST *act, COLLECTION *c){
270     switch(act->kind){
271           case explistK:
272                 return checkEXPLIST(act->val.explistA.left
                        , c);
273                 break;
274           case nilK:
275                 return NULL;
276                 break;
```

```
277         }
278     }
279
280     Ty_tyList *checkEXPLIST(EXPLIST *expl, COLLECTION *c){
281         switch(expl->kind){
282             Ty_tyList *list;
283             case expLK:
284                 checkEXP(expl->val.expL.left, c);
285                 list = Ty_TyList(expl->val.expL.left->
                        types.type, NULL);
286                 break;
287             case explistLK:
288                 checkEXP(expl->val.explistL.left, c);
289                 list=Ty_TyList(expl->val.explistL.left->
                        types.type, checkEXPLIST(expl->val.
                        explistL.right, c));
290                 break;
291         }
292     }
293
294     void checkFunction(FUNCTION *f, COLLECTION *c){
295         Ty_fieldList *variables;
296         variables = checkHEAD(f->val.function.left, c);
297         c=scopeCOLLECTION(c, checkTYPE(f->val.function.left
            ->val.head.right, c));
298
299         while(variables != NULL){
300             putSymbol(c->var, variables->head->name,
                    variables->head->ty);
301             variables=variables->tail;
302         }
303         checkBODY(f->val.function.middle, c);
304         c=c->next;
305         checkTAIL(f->val.function.right, c);
306     }
307
308     Ty_fieldList *checkHEAD(HEAD *h, COLLECTION *c){
309         Ty_fieldList *flist;
310         flist = checkPAR_DECL_LIST(h->val.head.left, c);
311         return flist;
312     }
313
314     void checkTAIL(TAIL *t, COLLECTION *c){
315     }
316
317     Ty_ty *checkTYPE(TYPE *t, COLLECTION *c){
```

```
318       Ty_ty *ty;
319       switch(t->kind){
320             case idTyK:
321                   ty = Ty_Name(t->val.id, c->type);
322                   return ty;
323                   break;
324             case intTyK:
325                   ty=Ty_Int();
326                   return ty;
327             case boolTyK:
328                   ty = Ty_Bool();
329                   return ty;
330                   break;
331             case arrayTyK:
332                   ty = Ty_Array(checkTYPE(t->val.array.left,
                          c));
333                   return ty;
334                   break;
335             case recTyK:
336                   ty = Ty_Record(checkVAR_DECL_LIST(t->val.
                         record.left, c));
337                   return ty;
338                   break;
339       }
340       fprintf(stderr, "error unusable type in line %d\n",
             t->lineno);
341       error=1;
342       return ty;
343 }
344
345 Ty_fieldList *checkPAR_DECL_LIST(PAR_DECL_LIST *p,
       COLLECTION *c){
346       switch(p->kind){
347             Ty_fieldList *tyfl;
348             case nilPK:
349                   tyfl=NULL;;
350                   return tyfl;
351                   break;
352             case vdlPK:
353                   tyfl = checkVAR_DECL_LIST(p->val.
                         vdecl_list.left, c);
354                   return tyfl;
355                   break;
356       }
357 }
358
```

```
359 Ty_fieldList *checkVAR_DECL_LIST(VAR_DECL_LIST *vdl,
      COLLECTION *c){
360       switch(vdl->kind){
361             case listVDLK:
362                   return Ty_FieldList(checkVAR_TYPE(vdl->val
                          .listV.left, c), checkVAR_DECL_LIST(vdl
                          ->val.listV.right, c));
363                   break;
364             case vtK:
365                   return Ty_FieldList(checkVAR_TYPE(vdl->val
                          .vtype.left, c), NULL);
366                   break;
367       }
368 }
369
370 Ty_field *checkVAR_TYPE(VAR_TYPE *v, COLLECTION *c){
371       return Ty_Field(v->val.typeV.id, checkTYPE(v->val.
            typeV.left, c));
372 }
373
374 void checkBODY(BODY *b, COLLECTION *c){
375       b->c=c;
376       NestScopeList *nl;
377       SymbolList *sl;
378       Ty_ty *ty;
379       Ty_tyList *TypesSeen = NULL;
380       Ty_tyList *NameCheck;
381       checkDECL_LIST(b->val.body.left, c);
382       //check for name types with no actual type
383       sl = AllSymbolsInScope(c->type);
384       while(sl != NULL){
385             ty = sl->head->value;
386             while(ty->kind == Ty_name){
387                   NameCheck = TypesSeen;
388                   while(NameCheck != NULL){
389                         if(NameCheck->head == ty){
390                               fprintf(stderr, "name recursion\
                                    n");
391                               break;
392                         }
393                         NameCheck = NameCheck->tail;
394                   }
395                   if(NameCheck != NULL){
396                         error = 1;
397                         break;
398                   }
```

```
399                 TypesSeen = Ty_TyList(ty, TypesSeen);
400                 ty = getSymbol(ty->u.name.table, ty->u.
                        name.name)->value;
401           }
402           TypesSeen = NULL;
403           sl = sl->tail;
404     }
405     //end check for name types with no actual type
406     checkSTATE_LIST(b->val.body.right, c);
407     nl = c->nestedlist;
408     while(nl != NULL){
409           checkFunction(nl->func, c);
410           nl=nl->next;
411     }
412 }
413
414 void declareFunction(FUNCTION *func, COLLECTION *c){
415     HEAD *h = func->val.function.left;
416     Ty_fieldList *flist;
417     flist = checkPAR_DECL_LIST(h->val.head.left, c);
418     putSymbol(c->function, h->val.head.id, Ty_Func(
            FListToList(flist), checkTYPE(h->val.head.right,
            c)));
419     appendNestList(func, c);
420 }
421
422 void checkDECLARATION(DECLARATION *d, COLLECTION *c){
423     Ty_fieldList *tyf;
424     switch(d->kind){
425           case typeIdK:
426                 putSymbol(c->type, d->val.typeIdD.id,
                        checkTYPE(d->val.typeIdD.left, c));
427                 break;
428           case declFuncK:
429                 declareFunction(d->val.declFuncD, c);
430                 break;
431           case declVarK:
432                 tyf = checkVAR_DECL_LIST(d->val.declVarD,
                        c);
433                 while(tyf != NULL){
434                       putSymbol(c->var,tyf->head->name, tyf
                              ->head->ty);
435                       tyf=tyf->tail;
436                 }
437                 break;
438     }
```

```
439 }
440
441 void checkDECL_LIST(DECL_LIST *dl, COLLECTION *c){
442       switch(dl->kind){
443             case declListK:
444                   checkDECLARATION(dl->val.declListDL.left,
                          c);
445                   checkDECL_LIST(dl->val.declListDL.right, c
                          );
446                   break;
447             case nilDK:
448                   break;
449       }
450 }
451
452 void checkSTATE_LIST(STATE_LIST *sl, COLLECTION *c){
453   switch(sl->kind){
454     case statementK:
455           checkSTATE(sl->val.statementSL, c);
456           break;
457     case statementListK:
458           checkSTATE(sl->val.statementListSL.left, c);
459           checkSTATE_LIST(sl->val.statementListSL.right,
                 c);
460           break;
461   }
462
463 }
464
465 void checkSTATE(STATE *s, COLLECTION *c){
466       switch(s->kind){
467             case returnK:
468                   checkEXP(s->val.returnS, c);
469                   if(tyComp(c->returns, s->val.returnS->
                          types.type) != 1){
470                     fprintf(stderr, "Error wrong return
                             type in line %d\n", s->lineno);
471                     error=1;
472                   }
473                   break;
474             case writeK:
475                   checkEXP(s->val.writeS, c);
476                   break;
477             case allocateK:
478                   checkVAR(s->val.allocateS, c);
479                   break;
```

```
480             case allocOfLengthK:
481                  checkVAR(s->val.allocOfLengthS.left, c);
482                  checkEXP(s->val.allocOfLengthS.right, c);
483                  if(tyComp(s->val.allocOfLengthS.right->
                         types.type, Ty_Int()) != 1){
484                       fprintf(stderr, "Error length not int
                              in line %d \n", s->lineno);
485                       error = 1;
486                  }
487                  break;
488             case assignK:
489                  checkEXP(s->val.assignS.right, c);
490                  if(tyComp(getVarType(s->val.assignS.left,
                         c), s->val.assignS.right->types.type)
                         != 1){
491                       fprintf(stderr, "Error assign in line
                              %d\n", s->val.assignS.right->
                             lineno);
492                       error = 1;
493                  }
494                  break;
495             case ifK:
496                  checkEXP(s->val.ifS.left, c);
497                  if(tyComp(s->val.ifS.left->types.type,
                         Ty_Bool()) != 1){
498                       fprintf(stderr, "Error if without
                              bool expression in line %d \n", s
                             ->lineno);
499                       error=1;
500                  }
501                  checkSTATE(s->val.ifS.right, c);
502                  break;
503             case ifElseK:
504                  checkEXP(s->val.ifElseS.left, c);
505                  if(tyComp(s->val.ifElseS.left->types.type,
                        Ty_Bool()) != 1){
506                       fprintf(stderr, "Error if without
                              bool expression in line %d \n", s
                             ->lineno);
507                       error = 1;
508                  }
509                  checkSTATE(s->val.ifElseS.middle, c);
510                  checkSTATE(s->val.ifElseS.right, c);
511                  break;
512             case whileK:
513                  checkEXP(s->val.whileS.left, c);
```

```
514                 if(tyComp(s->val.whileS.left->types.type,
                        Ty_Bool()) != 1){
515                   fprintf(stderr, "Error while without
                          bool expression in line %d \n", s
                          ->lineno);
516                   error=1;
517                 }
518                 checkSTATE(s->val.whileS.right, c);
519                 break;
520           case stateListK:
521                 checkSTATE_LIST(s->val.stateListS.left, c)
                        ;
522                 break;
523           case incK:
524                 checkVAR(s->val.incS, c);
525                 break;
526           case decK:
527                 checkVAR(s->val.decS, c);
528                 break;
529           case forincK:
530                 checkDECL_LIST(s->val.forincS.first, c);
531                 checkTERM(s->val.forincS.second, c);
532                 checkTERM(s->val.forincS.third, c);
533                 checkSTATE(s->val.forincS.fourth, c);
534                 break;
535           case fordecK:
536                 checkDECL_LIST(s->val.fordecS.first, c);
537                 checkTERM(s->val.fordecS.second, c);
538                 checkTERM(s->val.fordecS.third, c);
539                 checkSTATE(s->val.fordecS.fourth, c);
540                 break;
541       }
542 }
```

A.18 TEMP.h

```
1  #ifndef __TEMP_h
2  #define __TEMP_h
3  #include "memory.h"
4
5  int count;
6
7  typedef struct temp{
8      int num;
9  } temp;
10
```

```
11 temp *requestTemp();
12
13 #endif
```

A.19 TEMP.c

```
1  #include "TEMP.h"
2
3
4  temp *requestTemp(){
5      temp *t;
6      t = NEW(temp);
7      count=count+1;
8      t->num=count;
9      return t;
10 }
```

A.20 frame.h

```
1  #ifndef __frame_h
2  #define __frame_h
3
4  #include "tree.h"
5  #include "memory.h"
6  #include <string.h>
7  /* stackframe layout
8  Decending order arguments, each 16 bytes long.
9  Single link to enclosing frame 8 bytes
10 Return address 8 bytes
11 previous base pointer 8 bytes        <-stack pointer
       points here
12 argument count 8 bytes
13 local variable count 8 bytes
14 ascending order locals 16 bytes each
15 callee save registers 8 bytes
16 temporary values 8 bytes each
17 */
18 typedef struct linkedlist_ linkedlist;
19 typedef struct frame_ frame;
20 typedef struct faccess_ faccess;
21
22
23
24 typedef struct frame_{
25     int level;
26     linkedlist *formal;
```

```
27      linkedlist *local;
28      linkedlist *func;
29      frame *enclosing;
30      COLLECTION *c;
31  } frame;
32
33  typedef struct faccess_{
34      int depth;
35      int offset;
36  } faccess;
37
38  faccess *getAccess(char *c, frame *f);
39
40
41  typedef struct linkedlist_{char *name; linkedlist *next;}
        linkedlist;
42
43  frame *createFrame(frame *outerframe);
44  linkedlist *createList(char *c);
45  void addArg(char *arg, frame *f);
46  void addVars(linkedlist *list, frame *f);
47  void addFunc(char *func, frame *f);
48  void initFrame(HEAD *h, BODY *b, frame *fr);
49  void buildAccess(char *c, frame *f, faccess *fa);
50  faccess *getAccess(char *c,frame *f);
51  void defineLocals(BODY *b, frame *f);
52  void frameDECL_LIST(DECL_LIST *dl, frame *f);
53  void frameDECLARATION(DECLARATION *d, frame *f);
54  void defineParams(HEAD *h, frame *f);
55  linkedlist *framePAR_DECL_LIST(PAR_DECL_LIST *p, frame *f
      );
56  linkedlist *frameVAR_DECL_LIST(VAR_DECL_LIST *vdl, frame
      *f, linkedlist *list);
57  char *frameVAR_TYPE(VAR_TYPE *v, frame *f);
58  int locCount(frame *f);
59  int argCount(frame *f);
60  int getFuncDepth(frame *f, char * func, int i);
61
62  #endif
```

A.21 frame.c

```
1  #include "frame.h"
2
3
4  int const argdist = 24; //distance to 1 argument from
```

```
    frame pointer
5   int const localdist = 32; //distance in bytes to first
    local
6   int const size = 16; //size of values in bytes
7
8   int getFuncDepth(frame *f, char * func, int i){
9       frame *fr = f;
10      linkedlist *list = f->func;
11      while( list != NULL){
12          if(strcmp(func, list->name) == 0){
13              return i;
14          }
15          list = list->next;
16      }
17
18      if(f->level>0){
19          i = i+1;
20          return getFuncDepth(f->enclosing, func, i);
21      }
22
23      return -1;
24  }
25
26  int argCount(frame *f){
27      linkedlist *l;
28      int i = 0;
29      l=f->formal;
30      while(l != NULL){
31          i++;
32          l=l->next;
33      }
34      return i;
35  }
36
37  int locCount(frame *f){
38      linkedlist *l;
39      int i =0;
40      l=f->local;
41      while(l != NULL){
42          i++;
43          l=l->next;
44      }
45      return i;
46  }
47
48  faccess *getAccess(char *c,frame *f){
```

```
49      faccess *fa;
50      fa=NEW(faccess);
51      fa->depth=0;
52      buildAccess(c, f, fa);
53      return fa;
54  }
55
56  void buildAccess(char *c, frame *f, faccess *fa){
57      linkedlist *list;
58      list = f->formal;
59      int i = 0;
60      while(list != NULL){
61          if(strcmp(c, list->name) == 0){
62              fa->offset=size*i+argdist;
63              return;
64          }
65          i++;
66          list = list->next;
67      }
68
69      i = 0;
70      list = f->local;
71      while(list != NULL){
72          if(strcmp(c, list->name) == 0){
73              fa->offset=size*i-localdist;
74              return;
75          }
76          i--;
77          list = list->next;
78      }
79
80      if(f->level != 0){
81          fa->depth++;
82          buildAccess(c,f->enclosing,fa);
83          return;
84      }
85
86  }
87
88
89
90  void initFrame(HEAD *h, BODY *b, frame *fr){
91      if (h != NULL){
92          defineParams(h, fr);
93      }
94      if(b != NULL){
```

```
95          defineLocals(b, fr);
96      }
97      fr->c=b->c;
98  }
99
100 frame  *createFrame(frame *f){
101     frame *nf;
102     nf = NEW(frame);
103     nf->enclosing=f;
104     if(f==NULL){
105         nf->level=0;
106     } else {
107         nf->level=f->level + 1;
108     }
109     nf->local=NULL;
110     nf->func=NULL;
111     nf->formal=NULL;
112     return nf;
113 }
114
115 linkedlist *createList(char *c){
116     linkedlist *list;
117     list = NEW(linkedlist);
118     list->name=c;
119     return list;
120 }
121
122 void addVars(linkedlist *list, frame *f){
123     linkedlist *l;
124
125     if(f->local == NULL){
126         f->local = list;
127         return;
128     }
129
130     l=f->local;
131     while(l->next != NULL){
132         l=l->next;
133     }
134     l->next=list;
135 }
136
137 void addFunc(char *c, frame *f){
138     linkedlist *list;
139
140     if(f->func == NULL){
```

```
141             f->func = createList(c);
142             return;
143         }
144 
145         list = f->func;
146         while(list->next != NULL){
147             list=list->next;
148         }
149         list->next=createList(c);
150     }
151 
152     void defineLocals(BODY *b, frame *f){
153         frameDECL_LIST(b->val.body.left, f);
154     }
155 
156     void frameDECL_LIST(DECL_LIST *dl, frame *f){
157         switch(dl->kind){
158             case declListK:
159                 frameDECLARATION(dl->val.declListDL.left,
                        f);
160                 frameDECL_LIST(dl->val.declListDL.right, f
                        );
161                 break;
162             case nilDK:
163                 break;
164         }
165     }
166 
167     void frameDECLARATION(DECLARATION *d, frame *f){
168         switch(d->kind){
169             case typeIdK:
170                 break;
171             case declFuncK:
172                 addFunc(d->val.declFuncD->val.function.
                        left->val.head.id, f);
173                 break;
174             case declVarK:
175                 addVars(frameVAR_DECL_LIST(d->val.declVarD
                        , f, NULL), f);
176                 break;
177         }
178     }
179 
180 
181     void defineParams(HEAD *h, frame *f){
182         f->formal = framePAR_DECL_LIST(h->val.head.left, f);
```

```
183 }
184
185 linkedlist *framePAR_DECL_LIST(PAR_DECL_LIST *p, frame *f
    ){
186       linkedlist *list;
187       list=NULL;
188       switch(p->kind){
189             case nilPK:
190                   break;
191             case vdlPK:
192                   list = frameVAR_DECL_LIST(p->val.
                          vdecl_list.left, f, list);
193                   break;
194       }
195       return list;
196 }
197
198 linkedlist *frameVAR_DECL_LIST(VAR_DECL_LIST *vdl, frame
    *f, linkedlist *list){
199       switch(vdl->kind){
200             case listVDLK:
201                   list = createList(frameVAR_TYPE(vdl->val.
                          vtype.left, f));
202                   list->next = frameVAR_DECL_LIST(vdl->val.
                          listV.right, f, list->next);
203                   break;
204             case vtK:
205                   list = createList(frameVAR_TYPE(vdl->val.
                          vtype.left, f));
206                   break;
207       }
208       return list;
209 }
210
211 char *frameVAR_TYPE(VAR_TYPE *v, frame *f){
212       return v->val.typeV.id;
213 }
```

A.22 codegen.h

```
1 #ifndef __codegen_h
2 #define __codegen_h
3 #include "tree.h"
4 #include "TEMP.h"
5 #include "frame.h"
6 #include "typecheck.h"
```

```
7  #include <string.h>
8  #include "memory.h"
9  #include <stdio.h>
10
11 FILE *f;
12
13
14
15 void expADD(EXP *e, temp *tem, frame *fr);
16 void expSUB(EXP *e, temp *tem, frame *fr);
17 void expMUL(EXP *e, temp *tem, frame *fr);
18 void expDIV(EXP *e, temp *tem, frame *fr);
19 void EXPswitch(EXP *e, temp *tem, frame *fr);
20 void codeEXP(EXP *e, temp *tem, frame *fr);
21 void codeTERM(TERM *t, temp *tem, frame *fr);
22 void codeVAR(VAR *v, frame *fr, temp *tem);
23 int codeACTLIST(ACTLIST *act, frame *fr);
24 int codeEXPLIST(EXPLIST *expl, frame *fr);
25 void codeSTATE(STATE *s, frame *fr);
26 void codeSTATE_LIST(STATE_LIST *sl, frame *fr);
27 void codeDECLARATION(DECLARATION *d, frame *fr);
28 void codeDECL_LIST(DECL_LIST *dl, frame *fr);
29 void codeFunction(FUNCTION *fun, frame *fr);
30 void codeHEAD(HEAD *h, frame *fr);
31 void codeTAIL(TAIL *t, frame *fr);
32 void codeTYPE(TYPE *t, frame *fr);
33 void codePAR_DECL_LIST(PAR_DECL_LIST *p, frame *fr);
34 linkedlist *codeVAR_DECL_LIST(VAR_DECL_LIST *vdl, frame *
       fr);
35 char *codeVAR_TYPE(VAR_TYPE *v, frame *r);
36 void codeBODY(BODY *b, frame *f);
37 void allocLength();
38 void getMem();
39 void getVar(faccess *fa, temp *tem);
40 void generateLocalVariable(char *c, frame *fr);
41 void allocOfLength(VAR *var, temp *tvar, temp *argument,
       frame *fr);
42 void allocate(temp *tvar, frame *fr, VAR *v);
43 void writer();
44
45 #endif
```

A.23 codegen.c

```
1  #include "codegen.h"
2
```

```
3  int tempVar = 0;
4
5  int writecounter=0;
6
7  int ifcount=0;
8
9  int whilecount = 0;
10
11 int expjump = 0;
12
13 int forcount = 0;
14
15 int ANDcount = 0;
16
17 int ORcount = 0;
18
19 int ABScount = 0;
20
21
22
23
24 void codeGEN(BODY *b){
25       frame *fr;
26       NestScopeList *nl;
27       fr=createFrame(NULL);
28       initFrame(NULL, b, fr);
29       f = fopen("./output.asm", "w");
30       if(f == NULL){
31            fprintf(stderr, "error opening file\n");
32            return;
33       }
34       //data section
35       fprintf(f, ".section .data\n");
36       fprintf(f, "   initialBrk: .quad 0\n");
37       fprintf(f, "   currentBrk: .quad 0\n");
38       fprintf(f, "   newBrk: .quad 0\n");
39       fprintf(f, "   freelist: .quad 0\n");
40       fprintf(f, "   Callstack: .space 400\n");
41       fprintf(f, "   top: .quad 0\n");
42       //text section
43       fprintf(f, ".section .text\n");
44       //fprintf(f, ".globl _start\n\n");
45       fprintf(f, ".globl main\n\n");
46
47       //main program
48       //fprintf(f, "_start:\n");
```

```
49        fprintf(f, "main:\n");
50        initProgram();
51        fprintf(f, "    push %%rbp\n");
52        fprintf(f, "    movq %%rsp, %%rbp\n");
53        fprintf(f, "    call pushframe\n");
54        fprintf(f, "    push $%d\n", argCount(fr));
55        fprintf(f, "    push $%d\n", locCount(fr));
56
57       codeBODY(b, fr);
58
59        fprintf(f, "    pop %%rax\n");
60        fprintf(f, "    pop %%rax\n");
61        fprintf(f, "    call popframe\n");
62        fprintf(f, "    movq %%rbp, %%rsp\n");
63        fprintf(f, "    pop %%rbp\n");
64        fprintf(f, "    movq $60, %%rax\n");
65        fprintf(f, "    movq $0, %%rdi\n");
66        fprintf(f, "    syscall\n");
67        fprintf(f, "\n");
68       getMem();
69        allocLength();
70        popFrame(fr);
71        pushFrame(fr);
72        writer();
73
74        nl=fr->c->nestedlist;
75
76        while(nl != NULL){
77              codeFunction(nl->func,fr);
78              nl=nl->next;
79        }
80
81
82
83        fclose(f);
84  }
85
86  void initProgram(){
87        fprintf(f, "    movq $12, %%rax\n");
88   fprintf(f, "    movq $0, %%rdi\n");
89   fprintf(f, "    syscall\n");
90        fprintf(f, "    movq %%rax, (initialBrk)\n");
91        fprintf(f, "    movq %%rax, (currentBrk)\n");
92        fprintf(f, "    movq %%rax, (newBrk)\n");
93        fprintf(f, "    movq $Callstack, (top)\n");
94  }
```

```
 95
 96  void pushFrame(frame *fr){
 97       fprintf(f, "pushframe:\n");
 98       fprintf(f, "    movq (top), %%rax\n");
 99       fprintf(f, "    movq %%rbp, (%%rax)\n");
100       fprintf(f, "    addq $8, (top)\n");
101       fprintf(f, "    ret\n");
102  }
103
104  void popFrame(frame *fr){
105       fprintf(f, "popframe:\n");
106       fprintf(f, "    subq $8, (top)\n");
107       fprintf(f, "    movq (top), %%rax\n");
108       fprintf(f, "    movq $0, (%%rax)\n");
109       fprintf(f, "    ret\n");
110  }
111
112  void printPushCallee(){
113       fprintf(f, "    push %%r12\n");
114       fprintf(f, "    push %%r13\n");
115       fprintf(f, "    push %%r14\n");
116       fprintf(f, "    push %%r15\n");
117  }
118
119  void printPopCallee(){
120       fprintf(f, "    pop %%r15\n");
121       fprintf(f, "    pop %%r14\n");
122       fprintf(f, "    pop %%r13\n");
123       fprintf(f, "    pop %%r12\n");
124  }
125
126  void printPopCaller(){
127       fprintf(f, "    pop %%rbx\n");
128       fprintf(f, "    pop %%rcx\n");
129       fprintf(f, "    pop %%rdx\n");
130       fprintf(f, "    pop %%rsi\n");
131       fprintf(f, "    pop %%rdi\n");
132       fprintf(f, "    pop %%r8\n");
133       fprintf(f, "    pop %%r9\n");
134       fprintf(f, "    pop %%r10\n");
135       fprintf(f, "    pop %%r11\n");
136
137  }
138
139  void printPushCaller(){
140       fprintf(f, "    push %%r11\n");
```

```
141         fprintf(f, "    push %%r10\n");
142         fprintf(f, "    push %%r9\n");
143         fprintf(f, "    push %%r8\n");
144         fprintf(f, "    push %%rdi\n");
145         fprintf(f, "    push %%rsi\n");
146         fprintf(f, "    push %%rdx\n");
147         fprintf(f, "    push %%rcx\n");
148         fprintf(f, "    push %%rbx\n");
149 }
150
151 void codeEXP(EXP *e, temp *tem, frame *fr)
152 { switch (e->kind) {
153       case binopK:
154             EXPswitch(e, tem, fr);
155             break;
156       case termK:
157             codeTERM(e->val.termE.left, tem, fr);
158             break;
159   }
160 }
161 //add rest of operators
162 void EXPswitch(EXP *e, temp *tem, frame *fr){
163       if(strcmp("+", e->val.binopE.op) == 0){
164             expADD(e, tem, fr);
165       }
166       if(strcmp("-", e->val.binopE.op) == 0){
167             expSUB(e, tem, fr);
168       }
169       if(strcmp("*", e->val.binopE.op) == 0){
170             expMUL(e, tem, fr);
171       }
172       if(strcmp("/", e->val.binopE.op) == 0){
173             expDIV(e, tem, fr);
174       }
175       if(strcmp(" || ", e->val.binopE.op) == 0){
176             expOR(e, tem, fr);
177       }
178       if(strcmp("==", e->val.binopE.op) == 0){
179             expEQ(e, tem, fr);
180       }
181       if(strcmp("!=", e->val.binopE.op) == 0){
182             expNEQ(e, tem, fr);
183       }
184       if(strcmp(">=", e->val.binopE.op) == 0){
185             expGTE(e, tem, fr);
186       }
```

```
187         if(strcmp(">", e->val.binopE.op) == 0){
188             expGT(e, tem, fr);
189         }
190         if(strcmp("<=", e->val.binopE.op) == 0){
191             expLTE(e, tem, fr);
192         }
193         if(strcmp("<", e->val.binopE.op) == 0){
194             expLT(e, tem, fr);
195         }
196         if(strcmp("&&", e->val.binopE.op) == 0){
197             expAND(e, tem, fr);
198         }
199
200 }
201
202 void expAND(EXP *e, temp *tem, frame *fr){
203         temp *arg1;
204         temp *arg2;
205         int label = ANDcount;
206         ANDcount++;
207         arg1 = requestTemp();
208         arg2 = requestTemp();
209
210         codeEXP(e->val.binopE.left, arg1, fr);
211         fprintf(f, "    cmp $0, @%d\n",arg1->num);
212         fprintf(f, "    je lazyAND%d\n", label);
213         codeEXP(e->val.binopE.right, arg2, fr);
214         fprintf(f, "lazyAND%d:\n", label);
215
216         fprintf(f, "    AND @%d, @%d\n", arg1->num, arg2->num
                );
217         fprintf(f, "    movq @%d, @%d\n", arg2->num, tem->num
                );
218
219 }
220
221 void expOR(EXP *e, temp *tem, frame *fr){
222         temp *arg1;
223         temp *arg2;
224         int label = ORcount;
225         ORcount++;
226         arg1 = requestTemp();
227         arg2 = requestTemp();
228         fprintf(f, "    movq $0, @%d\n", arg1->num);
229         codeEXP(e->val.binopE.left, arg1, fr);
230         fprintf(f, "    movq $0, @%d\n", arg2->num);
```

```
231
232         fprintf(f, "    cmp $1, @%d\n", arg1->num);
233         fprintf(f, "    je lazyOR%d\n", label);
234
235         codeEXP(e->val.binopE.right, arg2, fr);
236
237         fprintf(f, "lazyOR%d:\n", label);
238         fprintf(f, "    OR @%d, @%d\n", arg1->num, arg2->num)
                ;
239         fprintf(f, "    movq @%d, @%d\n", arg2->num, tem->num
                );
240
241
242 }
243
244 void expEQ(EXP *e, temp *tem, frame *fr){
245         temp *arg1;
246         temp *arg2;
247         arg1 = requestTemp();
248         arg2 = requestTemp();
249
250         codeEXP(e->val.binopE.left, arg1, fr);
251         codeEXP(e->val.binopE.right, arg2, fr);
252
253         fprintf(f, "    cmp @%d, @%d\n", arg1->num, arg2->num
                );
254         fprintf(f, "    jne explabel%d\n", expjump);
255         fprintf(f, "    movq $1, @%d\n", tem->num);
256         fprintf(f, "    jmp expend%d\n", expjump);
257         fprintf(f, "explabel%d:\n", expjump);
258         fprintf(f, "    movq $0, @%d\n", tem->num);
259         fprintf(f, "expend%d:\n", expjump);
260         expjump++;
261 }
262
263 void expNEQ(EXP *e, temp *tem, frame *fr){
264         temp *arg1;
265         temp *arg2;
266         arg1 = requestTemp();
267         arg2 = requestTemp();
268
269         codeEXP(e->val.binopE.left, arg1, fr);
270         codeEXP(e->val.binopE.right, arg2, fr);
271
272         fprintf(f, "    cmp @%d, @%d\n", arg1->num, arg2->num
                );
```

```
273         fprintf(f, "    jne explabel%d\n", expjump);
274         fprintf(f, "    movq $0, @%d\n", tem->num);
275         fprintf(f, "    jmp expend%d\n", expjump);
276         fprintf(f, "explabel%d:\n", expjump);
277         fprintf(f, "    movq $1, @%d\n", tem->num);
278         fprintf(f, "expend%d:\n", expjump);
279         expjump++;
280 }
281
282 void expGT(EXP *e, temp *tem, frame *fr){
283         temp *arg1;
284         temp *arg2;
285         arg1 = requestTemp();
286         arg2 = requestTemp();
287
288         codeEXP(e->val.binopE.left, arg1, fr);
289         codeEXP(e->val.binopE.right, arg2, fr);
290
291         fprintf(f, "    cmp @%d, @%d\n", arg2->num, arg1->num
                );
292         fprintf(f, "    jg explabel%d\n", expjump);
293         fprintf(f, "    movq $0, @%d\n", tem->num);
294         fprintf(f, "    jmp expend%d\n", expjump);
295         fprintf(f, "explabel%d:\n", expjump);
296         fprintf(f, "    movq $1, @%d\n", tem->num);
297         fprintf(f, "expend%d:\n", expjump);
298         expjump++;
299 }
300
301 void expGTE(EXP *e, temp *tem, frame *fr){
302         temp *arg1;
303         temp *arg2;
304         arg1 = requestTemp();
305         arg2 = requestTemp();
306
307         codeEXP(e->val.binopE.left, arg1, fr);
308         codeEXP(e->val.binopE.right, arg2, fr);
309
310         fprintf(f, "    cmp @%d, @%d\n", arg2->num, arg1->num
                );
311         fprintf(f, "    jge explabel%d\n", expjump);
312         fprintf(f, "    movq $0, @%d\n", tem->num);
313         fprintf(f, "    jmp expend%d\n", expjump);
314         fprintf(f, "explabel%d:\n", expjump);
315         fprintf(f, "    movq $1, @%d\n", tem->num);
316         fprintf(f, "expend%d:\n", expjump);
```

```
317         expjump++;
318     }
319
320     void expLT(EXP *e, temp *tem, frame *fr){
321         temp *arg1;
322         temp *arg2;
323         arg1 = requestTemp();
324         arg2 = requestTemp();
325
326         codeEXP(e->val.binopE.left, arg1, fr);
327         codeEXP(e->val.binopE.right, arg2, fr);
328
329         fprintf(f, "    cmp @%d, @%d\n", arg2->num, arg1->num
                );
330         fprintf(f, "    jl explabel%d\n", expjump);
331         fprintf(f, "    movq $0, @%d\n", tem->num);
332         fprintf(f, "    jmp expend%d\n", expjump);
333         fprintf(f, "explabel%d:\n", expjump);
334         fprintf(f, "    movq $1, @%d\n", tem->num);
335         fprintf(f, "expend%d:\n", expjump);
336         expjump++;
337     }
338
339     void expLTE(EXP *e, temp *tem, frame *fr){
340         temp *arg1;
341         temp *arg2;
342         arg1 = requestTemp();
343         arg2 = requestTemp();
344
345         codeEXP(e->val.binopE.left, arg1, fr);
346         codeEXP(e->val.binopE.right, arg2, fr);
347
348         fprintf(f, "    cmp @%d, @%d\n", arg2->num, arg1->num
                );
349         fprintf(f, "    jle explabel%d\n", expjump);
350         fprintf(f, "    movq $0, @%d\n", tem->num);
351         fprintf(f, "    jmp expend%d\n", expjump);
352         fprintf(f, "explabel%d:\n", expjump);
353         fprintf(f, "    movq $1, @%d\n", tem->num);
354         fprintf(f, "expend%d:\n", expjump);
355         expjump++;
356     }
357
358     void expADD(EXP *e, temp *tem, frame *fr){
359         temp *arg1;
360         temp *arg2;
```

```
361         arg1 = requestTemp();
362         arg2 = requestTemp();
363 
364         codeEXP(e->val.binopE.left, arg1, fr);
365         codeEXP(e->val.binopE.right, arg2, fr);
366 
367         fprintf(f, "    movq @%d, %%rax\n", arg1->num);
368         fprintf(f, "    addq @%d, %%rax\n", arg2->num);
369         fprintf(f, "    movq %%rax, @%d\n", tem->num);
370 
371 }
372 
373 void expSUB(EXP *e, temp *tem, frame *fr){
374         temp *arg1;
375         temp *arg2;
376         arg1 = requestTemp();
377         arg2 = requestTemp();
378         codeEXP(e->val.binopE.left, arg1, fr);
379         codeEXP(e->val.binopE.right, arg2, fr);
380         fprintf(f, "    movq @%d, %%rax\n", arg1->num);
381         fprintf(f, "    subq @%d, %%rax\n", arg2->num);
382         fprintf(f, "    movq %%rax, @%d\n", tem->num);
383 }
384 
385 void expMUL(EXP *e, temp *tem, frame *fr){
386         temp *arg1;
387         temp *arg2;
388         arg1 = requestTemp();
389         arg2 = requestTemp();
390         codeEXP(e->val.binopE.left, arg1, fr);
391         codeEXP(e->val.binopE.right, arg2, fr);
392 
393         fprintf(f, "    movq @%d, %%rax\n", arg1->num);
394         fprintf(f, "    imulq @%d, %%rax\n", arg2->num);
395         fprintf(f, "    movq %%rax, @%d\n", tem->num);
396 }
397 
398 void expDIV(EXP *e, temp *tem, frame *fr){
399         temp *arg1;
400         temp *arg2;
401         arg1 = requestTemp();
402         arg2 = requestTemp();
403         codeEXP(e->val.binopE.left, arg1, fr);
404         codeEXP(e->val.binopE.right, arg2, fr);
405 
406         fprintf(f, "    push %%rdx\n");
```

```
407         fprintf(f, "    push %%rax\n");
408         fprintf(f, "    movq $0, %%rdx\n", arg1->num);
409         fprintf(f, "    movq @%d, %%rax\n", arg1->num);
410         fprintf(f, "    idivq @%d\n", arg2->num);
411         fprintf(f, "    movq %%rax, @%d\n", tem->num);
412         fprintf(f, "    pop %%rax\n");
413         fprintf(f, "    pop %%rdx\n");
414 }
415
416
417 void codeTERM(TERM *t, temp *tem, frame *fr){
418   temp *var;
419   temp *result;
420   int depth;
421   int count;
422   Ty_ty *type;
423   switch(t->kind){
424     case varK:
425           var = requestTemp();
426           codeVAR(t->val.varT.left, fr, var);
427           fprintf(f, "    movq @%d, %%rcx\n", var->num);
428           fprintf(f, "    movq 8(%%rcx), %%rdi\n", var->
                num);
429           fprintf(f, "    movq %%rdi, @%d\n", tem->num);
430           break;
431     case idfuncK:
432           count = codeACTLIST(t->val.idfuncT.left, fr);
433           depth = getFuncDepth(fr, t->val.idfuncT.id, 0);
434           fprintf(f, "    movq %%rbp, %%rdi\n");
435           while(depth != 0){
436                 fprintf(f, "    movq 16(%%rdi), %%rdi\n");
437                 depth--;
438           }
439           fprintf(f, "    push %%rdi\n");
440           fprintf(f, "    call %s\n", t->val.idfuncT.id);
441           fprintf(f, "    movq %%rax, @%d\n", tem->num);
442           fprintf(f, "    addq $8, %%rsp\n");
443           fprintf(f, "    addq $%d, %%rsp\n", 16*count);
444           break;
445     case expK:
446           result = requestTemp();
447           codeEXP(t->val.expT.left, result, fr);
448           fprintf(f, "    movq @%d, %%rax\n", result->num)
                ;
449           fprintf(f, "    movq %%rax, @%d\n", tem->num);
450           break;
```

```
451     case notTermK:
452           result = requestTemp();
453           codeTERM(t->val.nTermT.left, result, fr);
454           fprintf(f, "    movq @%d, @%d\n", result->num,
                  tem->num);
455           fprintf(f, "    xor $1, @%d\n",tem->num);
456           break;
457     case lenExpK:
458           result = requestTemp();
459           type = t->val.lenExpT.left->types.type;
460           while(type->kind == Ty_name){
461                 type = getSymbol(type->u.name.table, type
                        ->u.name.name)->value;
462           }
463           codeEXP(t->val.lenExpT.left, result, fr);
464           switch(type->kind){
465                 case Ty_array:
466                       fprintf(f, "    movq @%d, %%rsi\n",
                              result->num);
467                       fprintf(f, "    movq 16(%%rsi), @%d\n"
                              , tem->num);
468                       break;
469                 case Ty_int:
470                       fprintf(f, "    movq @%d, @%d\n",
                              result->num, tem->num);
471                       fprintf(f, "    cmp $0, @%d\n", tem->
                              num);
472                       fprintf(f, "    JGE Abs%d\n", ABScount
                              );
473                       fprintf(f, "    neg @%d\n", tem->num);
474                       fprintf(f, "Abs%d:\n", ABScount);
475                       ABScount++;
476                       break;
477                 default:
478                       fprintf(f, "# %d ", t->val.lenExpT.
                              left->types.type->kind);
479                       fprintf(f, "    movq $0, @%d\n", tem->
                              num);
480                       break;
481           }
482
483           break;
484     case numK:
485           fprintf(f, "    movq $%d, @%d\n", t->val.
                  intconstT, tem->num);
486           break;
```

```
487       case boolK:
488             fprintf(f, "    movq $%d, @%d\n", t->val.
                    boolconstT, tem->num);
489             if(t->val.boolconstT == 0){
490             } else if (t->val.boolconstT == 1)
491             {
492             }
493             break;
494       case nullK:
495             fprintf(f, "    movq $0, @%d\n", tem->num);
496             break;
497    }
498 }
499
500 void codeVAR(VAR *v, frame *fr, temp *tem){
501       temp *tvar = requestTemp();
502       temp *tlen = requestTemp();
503       int recIndex = 0;
504       Ty_ty *varTy;
505       Ty_fieldList *fl;
506       faccess *fa;
507       switch(v->kind){
508             case idVK:
509                   fa = getAccess(v->val.idV, fr);
510                   getVar(fa, tvar);
511                   fprintf(f, "    movq @%d, %%rdx\n", tvar->
                          num);
512                   fprintf(f, "    movq %%rdx, @%d\n", tem->
                          num);
513                   break;
514             case listVK:
515                   codeVAR(v->val.listV.left, fr, tvar);
516                   codeEXP(v->val.listV.right, tlen, fr);
517                   fprintf(f, "    movq @%d, %%rdx\n", tvar->
                          num);
518                   fprintf(f, "    movq 8(%%rdx), %%rsi\n");
519                   fprintf(f, "    addq $24, %%rsi\n");
520                   fprintf(f, "    movq @%d, %%rax\n", tlen->
                          num);
521                   fprintf(f, "    movq $16, %%rdi\n");
522                   fprintf(f, "    imulq %%rdi\n");
523                   fprintf(f, "    addq %%rax, %%rsi\n");
524                   fprintf(f, "    movq %%rsi, @%d\n", tem->
                          num);
525                   break;
526             case varidVK:
```

```
527                 codeVAR(v->val.varidV.left, fr, tvar);
528                 varTy = getVarType(v->val.varidV.left, fr
                        ->c);
529
530
531                 actualType:
532                 if(varTy->kind == Ty_name){
533                       varTy = getSymbol(varTy->u.name.table
                              , varTy->u.name.name)->value;
534                       goto actualType;
535                 }
536                 if(varTy->kind == Ty_array){
537                       varTy = varTy->u.array;
538                       goto actualType;
539                 }
540
541
542
543                 fl = varTy->u.record;
544                 fprintf(stderr, "%d, %d, %d\n", v->lineno,
                          varTy->kind, v->val.varidV.left->kind)
                        ;
545                 while(strcmp(v->val.varidV.id, fl->head->
                        name) != 0){
546                       recIndex++;
547                       fl = fl->tail;
548                 }
549                 fprintf(f, "    movq @%d, %%rdx\n", tvar->
                        num);
550                 fprintf(f, "    movq 8(%%rdx), %%rsi\n");
551                 fprintf(f, "    addq $24, %%rsi\n");
552                 fprintf(f, "    movq $%d, %%rax\n",
                        recIndex);
553                 fprintf(f, "    movq $16, %%rdi\n");
554                 fprintf(f, "    imulq %%rdi\n");
555                 fprintf(f, "    addq %%rax, %%rsi\n");
556                 fprintf(f, "    movq %%rsi, @%d\n", tem->
                        num);
557                 break;
558       }
559 }
560
561 void getVar(faccess *fa, temp *tem){
562       int depth;
563       depth = fa->depth;
564       fprintf(f, "    movq %%rbp, %%rdi\n");
```

```
565         while(depth != 0){
566             fprintf(f, "    movq 16(%%rdi), %%rdi\n");
567             depth--;
568         }
569         fprintf(f, "    addq $%d, %%rdi\n", fa->offset);
570         fprintf(f, "    movq %%rdi, @%d\n", tem->num);
571     }
572
573     int codeACTLIST(ACTLIST *act, frame *fr){
574         switch(act->kind){
575             case explistK:
576                 return codeEXPLIST(act->val.explistA.left,
                        fr);
577                 break;
578             case nilK:
579                 return 0;
580                 break;
581         }
582     }
583
584     int codeEXPLIST(EXPLIST *expl, frame *fr){
585         temp *arg = requestTemp();
586         int i;
587         Ty_ty *ty = expl->val.expL.left->types.type;
588         while(ty->kind == Ty_name){
589             ty= getSymbol(ty->u.name.table, ty->u.name.name
                    )->value;
590         }
591         switch(expl->kind){
592             case expLK:
593                 codeEXP(expl->val.expL.left, arg, fr);
594                 fprintf(f, "    push @%d\n", arg->num);
595                 switch(ty->kind){
596                     case Ty_int:
597                         fprintf(f, "    push $0\n");
598                         break;
599                     case Ty_bool:
600                         fprintf(f, "    push $0\n");
601                         break;
602                     case Ty_record:
603                         fprintf(f, "    push $1\n");
604                         break;
605                     case Ty_array:
606                         fprintf(f, "    push $1\n");
607                         break;
608                     default:
```

```
609                         break;
610                 }
611                 return 1;
612                 break;
613             case explistLK:
614                 codeEXP(expl->val.explistL.left, arg, fr);
615                 i = 1 + codeEXPLIST(expl->val.explistL.
                        right, fr);
616                 fprintf(f, "    push @%d\n", arg->num);
617                 switch(ty->kind){
618                     case Ty_int:
619                         fprintf(f, "    push $0\n");
620                         break;
621                     case Ty_bool:
622                         fprintf(f, "    push $0\n");
623                         break;
624                     case Ty_record:
625                         fprintf(f, "    push $1\n");
626                         break;
627                     case Ty_array:
628                         fprintf(f, "    push $1\n");
629                         break;
630                     default:
631                         break;
632                 }
633                 return i;
634                 break;
635         }
636 }
637
638 void codeFunction(FUNCTION *fun, frame *fr){
639         fprintf(f, "%s:\n", fun->val.function.left->val.head
                .id); //to be replaced with generated label
640         fprintf(f, "    push %%rbp\n");
641         fprintf(f, "    movq %%rsp, %%rbp\n");
642         //callee save here
643         NestScopeList *nl;
644         frame *newframe;
645         newframe = createFrame(fr);
646         initFrame(fun->val.function.left, fun->val.function.
                middle, newframe);
647         fprintf(f, "    push $%d\n", argCount(newframe));
648         fprintf(f, "    push $%d\n", locCount(newframe));
649         //codeHEAD(fun->val.function.left, newframe);
650         codeBODY(fun->val.function.middle, newframe);
651         fprintf(f, "    addq $16, %%rsp\n");
```

```
652
653         fprintf(f, "    movq %%rbp, %%rsp\n");
654         fprintf(f, "    pop %%rbp\n");
655         //codeTAIL(fun->val.function.right, newframe);
656         fprintf(f, "    ret\n");
657
658         nl = newframe->c->nestedlist;
659         while(nl != NULL){
660             codeFunction(nl->func, newframe);
661             nl = nl->next;
662         }
663
664 }
665
666 void codeHEAD(HEAD *h, frame *fr){
667         codePAR_DECL_LIST(h->val.head.left, fr);
668         codeTYPE(h->val.head.right, fr);
669 }
670
671 void codeTAIL(TAIL *t, frame *fr){
672 }
673
674 void codeTYPE(TYPE *t, frame *fr){
675         switch(t->kind){
676             case idTyK:
677                 break;
678             case intTyK:
679                 break;
680             case boolTyK:
681                 break;
682             case arrayTyK:
683                 codeTYPE(t->val.array.left, fr);
684                 break;
685             case recTyK:
686                 codeVAR_DECL_LIST(t->val.record.left, fr);
687                 break;
688         }
689 }
690
691 void codePAR_DECL_LIST(PAR_DECL_LIST *p, frame *fr){
692         switch(p->kind){
693             case nilPK:
694                 break;
695             case vdlPK:
696                 codeVAR_DECL_LIST(p->val.vdecl_list.left,
                        fr);
```

```
697                 break;
698         }
699 }
700
701 linkedlist *codeVAR_DECL_LIST(VAR_DECL_LIST *vdl, frame *
        fr){
702         linkedlist *list;
703         switch(vdl->kind){
704                 case listVDLK:
705                         list = createList(codeVAR_TYPE(vdl->val.
                                vtype.left, fr));
706                         list->next=codeVAR_DECL_LIST(vdl->val.
                                listV.right, fr);
707                         break;
708                 case vtK:
709                         list = createList(codeVAR_TYPE(vdl->val.
                                vtype.left, fr));
710                         break;
711         }
712         return list;
713 }
714
715 char *codeVAR_TYPE(VAR_TYPE *v, frame *fr){
716         codeTYPE(v->val.typeV.left, fr);
717         return v->val.typeV.id;
718 }
719
720 void codeBODY(BODY *b, frame *fr){
721         int locals = locCount(fr);
722         codeDECL_LIST(b->val.body.left, fr);
723         printPushCallee();
724         codeSTATE_LIST(b->val.body.right, fr);
725         printPopCallee();
726         while(locals != 0){
727                 /*fprintf(f, "    pop %%rax\n");
728                 fprintf(f, "    pop %%rax\n");*/
729                 locals--;
730         }
731 }
732
733 void codeDECLARATION(DECLARATION *d, frame *fr){
734         linkedlist *locals;
735         switch(d->kind){
736                 case typeIdK:
737                         //codeTYPE(d->val.typeIdD.left, fr);
738                         // printf(";\n");
```

```
739                 break;
740             case declFuncK:
741                 //codeFunction(d->val.declFuncD, fr);
742                 break;
743             case declVarK:
744                 locals = codeVAR_DECL_LIST(d->val.declVarD
                        , fr);
745                 while(locals != NULL){
746                     generateLocalVariable(locals->name,
                            fr);
747                     locals = locals->next;
748                 }
749                 break;
750         }
751 }
752
753 void generateLocalVariable(char *c, frame *fr){
754         int recSize=0;
755         int recLength=0;
756         int recIndex=0;
757         Ty_fieldList *list;
758         Ty_ty *ty = getSymbol(fr->c->var, c)->value;
759         while(ty->kind == Ty_name){
760             ty=getSymbol(ty->u.name.table, ty->u.name.name)
                    ->value;
761         }
762         switch(ty->kind){
763             case Ty_int:
764                 fprintf(f, "    push $0\n");
765                 fprintf(f, "    push $0\n"); //header
766                 break;
767             case Ty_bool:
768                 fprintf(f, "    push $0\n");
769                 fprintf(f, "    push $0\n"); //header
770                 break;
771             case Ty_record:
772                 fprintf(f, "    push $0\n");
773                 fprintf(f, "    push $1\n"); //header
774
775                 /*recSize = 3;
776                 list = ty->u.record;
777                 while(list != NULL){
778                     recSize = recSize +2;
779                     recLength++;
780                     list=list->tail;
781                 }
```

```
782                 fprintf(f, "    movq $%d, %%rax\n", recSize
                        );
783                 fprintf(f, "    call getMem\n");
784                 fprintf(f, "    push %%rax\n");
785                 fprintf(f, "    push $1\n"); //header
786
787                 fprintf(f, "    movq $0, (%%rax)\n");
788                 fprintf(f, "    movq $%d, 8(%%rax)\n",
                        recSize);
789                 fprintf(f, "    movq $%d, 16(%%rax)\n",
                        recLength);
790                 list=ty->u.record;
791                 while(list != NULL){
792                     ty = list->head->ty;
793                     while(ty->kind == Ty_name){
794                         ty=getSymbol(ty->u.name.table,
                                ty->u.name.name)->value;
795                     }
796
797                     switch(ty->kind){
798                         case Ty_int:
799                             fprintf(f, "    movq $0, %d
                                    (%%rax)\n", (3+recIndex
                                    *2)*8 ); //header
800                             fprintf(f, "    movq $0, %d
                                    (%%rax)\n", (4+recIndex
                                    *2)*8 );
801                             break;
802                         case Ty_bool:
803                             fprintf(f, "    movq $0, %d
                                    (%%rax)\n", (3+recIndex
                                    *2)*8); //header
804                             fprintf(f, "    movq $0, %d
                                    (%%rax)\n", (4+recIndex
                                    *2)*8 );
805                             break;
806                         case Ty_record:
807                             fprintf(f, "    movq $1, %d
                                    (%%rax)\n", (3+recIndex
                                    *2)*8); //header
808                             fprintf(f, "    movq $0, %d
                                    (%%rax)\n", (4+recIndex
                                    *2)*8 );
809                             break;
810                         case Ty_array:
811                             fprintf(f, "    movq $1, %d
```

```
                                    (%%rax)\n", (3+recIndex
                                    *2)*8); //header
812                                 fprintf(f, "    movq $0, %d
                                    (%%rax)\n", (4+recIndex
                                    *2)*8 );
813                                 break;
814                             default:
815                                 break;
816                         }
817                         recIndex++;
818                         list = list->tail;
819                     }*/
820                     break;
821             case Ty_array:
822                     fprintf(f, "    push $0\n");
823                     fprintf(f, "    push $1\n"); //header
824                     break;
825             default:
826                     break;
827
828         }
829 }
830
831
832 //reserves space equal to 8 * rax in bytes, and returns
        address in rax
833 void getMem(){
834         fprintf(f, "getMem:\n");
835         printPushCallee();
836         printPushCaller();
837         fprintf(f, "    movq (currentBrk), %%rdi\n");
838         fprintf(f, "    movq $8, %%rdx\n");
839         fprintf(f, "    imulq %%rdx\n");
840         fprintf(f, "    addq %%rax, %%rdi\n");
841         fprintf(f, "    movq $12, %%rax\n");
842         fprintf(f, "    syscall\n");
843         fprintf(f, "    movq %%rax, (newBrk)\n");
844         fprintf(f, "    movq %%rax, (currentBrk);\n");
845         printPopCaller();
846         printPopCallee();
847         fprintf(f, "    ret\n");
848 }
849
850 void codeDECL_LIST(DECL_LIST *dl, frame *fr){
851         switch(dl->kind){
852             case declListK:
```

```
853                 codeDECLARATION(dl->val.declListDL.left,
                        fr);
854                 codeDECL_LIST(dl->val.declListDL.right, fr
                        );
855                 break;
856           case nilDK:
857                 break;
858     }
859 }
860
861 void codeSTATE_LIST(STATE_LIST *sl, frame *fr){
862   switch(sl->kind){
863     case statementK:
864           codeSTATE(sl->val.statementSL, fr);
865           break;
866     case statementListK:
867           codeSTATE(sl->val.statementListSL.left, fr);
868           codeSTATE_LIST(sl->val.statementListSL.right,
                  fr);
869           break;
870   }
871
872 }
873
874 void codeSTATE(STATE *s, frame *fr){
875     temp *write;
876     temp *wcount;
877     temp *tvar;
878     temp *argument;
879     Ty_ty *varTy;
880     faccess *fa;
881     int label;
882     switch(s->kind){
883           case returnK:
884                 tvar = requestTemp();
885                 codeEXP(s->val.returnS, tvar, fr);
886                 fprintf(f, "    movq @%d, %%rax\n", tvar->
                      num);
887                 printPopCallee();
888                 fprintf(f, "    movq %%rbp, %%rsp\n");
889                 fprintf(f, "    pop %%rbp\n");
890                 fprintf(f, "    ret\n");
891                 break;
892           case writeK:
893                 write = requestTemp();
894                 wcount = requestTemp();
```

```
895                 codeEXP(s->val.writeS, write, fr);
896
897                 fprintf(f, "    push %%rbp\n");
898                 fprintf(f, "    movq %%rsp, %%rbp\n");
899
900                 fprintf(f, "    push %%rax\n");
901                 fprintf(f, "    push %%rcx\n");
902                 fprintf(f, "    push %%rdx\n");
903                 fprintf(f, "    push %%rdi\n");
904                 fprintf(f, "    push %%rsi\n");
905
906                 fprintf(f, "    push @%d\n", write->num);
907                 fprintf(f, "    movq @%d, %%rax\n", write->
                        num);
908                 fprintf(f, "    call write\n");
909                 fprintf(f, "    pop @%d\n", write->num);
910
911                 fprintf(f, "    pop %%rsi\n");
912                 fprintf(f, "    pop %%rdi\n");
913                 fprintf(f, "    pop %%rdx\n");
914                 fprintf(f, "    pop %%rcx\n");
915                 fprintf(f, "    pop %%rax\n");
916
917                 fprintf(f, "    movq %%rbp, %%rsp\n");
918                 fprintf(f, "    pop %%rbp\n");
919
920                 /*fprintf(f, "    push %%rbp\n");
921                 fprintf(f, "    movq %%rsp, %%rbp\n");
922                 fprintf(f, "    push %%rax\n");
923                 fprintf(f, "    push %%rcx\n");
924                 fprintf(f, "    push %%rdx\n");
925                 fprintf(f, "    push %%rdi\n");
926                 fprintf(f, "    push %%rsi\n");
927
928                 fprintf(f, "    movq $0, @%d\n", wcount->
                        num);
929                 fprintf(f, "    push $10\n");
930                 fprintf(f, "    addq $1, @%d\n", wcount->
                        num);
931                 fprintf(f, "    cmp $0, @%d\n", write->num)
                        ;
932                 fprintf(f, "    jge positive%d\n",
                        writecounter);
933
934
935                 fprintf(f, "    push $45\n");
```

```
936                 fprintf(f, "    movq $1, %%rax\n");
937                 fprintf(f, "    movq $1, %%rdi\n");
938                 fprintf(f, "    movq %%rsp, %%rsi\n");
939                 fprintf(f, "    movq $1, %%rdx\n");
940                 fprintf(f, "    syscall\n");
941                 fprintf(f, "    addq $8, %%rsp\n");
942
943                 fprintf(f, "    movq @%d, %%rsi\n", write->
                        num);
944                 fprintf(f, "    neg %%rsi\n");
945                 fprintf(f, "    movq %%rsi, @%d\n", write->
                        num);
946                 fprintf(f, "    addq $0, @%d\n", write->num
                        );
947                 fprintf(f, "    movq @%d, %%rax\n", write->
                        num);
948                 fprintf(f, "    jmp writeloop%d\n",
                        writecounter);
949                 fprintf(f, "positive%d:\n", writecounter);
950                 fprintf(f, "    movq @%d, %%rax\n", write->
                        num);
951                 fprintf(f, "writeloop%d: \n", writecounter
                        );
952                 fprintf(f, "    movq $0, %%rdx\n");
953                 fprintf(f, "    movq $10, %%rcx\n");
954                 fprintf(f, "    idivq %%rcx\n");
955                 fprintf(f, "    addq $48, %%rdx\n");
956                 fprintf(f, "    push %%rdx\n");
957                 fprintf(f, "    addq $1, @%d\n", wcount->
                        num);
958                 fprintf(f, "    cmp $0, %%rax\n");
959                 fprintf(f, "    jne writeloop%d\n",
                        writecounter);
960
961                 fprintf(f, "printloop%d:\n", writecounter)
                        ;
962                 fprintf(f, "    movq $1, %%rax\n");
963                 fprintf(f, "    movq $1, %%rdi\n");
964                 fprintf(f, "    movq %%rsp, %%rsi\n");
965                 fprintf(f, "    movq $1, %%rdx\n");
966                 fprintf(f, "    syscall\n");
967                 fprintf(f, "    addq $8, %%rsp\n");
968                 fprintf(f, "    addq $-1, @%d\n", wcount->
                        num);
969                 fprintf(f, "    cmp $0, @%d\n", wcount->num
                        );
```

```
970                 fprintf(f, "    jne printloop%d\n",
                        writecounter);
971
972                 fprintf(f, "    pop %%rsi\n");
973                 fprintf(f, "    pop %%rdi\n");
974                 fprintf(f, "    pop %%rdx\n");
975                 fprintf(f, "    pop %%rcx\n");
976                 fprintf(f, "    pop %%rax\n");
977                 fprintf(f, "    movq %%rbp, %%rsp\n");
978                 fprintf(f, "    pop %%rbp\n");
979                 writecounter++;*/
980                 break;
981             case allocateK:
982                 tvar = requestTemp();
983                 codeVAR(s->val.allocateS, fr, tvar);
984                 allocate(tvar, fr, s->val.allocateS);
985                 break;
986             case allocOfLengthK:
987                 tvar = requestTemp();
988                 argument = requestTemp();
989                 codeVAR(s->val.allocOfLengthS.left, fr,
                        tvar);
990                 codeEXP(s->val.allocOfLengthS.right,
                        argument, fr);
991                 allocOfLength(s->val.allocOfLengthS.left,
                        tvar, argument, fr);
992                 break;
993             case assignK:
994                 tvar = requestTemp();
995                 argument = requestTemp();
996                 codeVAR(s->val.assignS.left, fr, tvar);
997                 codeEXP(s->val.assignS.right, argument, fr
                        );
998                 fprintf(f, "    movq @%d, %%rcx\n", tvar->
                        num);
999                 fprintf(f, "    movq @%d, 8(%%rcx)\n",
                        argument->num);
1000                break;
1001            case ifK:
1002                argument = requestTemp();
1003                label = ifcount;
1004                ifcount++;
1005                fprintf(f, "if%d:\n", label);
1006                codeEXP(s->val.ifS.left, argument, fr);
1007                fprintf(f, "    cmp $1, @%d\n", argument->
                        num);
```

```
1008                fprintf(f, "    jne ifend%d\n", label);
1009                codeSTATE(s->val.ifS.right, fr);
1010                fprintf(f, "ifend%d:\n", label);
1011                break;
1012            case ifElseK:
1013                argument= requestTemp();
1014                label = ifcount;
1015                ifcount++;
1016                fprintf(f, "if%d:\n", label);
1017                codeEXP(s->val.ifElseS.left, argument, fr)
                        ;
1018                fprintf(f, "    cmp $1, @%d\n", argument->
                       num);
1019                fprintf(f, "    jne ifelse%d\n", label);
1020                codeSTATE(s->val.ifElseS.middle, fr);
1021                fprintf(f, "    jmp ifend%d\n", label);
1022                fprintf(f, "ifelse%d:\n", label);
1023                codeSTATE(s->val.ifElseS.right, fr);
1024                fprintf(f, "ifend%d:\n", label);
1025                break;
1026            case whileK:
1027                argument = requestTemp();
1028                int label = whilecount;
1029                whilecount++;
1030                fprintf(f, "while%d:\n", label);
1031                codeEXP(s->val.whileS.left, argument, fr);
1032                fprintf(f, "    cmp $1, @%d\n", argument->
                       num);
1033                fprintf(f, "    jne whileend%d\n", label);
1034                codeSTATE(s->val.whileS.right, fr);
1035                fprintf(f, "    jmp while%d\n", label);
1036                fprintf(f, "whileend%d:\n", label);
1037                break;
1038            case stateListK:
1039                codeSTATE_LIST(s->val.stateListS.left, fr)
                        ;
1040                break;
1041            case incK:
1042                tvar = requestTemp();
1043                codeVAR(s->val.incS, fr, tvar);
1044             fprintf(f, "    movq @%d, %%rcx\n", tvar->num);
1045                fprintf(f, "    addq $1, 8(%%rcx)\n");
1046                break;
1047            case decK:
1048                tvar = requestTemp();
1049                codeVAR(s->val.decS, fr, tvar);
```

```
1050                fprintf(f, "    movq @%d, %%rcx\n", tvar->
                        num);
1051                fprintf(f, "    subq $1, 8(%%rcx)\n");
1052                break;
1053            case forincK:
1054                tvar = requestTemp();
1055                codeDECL_LIST(s->val.forincS.first, fr);
1056                fa = getAccess(s->val.forincS.first->val.
                        varT.left->val.idV, fr);
1057                getVar(fa, tvar);
1058                //fprintf(f, "    xor @%d, @%d\n", tvar->
                        num, tvar->num);
1059                fprintf(f, "    movq $%d, @%d\n", s->val.
                        forincS.second->val.intconstT, tvar->
                        num);
1060                fprintf(f, "for%d:\n", forcount);
1061                fprintf(f, "    cmp $%d, @%d\n", s->val.
                        forincS.third->val.intconstT, tvar->num
                        );
1062                fprintf(f, "    jg forend%d\n", forcount);
1063                codeSTATE(s->val.forincS.fourth, fr);
1064                fprintf(f, "    addq $1, @%d\n", tvar->num)
                        ;
1065                fprintf(f, "    jmp for%d\n", forcount);
1066                fprintf(f, "forend%d:\n", forcount);
1067                forcount++;
1068                break;
1069            case fordecK:
1070                tvar = requestTemp();
1071                codeDECL_LIST(s->val.fordecS.first, fr);
1072                fa = getAccess(s->val.fordecS.first->val.
                        varT.left->val.idV, fr);
1073                getVar(fa, tvar);
1074                fprintf(f, "    xor @%d, @%d\n", tvar->num,
                         tvar->num);
1075                fprintf(f, "    movq $%d, @%d\n", s->val.
                        fordecS.second->val.intconstT, tvar->
                        num);
1076                fprintf(f, "for%d:\n", forcount);
1077                fprintf(f, "    cmp $%d, @%d\n", s->val.
                        fordecS.third->val.intconstT, tvar->num
                        );
1078                fprintf(f, "    jl forend%d\n", forcount);
1079                codeSTATE(s->val.fordecS.fourth, fr);
1080                fprintf(f, "    subq $1, @%d\n", tvar->num)
                        ;
```

```
1081                fprintf(f, "   jmp for%d\n", forcount);
1082                fprintf(f, "forend%d:\n", forcount);
1083                forcount++;
1084                break;
1085        }
1086  }
1087
1088  void allocOfLength(VAR *var, temp *tvar, temp *argument,
        frame *fr){
1089        Ty_ty *varty = getVarType(var, fr->c);
1090        while(varty->kind == Ty_name){
1091              varty = getSymbol(varty->u.name.table, varty->u
                    .name.name)->value;
1092        }
1093        fprintf(f, "   movq @%d, %%rax\n", argument->num);
1094        switch (varty->u.array->kind)
1095        {
1096        case Ty_int:
1097              fprintf(f, "   movq $0, %%rdi\n");
1098              break;
1099        case Ty_bool:
1100              fprintf(f, "   movq $0, %%rdi\n");
1101              break;
1102        case Ty_record:
1103              fprintf(f, "   movq $1, %%rdi\n");
1104              break;
1105        case Ty_array:
1106              fprintf(f, "   movq $1, %%rdi\n");
1107              break;
1108        default:
1109              break;
1110        }
1111
1112        fprintf(f, "   call allocL\n");
1113        fprintf(f, "   movq @%d, %%rdi\n", tvar->num);
1114        fprintf(f, "   movq %%rax, 8(%%rdi)\n");
1115  }
1116
1117  void allocate(temp *tvar, frame *fr, VAR *v){
1118        int recSize=0;
1119        int recLength=0;
1120        int recIndex=0;
1121        Ty_fieldList *list;
1122        Ty_ty *ty = getVarType(v, fr->c);
1123
1124
```

```
1125        allocateActualType:
1126        if(ty->kind == Ty_name){
1127              ty=getSymbol(ty->u.name.table, ty->u.name.name)
                      ->value;
1128              goto allocateActualType;
1129        }
1130        if(ty->kind == Ty_array){
1131              ty= ty->u.array;
1132              goto allocateActualType;
1133        }
1134
1135
1136        switch(ty->kind){
1137              case Ty_record:
1138                    recSize = 3;
1139                    list = ty->u.record;
1140                    while(list != NULL){
1141                          recSize = recSize +2;
1142                          recLength++;
1143                          list=list->tail;
1144                    }
1145                    list=ty->u.record;
1146                    fprintf(f, "    movq $%d, %%rax\n", recSize
                            );
1147                    fprintf(f, "    push @%d\n", tvar->num);
1148                    fprintf(f, "    call getMem\n");
1149                    fprintf(f, "    pop @%d\n", tvar->num);
1150                    fprintf(f, "    movq @%d, %%rcx\n", tvar->
                            num);
1151                    fprintf(f, "    movq %%rax, 8(%%rcx)\n");
1152                    fprintf(f, "    movq $0, (%%rax)\n");
1153                    fprintf(f, "    movq $%d, 8(%%rax)\n",
                            recSize);
1154                    fprintf(f, "    movq $%d, 16(%%rax)\n",
                            recLength);
1155                    while(list != NULL){
1156                          ty = list->head->ty;
1157                          while(ty->kind == Ty_name){
1158                                ty=getSymbol(ty->u.name.table,
                                        ty->u.name.name)->value;
1159                          }
1160                          switch(ty->kind){
1161                                case Ty_int:
1162                                      fprintf(f, "    movq $0, %d
                                            (%%rax)\n", (3+recIndex
                                            *2)*8 ); //header
```

```
1163                                    fprintf(f, "    movq $0, %d
                                            (%%rax)\n", (4+recIndex
                                            *2)*8 );
1164                                    break;
1165                                case Ty_bool:
1166                                    fprintf(f, "    movq $0, %d
                                            (%%rax)\n", (3+recIndex
                                            *2)*8); //header
1167                                    fprintf(f, "    movq $0, %d
                                            (%%rax)\n", (4+recIndex
                                            *2)*8 );
1168                                    break;
1169                                case Ty_record:
1170                                    fprintf(f, "    movq $1, %d
                                            (%%rax)\n", (3+recIndex
                                            *2)*8); //header
1171                                    fprintf(f, "    movq $0, %d
                                            (%%rax)\n", (4+recIndex
                                            *2)*8 );
1172                                    break;
1173                                case Ty_array:
1174                                    fprintf(f, "    movq $1, %d
                                            (%%rax)\n", (3+recIndex
                                            *2)*8); //header
1175                                    fprintf(f, "    movq $0, %d
                                            (%%rax)\n", (4+recIndex
                                            *2)*8 );
1176                                    break;
1177                                default:
1178                                    break;
1179                            }
1180                            recIndex++;
1181                            list = list->tail;
1182                        }
1183                        break;
1184                default:
1185                        break;
1186        }
1187
1188
1189  }
1190
1191
1192  //expects value to write in %rax
1193  void writer(){
1194        fprintf(f, "write:\n");
```

```
1195        fprintf(f, "    push %%rbp\n");
1196        fprintf(f, "    movq %%rsp, %%rbp\n");
1197        fprintf(f, "    push %%rax\n");
1198        printPushCaller();
1199        printPushCallee();
1200
1201        //wcount r15, write r14
1202        fprintf(f, "    movq $0, %%r15\n");
1203        fprintf(f, "    movq %%rax, %%r14\n");
1204        fprintf(f, "    push $10\n");
1205        fprintf(f, "    addq $1, %%r15\n");
1206        fprintf(f, "    cmp $0, %%r14\n");
1207        fprintf(f, "    jge positive\n");
1208
1209        fprintf(f, "    push $45\n");
1210        fprintf(f, "    movq $1, %%rax\n");
1211        fprintf(f, "    movq $1, %%rdi\n");
1212        fprintf(f, "    movq %%rsp, %%rsi\n");
1213        fprintf(f, "    movq $1, %%rdx\n");
1214        fprintf(f, "    syscall\n");
1215        fprintf(f, "    addq $8, %%rsp\n");
1216
1217        fprintf(f, "    movq %%r14, %%rsi\n");
1218        fprintf(f, "    neg %%rsi\n");
1219        fprintf(f, "    movq %%rsi, %%r14\n");
1220        fprintf(f, "    movq %%r14, %%rax\n");
1221        fprintf(f, "    jmp writeloop\n");
1222        fprintf(f, "positive:\n");
1223        fprintf(f, "    movq %%r14, %%rax\n");
1224        fprintf(f, "writeloop: \n");
1225        fprintf(f, "    movq $0, %%rdx\n");
1226        fprintf(f, "    movq $10, %%rcx\n");
1227        fprintf(f, "    idivq %%rcx\n");
1228        fprintf(f, "    addq $48, %%rdx\n");
1229        fprintf(f, "    push %%rdx\n");
1230        fprintf(f, "    addq $1, %%r15\n");
1231        fprintf(f, "    cmp $0, %%rax\n");
1232        fprintf(f, "    jne writeloop\n");
1233
1234        fprintf(f, "printloop:\n");
1235        fprintf(f, "    movq $1, %%rax\n");
1236        fprintf(f, "    movq $1, %%rdi\n");
1237        fprintf(f, "    movq %%rsp, %%rsi\n");
1238        fprintf(f, "    movq $1, %%rdx\n");
1239        fprintf(f, "    syscall\n");
1240        fprintf(f, "    addq $8, %%rsp\n");
```

```
1241        fprintf(f, "    addq $-1, %%r15\n");
1242        fprintf(f, "    cmp $0, %%r15\n");
1243        fprintf(f, "    jne printloop\n");
1244
1245
1246        printPopCallee();
1247        printPopCaller();
1248        fprintf(f, "    pop %%rax\n");
1249        fprintf(f, "    movq %%rbp, %%rsp\n");
1250        fprintf(f, "    pop %%rbp\n");
1251        fprintf(f, "    ret\n");
1252    }
1253
1254    //expects length in %rax, value descriptor in %%rdi, and
          returns pointer to memory in rax
1255    void allocLength(){
1256        fprintf(f, "allocL:\n");
1257        fprintf(f, "    movq %%rax, %%rsi\n");
1258        fprintf(f, "    movq $2, %%rcx\n");
1259        fprintf(f, "    imulq %%rcx\n");
1260        fprintf(f, "    addq $3, %%rax\n");
1261        fprintf(f, "    movq %%rax, %%rcx\n");
1262        fprintf(f, "    push %%rdi\n");
1263        fprintf(f, "    call getMem\n");
1264        fprintf(f, "    pop %%rdi\n");
1265        fprintf(f, "    movq $0, (%%rax)\n");
1266        fprintf(f, "    movq %%rcx, 8(%%rax)\n");
1267        fprintf(f, "    movq %%rsi, 16(%%rax)\n");
1268        fprintf(f, "    movq %%rax, %%rcx\n");
1269        fprintf(f, "    addq $24, %%rcx\n");
1270        fprintf(f, "allocLloop:\n");
1271        fprintf(f, "    cmp $0, %%rsi\n");
1272        fprintf(f, "    je allocLloopEnd\n");
1273        fprintf(f, "    movq %%rdi, (%%rcx)\n");
1274        fprintf(f, "    movq $0, 8(%%rcx)\n");
1275        fprintf(f, "    addq $16, %%rcx\n");
1276        fprintf(f, "    subq $1, %%rsi\n");
1277        fprintf(f, "    jmp allocLloop\n");
1278        fprintf(f, "allocLloopEnd:\n");
1279        fprintf(f, "    ret\n");
1280    }
```

A.24 registers.h

```
1  #ifndef __registers_h
2  #define __registers_h
```

```
3
4  typedef struct livenessNode{
5      int id;
6      char line[100];
7      struct parsedLine *pline;
8      struct stringBuffer *use;
9      struct stringBuffer *def;
10     struct stringBuffer *out;
11     struct stringBuffer *in;
12     struct nodeList *succ;
13     struct nodeList *pred;
14     char jumpto[100];
15     enum {movq, addq, subq, imulq, idivq, cmp, push, pop,
           jump, label, XOR, AND, OR, neg} op;
16 } livenessNode;
17
18 typedef struct stringBuffer{
19     char buffer[100];
20     struct stringBuffer *next;
21     struct stringBuffer *prev;
22 } stringBuffer;
23
24 typedef struct nodeList{
25     char label[100];
26     struct livenessNode *node;
27     struct nodeList *next;
28     struct nodeList *prev;
29 } nodeList;
30
31 typedef struct parsedLine{
32     char line[100];
33     struct parsedLine *next;
34 } parsedLine;
35 livenessNode *initNode(char *buffer);
36 nodeList *initnodeList();
37 stringBuffer *initstringBuffer();
38 parsedLine *initparsedLine();
39 nodeList *scanFile(char *file);
40
41 char *getString(char *str);
42 parsedLine *parseLine(livenessNode *node);
43 void operatorhandler(livenessNode *node);
44 int operator(livenessNode *node);
45 void printer(livenessNode *node);
46 int getEnum(char *line);
47 #endif
```

A.25 register.c

```
1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "registers.h"
5  #include "symbol.h"
6  #include <string.h>
7  #include "memory.h"
8  int nodeCount = 0;
9  nodeList *labels;
10
11 nodeList *scanFile(char *file){
12     FILE *fp;
13     nodeList *ret = NEW(nodeList);
14     char * buffer = NULL;
15     size_t len = 0;
16     size_t read;
17     fp = fopen(file, "r");
18
19     if (fp == NULL){
20         fprintf(stderr,"Could not open file %s",file);
21         exit(1);
22     }
23     livenessNode *temp;
24     livenessNode *node;
25     livenessNode *root;
26     while ((read = getline(&buffer, &len, fp)) != -1) {
27
28         // 1. making the node //
29         ////////////////////////
30         node = initNode(buffer);
31         if(nodeCount == 1){
32             node->pred = NULL;
33             root = node;
34         }else{
35             temp->succ->node = node;
36             node->pred->node = temp;
37         }
38         ////////////////////////
39
40         // 2. parsing the line //
41         ////////////////////////
42         node->pline = parseLine(node);
43         ////////////////////////
44
```

```
45          // 3. get operand enum //
46          //////////////////////
47          node->op = operator(node);
48          //////////////////////
49
50          // 4. argument handler //
51          //////////////////////
52          operatorhandler(node);
53          //////////////////////
54          temp = node;
55      }
56      ret->node = root;
57      ret->next = labels;
58      return ret;
59  }
60
61
62
63  // 1. ////////////////////////////
64  livenessNode *initNode(char *buffer){
65      livenessNode *n = NEW(livenessNode);
66      n->id = ++nodeCount;
67      strcpy(n->line, buffer);
68      n->pline = initparsedLine();
69      n->use = NULL;
70      n->def = NULL;
71      n->out = NULL;
72      n->in = NULL;
73      n->succ = initnodeList();
74      n->pred = initnodeList();
75      return n;
76  }
77
78  nodeList *initnodeList(){
79      nodeList *nl = NEW(nodeList);
80      nl->next = NULL;
81      nl->node = NULL;
82      nl->prev = NULL;
83      return nl;
84  }
85
86  stringBuffer *initstringBuffer(){
87      stringBuffer *sb = NEW(stringBuffer);
88      sb->next = NULL;
89      sb->prev = NULL;
90      return sb;
```

```
91  }
92  //////////////////////////////////
93
94
95
96  // 2. ////////////////////////////
97  parsedLine *parseLine(livenessNode *node){
98      char *buffer;
99      char tmpstr[100];
100     parsedLine *root;
101     parsedLine *temp;
102     int count = 0;
103
104     strcpy(tmpstr, node->line);
105     buffer = strtok(tmpstr, " ");
106
107     parsedLine *str = NEW(parsedLine);
108     while (buffer) {
109         str = initparsedLine();
110         if(count == 0){
111             root = str;
112             strcpy(str->line, buffer);
113         }else{
114             temp->next = str;
115             strncpy(str->line, buffer, strlen(buffer)-1);
116         }
117         temp = str;
118         count++;
119         buffer = strtok (NULL, " ");
120     }
121     return root;
122 }
123
124 parsedLine *initparsedLine(){
125     parsedLine *pl = NEW(parsedLine);
126     pl->next = NULL;
127     return pl;
128 }
129 //////////////////////////////////
130
131
132 // 3. ////////////////////////////
133 int operator(livenessNode *node){
134     parsedLine *pl = node->pline;
135     char *op;
136     if(pl->next == NULL && pl->line[(strlen(pl->line)-2)]
```

```
            == ':'){
137             op = "label";
138         }else if((pl->line[0] == 'j') || (strncmp(pl->line, "
                ca", 2)) == 0){
139             op = "jump";
140         }else{
141             op = pl->line;
142         }
143         return getEnum(op);
144     }
145
146     int getEnum(char *line){
147
148         if(strcmp(line, "movq") == 0){
149             return 0;
150         }else if(strcmp(line, "addq") == 0){
151             return 1;
152         }else if(strcmp(line, "subq") == 0){
153             return 2;
154         }else if(strcmp(line, "imulq") == 0){
155             return 3;
156         }else if(strcmp(line, "idivq") == 0){
157             return 4;
158         }else if(strcmp(line, "cmp") == 0){
159             return 5;
160         }else if(strcmp(line, "push") == 0){
161             return 6;
162         }else if(strcmp(line, "pop") == 0){
163             return 7;
164         }else if(strcmp(line, "jump") == 0){
165             return 8;
166         }else if(strcmp(line, "label") == 0){
167             return 9;
168         }else if(strcmp(line, "XOR") == 0){
169             return 10;
170         }else if(strcmp(line, "AND") == 0){
171             return 11;
172         }else if(strcmp(line, "OR") == 0){
173             return 12;
174         }else if(strcmp(line, "neg") == 0){
175             return 13;
176         }else{
177             return 14;
178         }
179     }
180     ////////////////////////////////////
```

```
181
182
183
184 // 4. ////////////////////////////
185 char *getString(char *str){
186     char *ret = malloc(sizeof(char)*100);
187     int retpos = 0;
188     for (unsigned int i = 0; i < strlen(str); i++){
189         if( str[i] == '(' && str[i+1] == '@'){
190             while(str[i] != ')'){
191                 i++;
192                 ret[retpos] = str[i];
193                 retpos++;
194             }
195             ret[retpos] = '\0';
196         }
197     }
198     if(strncmp(ret, "@", 1) != 0){
199         strcpy(ret, str);
200     }
201     return ret;
202 }
203
204 void operatorhandler(livenessNode *node){//REFACTOR!!!
205
206     parsedLine *line = node->pline;
207     char first[30];
208     char second[30];
209     nodeList *tmp;
210     stringBuffer *s;
211
212     switch (node->op)
213     {
214     case movq:
215
216         strcpy(first, getString(line->next->line));
217         strcpy(second, getString(line->next->next->line))
                ;
218
219         if(strncmp(first, "@", 1) == 0){
220             node->use = initstringBuffer();
221             strcpy(node->use->buffer, first);
222         }if(strncmp(second, "@", 1) == 0){
223             node->def = initstringBuffer();
224             strcpy(node->def->buffer, second);
225         }
```

```
226             break;
227
228         case addq:
229             strcpy(first, getString(line->next->line));
230             strcpy(second, getString(line->next->next->line))
                    ;
231             if(strncmp(first, "@", 1) == 0){
232                 s = initstringBuffer();
233                 s->next=node->use;
234                 node->use=s;
235                 strcpy(node->use->buffer, first);
236             }if(strncmp(second, "@", 1) == 0){
237                 if(strncmp(first, "@", 1) == 0){
238                     node->use->next = initstringBuffer();
239                     strcpy(node->use->next->buffer, second);
240                 }else{
241                     node->use = initstringBuffer();
242                     strcpy(node->use->buffer, second);
243                 }
244             }
245             break;
246
247         case subq:
248             strcpy(first, getString(line->next->line));
249             strcpy(second, getString(line->next->next->line))
                    ;
250             if(strncmp(first, "@", 1) == 0){
251                 s = initstringBuffer();
252                 s->next=node->use;
253                 node->use=s;
254                 strcpy(node->use->buffer, first);
255             }if(strncmp(second, "@", 1) == 0){
256                 if(strncmp(first, "@", 1) == 0){
257                     node->use->next = initstringBuffer();
258                     strcpy(node->use->next->buffer, second);
259                 }else{
260                     node->use = initstringBuffer();
261                     strcpy(node->use->buffer, second);
262                 }
263             }
264             break;
265
266         case imulq:
267             strcpy(first, getString(line->next->line));
268             if(strncmp(first, "@", 1) == 0){
269                 s = initstringBuffer();
```

```
270             s->next=node->use;
271             node->use=s;
272             strcpy(node->use->buffer, first);
273         }
274         if(line->next->next != NULL){
275             strcpy(second, getString(line->next->next->
                    line));
276             if(strncmp(second, "@", 1) == 0){
277                 if(strncmp(first, "@", 1) == 0){
278                     node->use->next = initstringBuffer();
279                     strcpy(node->use->next->buffer,
                            second);
280                 }else{
281                     node->use = initstringBuffer();
282                     strcpy(node->use->buffer, second);
283             }
284         }
285         }
286         break;
287
288     case idivq:
289         strcpy(first, getString(line->next->line));
290         if(strncmp(first, "@", 1) == 0){
291             node->use = initstringBuffer();
292             strcpy(node->use->buffer, first);
293         }
294         break;
295
296     case cmp:
297         strcpy(first, getString(line->next->line));
298         strcpy(second, getString(line->next->next->line))
                ;
299         if(strncmp(first, "@", 1) == 0){
300             s = initstringBuffer();
301             s->next=node->use;
302             node->use=s;
303             strcpy(node->use->buffer, first);
304         }if(strncmp(second, "@", 1) == 0){
305             if(strncmp(first, "@", 1) == 0){
306                 node->use->next = initstringBuffer();
307                 strcpy(node->use->next->buffer, second);
308             }else{
309                 node->use = initstringBuffer();
310                 strcpy(node->use->buffer, second);
311             }
312         }
```

```
313             break;
314
315         case push:
316             strcpy(first, getString(line->next->line));
317             if(strncmp(first, "@", 1) == 0){
318                 node->use = initstringBuffer();
319                 strcpy(node->use->buffer, first);
320             }
321             break;
322
323         case pop:
324             strcpy(first, getString(line->next->line));
325             if(strncmp(first, "@", 1) == 0){
326                 node->def = initstringBuffer();
327                 strcpy(node->def->buffer, first);
328             }
329             break;
330
331         case jump:
332             strcpy(first, getString(line->next->line));
333             strcpy(node->jumpto, first);
334             break;
335
336         case label:
337             tmp = initnodeList();
338             tmp->node = node;
339             tmp->next = labels;
340             strncpy(tmp->label, line->line, strlen(line->line
                    ) -1);
341             labels = tmp;
342             break;
343
344         case XOR:
345             strcpy(first, getString(line->next->line));
346             strcpy(second, getString(line->next->next->line))
                    ;
347             if(strncmp(first, "@", 1) == 0){
348                 s = initstringBuffer();
349                 s->next=node->use;
350                 node->use=s;
351                 strcpy(node->use->buffer, first);
352             }if(strncmp(second, "@", 1) == 0){
353                 if(strncmp(first, "@", 1) == 0){
354                     node->use->next = initstringBuffer();
355                     strcpy(node->use->next->buffer, second);
356                 }else{
```

```
357                 node->use = initstringBuffer();
358                 strcpy(node->use->buffer, second);
359             }
360         }
361         break;
362 
363     case AND:
364         strcpy(first, getString(line->next->line));
365         strcpy(second, getString(line->next->next->line))
                ;
366         if(strncmp(first, "@", 1) == 0){
367             s = initstringBuffer();
368             s->next=node->use;
369             node->use=s;
370             strcpy(node->use->buffer, first);
371         }if(strncmp(second, "@", 1) == 0){
372             if(strncmp(first, "@", 1) == 0){
373                 node->use->next = initstringBuffer();
374                 strcpy(node->use->next->buffer, second);
375             }else{
376                 node->use = initstringBuffer();
377                 strcpy(node->use->buffer, second);
378             }
379         }
380         break;
381 
382     case OR:
383         strcpy(first, getString(line->next->line));
384         strcpy(second, getString(line->next->next->line))
                ;
385         if(strncmp(first, "@", 1) == 0){
386             s = initstringBuffer();
387             s->next=node->use;
388             node->use=s;
389             strcpy(node->use->buffer, first);
390         }if(strncmp(second, "@", 1) == 0){
391             if(strncmp(first, "@", 1) == 0){
392                 node->use->next = initstringBuffer();
393                 strcpy(node->use->next->buffer, second);
394             }else{
395                 node->use = initstringBuffer();
396                 strcpy(node->use->buffer, second);
397             }
398         }
399         break;
400 
```

```
401         case neg:
402             strcpy(first, getString(line->next->line));
403             if(strncmp(first, "@", 1) == 0){
404                 node->use = initstringBuffer();
405                 strcpy(node->use->buffer, first);
406             }
407             break;
408 
409         default:
410             break;
411         }
412     }
413     //////////////////////////////////
414 
415     // Extra for debugging //////////////
416     void printer(livenessNode *node){
417         fprintf(stderr,"node->id = %d\n", node->id);
418         fprintf(stderr,"node->line = %s", node->line);
419         parsedLine *tmp = node->pline;
420         while(tmp->line != NULL){
421             fprintf(stderr,"node->pline = %s\n", tmp->line);
422             tmp = tmp->next;
423         }
424         if(node->use != NULL){
425             fprintf(stderr,"node->use = %s", node->use->
                    buffer);
426             if(node->use->next != NULL){
427                 fprintf(stderr," %s", node->use->next->buffer
                        );
428             }fprintf(stderr,"\n");
429         }if(node->def != NULL){
430             fprintf(stderr,"node->def = %s\n", node->def->
                    buffer);
431         }
432         fprintf(stderr,"node->jumpto = %s\n", node->jumpto);
433         fprintf(stderr,"node->op = %d\n", node->op);
434 
435         fprintf(stderr,"\n");
436     }
437     //////////////////////////////////
```

A.26 line.h

```
1   #include "TEMP.h"
2   
3   
```

```
4  typedef struct line{
5      enum{other, movq, operation, label, root, jump} kind;
6      char *operand;
7      linkedlist *useargs;
8      linkedlist *defargs;
9      line *next;
10 } line;
11
12 typedef struct reg{
13     char *name;
14 } reg;
15
16 typedef struct arg_{
17     enum {tempo, regi, lit} regkind;
18     enum {notmem, mem} memkind;
19     union{
20         struct temp *tmp;
21         struct reg *reg;
22         char *literal;
23     } u;
24     int memoffset;
25 } arg;
26
27
28 #define makeArg(...) OVERLOAD(makeArg, (__VA_ARGS__), \
29     (print_ii, (int, int)), \
30     (print_id, (int, double)), \
31     (print_di, (double, int)), \
32     (print_dd, (double, double)), \
33     (print_iii, (int, int, int)) \
34 )
35
36 line *makeLine(int type, char *op, arg *src, arg *dst,
       int args, int srckind, int dstkind);
37
38 arg *makeTempArg(int ismem, temp *tem, int offset);
39
40 arg *makeRegArg(int ismem, reg *r, int offset);
41
42 arg *makeLitArg(int ismem, char *name, int offset);
43
44 reg *makeReg(char *id);
```

A.27 line.c

```
1  #include "line.h"
```

```
2  #include "memory.h"
3  #include "stdio.h"
4
5  line *currentline;
6
7  void printLines(line *root){
8      line *lin = root;
9      while(lin != NULL){
10         printline(lin);
11         lin = lin->next;
12     }
13 }
14
15 void printline(line *line){
16     switch(line->kind){
17         case other:
18             fprintf(stderr, line->operand);
19     }
20
21     fprintf(stderr, "\n");
22 }
23
24 line *makeLine(int type, char *op, arg *src, arg *dst,
       int args, int srckind, int dstkind){
25     line *newline = NEW(line);
26     newline->args=args;
27     newline->dest=dst;
28     newline->kind=type;
29     newline->operand=op;
30     newline->source=src;
31     newline->skind=srckind;
32     newline->dkind=dstkind;
33
34     if(currentline == NULL){
35         currentline = newline;
36     } else {
37         currentline->next=newline;
38         currentline=newline;
39     }
40
41     return newline;
42 }
43
44 arg *makeTempArg(int ismem, temp *tem, int offset){
45     arg *a=NEW(arg);
46     a->regkind=tempo;
```

```
47      a->memkind=ismem;
48      a->memoffset=offset;
49      a->u.tmp=tem;
50      return a;
51  }
52
53  arg *makeRegArg(int ismem, reg *r, int offset){
54      arg *a=NEW(arg);
55      a->regkind=regi;
56      a->memkind=ismem;
57      a->memoffset=offset;
58      a->u.reg=r;
59      return a;
60  }
61
62  arg *makeLitArg(int ismem, char *name, int offset){
63      arg *a=NEW(arg);
64      a->regkind=lit;
65      a->memkind=ismem;
66      a->memoffset=offset;
67      a->u.literal==name;
68      return a;
69  }
70
71  reg *makeReg(char *id){
72      reg *r = NEW(reg);
73      r->name=id;
74  }
```

A.28 liveness.h

```
1   #ifndef __liveness_h
2   #define __liveness_h
3   #include "registers.h"
4   #include<stdbool.h>
5
6
7   livenessNode *appendLabels();
8   void jumpCheck(livenessNode *node, nodeList *labels);
9   livenessNode *livenessAnalysis();
10  stringBuffer *in(livenessNode *node);
11  stringBuffer *out(livenessNode *node);
12  void insert(stringBuffer** head_ref, char *new_data);
13  bool isPresent(stringBuffer *head, char *data);
14  stringBuffer *getUnion(stringBuffer *head1, stringBuffer
        *head2);
```

```
15 stringBuffer *getSub(stringBuffer *head1, stringBuffer *
       head2);
16
17 #endif
```

A.29 liveness.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <stdbool.h>
 4 #include "registers.h"
 5 #include "symbol.h"
 6 #include <string.h>
 7 #include "memory.h"
 8 #include "liveness.h"
 9
10 livenessNode *appendLabels(){
11     livenessNode *node;
12     nodeList *labels;
13     nodeList *ret = scanFile("./output.asm");
14     node = ret->node;
15     labels = ret->next;
16     do{
17         node = node->succ->node;
18         if(node->op == 8){
19             jumpCheck(node, labels);
20         }
21     }while (node->succ->node != NULL);
22     return node;
23 }
24
25 void jumpCheck(livenessNode *node, nodeList *labels){
26     nodeList *newSucc;
27     while(labels != NULL){
28         if(strncmp(node->jumpto, labels->label, strlen(
               node->jumpto)) == 0){
29             newSucc = initnodeList();
30             node->succ->next = newSucc;
31             newSucc->node = labels->node;
32         }
33         if(labels->next == NULL){
34             break;
35         }else{
36             labels = labels->next;
37         }
38     }
```

```
39 }
40
41 livenessNode *livenessAnalysis(){
42     livenessNode *node = appendLabels();
43     livenessNode *root = node;
44     livenessNode *top;
45     stringBuffer *intmp;
46     stringBuffer *outtmp;
47     stringBuffer *incheck;
48     stringBuffer *outcheck;
49     int check = 0;
50     int firstloop = 0;
51     int iteration = 1;
52     do{
53         check = 0;
54         for (int i = node->id; i > 0; i--){
55             if(node->pred != NULL){
56                 node = node->pred->node;
57             }
58             intmp = node->in;
59             outtmp = node->out;
60             node->in = in(node);
61             node->out = out(node);
62             incheck = node->in;
63             outcheck = node->out;
64             fprintf(stderr,"line no: %d\n", node->id);
65             if(intmp->buffer != NULL){
66                 fprintf(stderr," -intmp: %s", intmp->
                       buffer);
67                 if(intmp->next != NULL){
68                     fprintf(stderr," %s", intmp->next->
                           buffer);
69                 }fprintf(stderr,"\n");
70             }
71             if(incheck->buffer != NULL){
72                 fprintf(stderr," -incheck: %s", incheck->
                       buffer);
73                 if(incheck->next != NULL){
74                     fprintf(stderr," %s", incheck->next->
                           buffer);
75                 }fprintf(stderr,"\n");
76             }if(outtmp->buffer != NULL){
77                 fprintf(stderr," -outtmp: %s", outtmp->
                       buffer);
78                 if(outtmp->next != NULL){
79                     fprintf(stderr," %s", outtmp->next->
```

```
                buffer);
80              }fprintf(stderr,"\n");
81          }
82          if(outcheck->buffer != NULL){
83              fprintf(stderr," -outcheck: %s", outcheck
                    ->buffer);
84              if(outcheck->next != NULL){
85                  fprintf(stderr," %s", outcheck->next
                        ->buffer);
86              }fprintf(stderr,"\n");
87          }
88          if(intmp->buffer != NULL && incheck->buffer
                != NULL){
89              //fprintf(stderr," -intmp: %s\n", intmp->
                    buffer);
90              //fprintf(stderr," -incheck: %s\n",
                    incheck->buffer);
91              while((intmp != NULL) && (incheck != NULL
                    )){
92                  if(strcmp(intmp->buffer, incheck->
                        buffer) != 0){
93                      check = 1;
94                  }
95                  intmp = intmp->next;
96                  incheck = incheck->next;
97                  if(intmp == NULL ^ incheck == NULL){
98                      check = 1;
99                  }
100             }
101         } else if(intmp->buffer != NULL ^ incheck->
                buffer != NULL){
102             check = 1;
103         }
104         if(outtmp->buffer != NULL && outcheck->buffer
                != NULL){
105             //fprintf(stderr," -outtmp: %s\n", outtmp
                    ->buffer);
106             //fprintf(stderr," -outcheck: %s\n",
                    outcheck->buffer);
107             while(outtmp != NULL && outcheck != NULL)
                    {
108                 if(strcmp(outtmp->buffer, outcheck->
                        buffer) != 0){
109                     check = 1;
110                 }
111                 outtmp = outtmp->next;
```

```
112                     outcheck = outcheck->next;
113                     if(outtmp == NULL ^ outcheck == NULL)
                            {
114                         check = 1;
115                     }
116                 }
117             } else if(outtmp->buffer != NULL ^ outcheck->
                    buffer != NULL){
118                 check = 1;
119             }
120         }
121         if(firstloop == 0){
122             top = node;
123             check = 1;
124             firstloop = 1;
125         }fprintf(stderr,"iteration: %d\n", iteration++);
126         node = root;
127         fprintf(stderr,"check print: %d\n", check);
128     }while(check != 0);
129     return top;
130 }
131
132 stringBuffer *in(livenessNode *node){
133     stringBuffer *ret = NEW(stringBuffer);
134     stringBuffer *right = NEW(stringBuffer);
135     right = getSub(node->out, node->def);
136     ret = getUnion(node->use, right);
137     return ret;
138 }
139 stringBuffer *out(livenessNode *node){
140     stringBuffer *ret = NEW(stringBuffer);
141     stringBuffer *tmp = NEW(stringBuffer);
142     nodeList *succ = node->succ;
143     livenessNode *succNode = succ->node;
144     ret = succNode->in;
145     succ = succ->next;
146     if(succ != NULL && succ->node != NULL){
147         succNode = succ->node;
148         tmp = succNode->in;
149         ret = getUnion(ret, tmp);
150     }
151     return ret;
152 }
153
154 stringBuffer *getUnion(stringBuffer *head1, stringBuffer
      *head2)
```

```
155 {
156  stringBuffer *result = NULL;
157  stringBuffer *t1 = head1, *t2 = head2;
158
159  while (t1 != NULL) {
160   insert(&result, t1->buffer);
161   t1 = t1->next;
162  }
163
164  while (t2 != NULL) {
165   if (!isPresent(result, t2->buffer))
166    insert(&result, t2->buffer);
167   t2 = t2->next;
168  }
169
170  return result;
171 }
172 stringBuffer *getSub(stringBuffer *head1, stringBuffer *
      head2)
173 {
174  stringBuffer *result = NULL;
175  stringBuffer *t1 = head1, *t2 = head2;
176
177  while (t1 != NULL) {
178         if (!isPresent(t2, t1->buffer))
179    insert(&result, t1->buffer);
180   t1 = t1->next;
181  }
182
183  return result;
184 }
185
186 void insert(stringBuffer** head_ref, char *new_data)
187 {
188  struct stringBuffer *new_node = (struct stringBuffer*)
       malloc(sizeof(struct stringBuffer));
189      strcpy(new_node->buffer, new_data);
190
191  /* link the old list off the new node */
192  new_node->next = (*head_ref);
193
194  /* move the head to point to the new node */
195  (*head_ref) = new_node;
196 }
197
198 /* A utility function that returns true if data is
```

```
199 present in linked list else return false */
200 bool isPresent (stringBuffer *head, char *data)
201 {
202  stringBuffer *t = head;
203  while (t != NULL)
204  {
205   if (strcmp(t->buffer, data) == 0)
206    return 1;
207   t = t->next;
208  }
209  return 0;
210 }
```

A.30 graphcolor.h

```
 1 #ifndef __graphcolor_h
 2 #define __graphcolor_h
 3 #include "registers.h"
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <stdbool.h>
 7 #include "memory.h"
 8 #include "liveness.h"
 9 #include <string.h>
10
11 #define numOfColors 4
12
13 typedef struct edgelist_ edgelist;
14 typedef struct nodelist_ nodelist;
15 typedef struct node_ node;
16 typedef struct edge_ edge;
17 typedef struct stack_ stack;
18
19 typedef struct stack_{
20     node *object;
21     stack *next;
22 };
23
24 typedef struct graph{
25     nodelist *nodes;
26     edgelist *edges;
27     stack *stack;
28 }graph;
29
30 typedef struct node_{
31     char *id;
```

```
32      int onStack;
33      int spill;
34      int spillnumber;
35      int color;
36      edgelist *neighbors;
37      int numOfNeighbors;
38      int degreelimit;
39  }node;
40
41  typedef struct nodelist_{
42      node *head;
43      nodelist *tail;
44  }nodelist;
45
46  typedef struct edge_{
47      node *from;
48      node *to;
49  }edge;
50
51  typedef struct edgelist_{
52      edge *head;
53      edgelist *tail;
54  }edgelist;
55
56
57  void buildgraph(livenessNode *node);
58  void addnode(char *id);
59  void addedge(char *id1, char *id2);
60  edgelist *createEdgelist(edge *head, edgelist *tail);
61  nodelist *createNodelist(node *head, nodelist *tail);
62  node *createNode(char *id);
63  edge *createEdge(char *id1, char *id2);
64  node *getnode(char *id);
65  void simplifyNode(node *n);
66  int simplify();
67  void pushnode(node *n);
68  void color();
69  void spillNode(node *n);
70  node *popNode();
71  int checkColor(int spilled, int color, node *n);
72  void registerAllocation();
73  void writefile(livenessNode *lnode);
74  void printcolor(node *n);
75
76  #endif
```

A.31 graphcolor.c

```
1  #include "graphcolor.h"
2
3
4  graph *g;
5  FILE *file;
6  int spillcount = 0;
7
8  void registerAllocation(){
9      livenessNode *node;
10     node = livenessAnalysis();
11     file = fopen("./final.s", "w");
12     buildgraph(node);
13     color();
14     while(true){
15         writefile(node);
16         if(node->succ->node == NULL){
17             break;
18         }else{
19             node = node->succ->node;
20         }
21     }
22 }
23
24 /*void pushlive(stringBuffer *in){
25     if(in == NULL){
26         return;
27     }
28     fprintf(file, "    push ");
29     printcolor(getnode(in->buffer));
30     fprintf(file, "\n");
31     pushlive(in->next);
32 }
33
34 void poplive(stringBuffer *out){
35     if(out == NULL){
36         return;
37     }
38     poplive(out->next);
39     fprintf(file, "    pop ");
40     printcolor(getnode(out->buffer));
41     fprintf(file, "\n");
42 }*/
43
44 void writefile(livenessNode *lnode){
```

```
45      char* temp = malloc(sizeof(char)*10);
46      node *n;
47      int index = 0;
48      /*livenessNode *call;
49      if(strcmp(lnode->pline->line, "call") == 0){
50          pushlive(lnode->in);
51          fprintf(file, "%s", lnode->line);
52          poplive(lnode->out);
53          return;
54      }*/
55
56      //currently xor does not use or define
57
58      if((lnode->def == NULL) && (lnode->use == NULL) &&
            strcmp(lnode->pline, "xor") != 0){
59          if(strcmp(lnode->pline, "xor") == 0){
60              fprintf(stderr, "xor\n");
61          }
62          fprintf(stdout, "%s", lnode->line);
63          if(strcmp(lnode->line, ".section .data\n") == 0){
64              fprintf(stdout, "    spilling: .space %d\n",
                    spillcount*8);
65          }
66          return;
67      }
68      char *line = lnode->line;
69      int unsigned i = 0;
70      while(i < strlen(line)){
71          if(line[i] == '@'){
72              temp[index] = line[i];
73              index++;
74              i++;
75              while((47 < (int) line[i]) && ( (int) line[i]
                        < 58)){
76                  temp[index] = line[i];
77                  index++;
78                  i++;
79              }
80              temp[index] = '\0';
81              fprintf(stderr, "%s\n", temp);
82              n = getnode(temp);
83              printcolor(n);
84              index = 0;
85              free(temp);
86              temp = malloc(sizeof(char) * 10);
87          }else{
```

```
88              fprintf(stdout, "%c", line[i]);
89              i++;
90          }
91      }
92  }
93
94  void printcolor(node *n){
95      if(n->spill == 0){
96          switch(n->color)
97          {
98              case 1:
99                  fprintf(stdout, "%%r15");
100                 break;
101             case 2:
102                 fprintf(stdout, "%%r14");
103                 break;
104             case 3:
105                 fprintf(stdout, "%%r13");
106                 break;
107             case 4:
108                 fprintf(stdout, "%%r12");
109                 break;
110             case 5:
111                 fprintf(stdout, "%%r11");
112                 break;
113             case 6:
114                 fprintf(stdout, "%%r10");
115                 break;
116         }
117     } else if(n->spill == 1){
118         if(n->spillnumber == 0){
119             fprintf(stdout, "(spilling)");
120         } else {
121             fprintf(stdout, "(spilling+%d)", n->
                    spillnumber*8);
122         }
123
124     }
125 }
126
127 void pushnode(node *n){
128     stack *stck = NEW(stack);
129     stck->object=n;
130     stck->next = g->stack;
131     g->stack = stck;
132 }
```

```
133
134 node *popNode(){
135     node *n = g->stack->object;
136     g->stack = g->stack->next;
137     return n;
138 }
139
140 void spillNode(node *n){
141     n->spill = 1;
142     n->spillnumber = spillcount;
143     spillcount++;
144     simplifyNode(n);
145 }
146
147 int simplify(){
148     nodelist *nodes = g->nodes;
149     while(nodes != NULL){
150         if((nodes->head->degreelimit > nodes->head->
                numOfNeighbors) && (nodes->head->onStack == 0)
                ){
151             simplifyNode(nodes->head);
152             return 0;
153         }
154         nodes = nodes->tail;
155     }
156     nodes = g->nodes;
157     while(nodes != NULL){
158         if(nodes->head->onStack != 1){
159             spillNode(nodes->head);
160             return 0;
161         }
162         nodes = nodes->tail;
163     }
164     return 1;
165 }
166
167 void simplifyNode(node *n){
168     edgelist *neighbors = n->neighbors;
169     pushnode(n);
170     n->onStack = 1;
171     while(neighbors != NULL){
172         neighbors->head->to->numOfNeighbors = neighbors->
              head->to->numOfNeighbors - 1;
173         //neighbors->head->to->degreelimit = neighbors->
              head->to->degreelimit - 1;
174         neighbors = neighbors->tail;
```

```
175     }
176 }
177
178 int checkColor(int spilled, int color, node *n){
179     edgelist *neighbors = n->neighbors;
180     while (neighbors != NULL){
181         if( (spilled == neighbors->head->to->spill) && (
                color == neighbors->head->to->color)){
182             return 0;
183         }
184         neighbors = neighbors->tail;
185     }
186     return 1;
187 }
188
189 void color(){
190     stack *s;
191     node *n;
192     int simplified = 0;
193     int colored;
194     nodelist *nl = g->nodes;
195     int spillcolor=0;
196     while(simplified == 0){
197         simplified = simplify();
198     }
199
200     s = g->stack;
201     while(s != NULL){
202         n = s->object;
203         //fprintf(stderr, "%s : %d\n", n->id, n->spill);
204         s = s->next;
205     }
206
207     while(g->stack != NULL){
208         n = popNode();
209         if(n->spill == 0){
210             for(int i = 1; i <= numOfColors; i++){
211                 colored = checkColor(0, i, n);
212                 if(colored == 1){
213                     n->color=i;
214                     break;
215                 }
216             }
217         } else if(n->spill == 1){
218             spillcolor = 0;
219             while(true){
```

```
220                 colored = checkColor(1, spillcolor, n);
221                 if(colored == 1){
222                     n->color = spillcolor;
223                     break;
224                 }
225                 spillcolor++;
226             }
227         }
228     }
229
230     nl = g->nodes;
231     while(nl != NULL){
232         fprintf(stderr, "id: %s, spill: %d, color: %d\n",
                nl->head->id, nl->head->spill, nl->head->
            color);
233         nl = nl->tail;
234     }
235 }
236
237 void addNeighbors(stringBuffer *interference){
238     if (interference == NULL){
239         return;
240     }
241     stringBuffer *index = interference;
242     stringBuffer *remaining = interference->next;
243     if (interference == NULL){
244         return;
245     }
246
247     while(index != NULL){
248         addnode(index->buffer);
249         index = index->next;
250     }
251     index = interference;
252
253     while(index != NULL){
254         remaining = index->next;
255         while(remaining != NULL){
256             addedge(index->buffer, remaining->buffer);
257             addedge(remaining->buffer, index->buffer);
258             remaining=remaining->next;
259         }
260         index = index->next;
261     }
262
263 }
```

```
264
265 void buildgraph(livenessNode *node){
266     nodelist *n;
267     edgelist *e;
268     g = NEW(graph);
269     g->edges = NULL;
270     g->stack = NULL;
271     g->nodes = NULL;
272     while(true){
273         addNeighbors(node->in);
274         addNeighbors(node->out);
275         if(node->succ->node == NULL){
276             break;
277         }else{
278             node = node->succ->node;
279         }
280     }
281
282
283
284     n = g->nodes;
285     while(n != NULL){
286         e = n->head->neighbors;
287         fprintf(stderr, "%s with edges", n->head->id);
288         while (e != NULL){
289             fprintf(stderr, " %s -> %s,", e->head->from->
                    id, e->head->to->id);
290             e = e->tail;
291         }
292         fprintf(stderr, "\n");
293         n=n->tail;
294     }
295 }
296
297 node *createNode(char *id){
298     node *n = NEW(node);
299     n->id = id;
300     n->numOfNeighbors=0;
301     n->onStack=0;
302     n->spill=0;
303     n->color=0;
304     n->spillnumber=0;
305     n->degreelimit=numOfColors;
306     n->neighbors=NULL;
307     return n;
308 }
```

```
309
310 nodelist *createNodelist(node *head, nodelist *tail){
311     nodelist *nl = NEW(nodelist);
312     nl->head=head;
313     nl->tail=tail;
314     return nl;
315 }
316
317 void addnode(char *id){
318     nodelist *d = g->nodes;
319     node *n;
320     if(d != NULL){
321         while(d != NULL){
322             if(strcmp(d->head->id, id) == 0){
323                 return;
324             }
325             d = d->tail;
326         }
327     }
328     n = createNode(id);
329     g->nodes=createNodelist(n, g->nodes);
330 }
331
332 node *getnode(char *id){
333     nodelist *nl = g->nodes;
334     while(nl != NULL){
335         if(strcmp(nl->head->id, id) == 0){
336             return nl->head;
337         }
338         nl = nl->tail;
339     }
340     //fprintf(stderr, "No such node: %s\n", id);
341     return NULL;
342 }
343
344 edge *createEdge(char *id1, char *id2){
345     edge *e = NEW(edge);
346     e->from = getnode(id1);
347     e->from->numOfNeighbors = e->from->numOfNeighbors+1;
348     e->to = getnode(id2);
349     return e;
350 }
351
352 edgelist *createEdgelist(edge *head, edgelist *tail){
353     edgelist *el = NEW(edgelist);
354     el->head = head;
```

```
355     el->tail = tail;
356     return el;
357 }
358
359 void addedge(char *id1, char *id2){
360     node *n = getnode(id1);
361     edge *e;
362     edgelist *el = n->neighbors;
363     while(el != NULL){
364         if (strcmp(el->head->to->id, id2) == 0){
365             return;
366         }
367         el = el->tail;
368     }
369     e = createEdge(id1, id2);
370     n->neighbors = createEdgelist(e, n->neighbors);
371 }
```

A.32 regallocation.h

```
1  #ifndef __regallocation_h
2  #define __regallocation_h
3  #include "registers.h"
4
5  typedef struct registers{
6      char reg[5];
7      char *temp;
8  } registers;
9
10 void printReg(livenessNode *node);
11
12 void regAllocation();
13 void initRegisters();
14 void requestRegister(char *str);
15 void freeRegs(livenessNode *node);
16 void unassignTemp(char *str);
17 void regCheck(livenessNode *node);
18 #endif
```

A.33 recallocation.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include "registers.h"
5  #include "symbol.h"
```

```
 6 #include <string.h>
 7 #include "memory.h"
 8 #include "liveness.h"
 9 #include "regallocation.h"
10 #include "graphcolor.h"
11
12 /*
13 currently only uses registers 12-15, to avoid syscalls
       overwriting contents. currently register might not be
       the same if temporary dies so code generation must be
       careful
14 might use graph coloring to assign specific register to
       temporary, then would be able to push and pop into
       same register.
15 */
16 registers regs[4];
17 FILE *file;
18
19 void regAllocation(){
20     livenessNode *node;
21     node = livenessAnalysis();
22     livenessNode *node1 = node;
23     initRegisters();
24     file = fopen("./final.s", "w");
25
26     while(true){
27         regCheck(node);
28         if(node->succ->node == NULL){
29             break;
30         }else{
31             node = node->succ->node;
32         }
33     }
34
35     close(file);
36
37
38     buildgraph(node1);
39     color();
40     writefile(node1);
41 }
42
43 void initRegisters(){
44     for(int i = 0; i < 4; i++){
45         sprintf(regs[i].reg, "%%r%d", i +12);
46         regs[i].temp = NULL;
```

```
47      }
48  }
49
50  void regCheck(livenessNode *node){
51      if(node->def != NULL){
52          requestRegister(node->def->buffer);
53      }
54      if(node->use != NULL){
55          requestRegister(node->use->buffer);
56          if(node->use->next != NULL){
57              requestRegister(node->use->next->buffer);
58          }
59      }
60      printReg(node);
61      freeRegs(node);
62  }
63
64  void printReg(livenessNode *node){
65      char* temp = malloc(sizeof(char)*10);
66      int index = 0;
67      if((node->def == NULL) && (node->use == NULL)){
68          fprintf(file, "%s", node->line);
69          return;
70      }
71      char *line = node->line;
72      int unsigned i = 0;
73      while(i < strlen(line)){
74          if(line[i] == '@'){
75              temp[index] = line[i];
76              index++;
77              i++;
78              while((47 < (int) line[i]) && ( (int) line[i]
                        < 58)){
79                  temp[index] = line[i];
80                  index++;
81                  i++;
82              }
83              temp[index] = '\0';
84              for(int y = 0; y < 4; y++){
85                  if(regs[y].temp != NULL){
86                      if(strcmp(regs[y].temp, temp) == 0){
87                      fprintf(file, "%s" ,regs[y].reg);
88                      }
89                  }
90
91              }
```

```
92              index = 0;
93              free(temp);
94              temp = malloc(sizeof(char) * 10);
95          }else{
96              fprintf(file, "%c", line[i]);
97              i++;
98          }
99      }
100 }
101
102 void requestRegister(char *str){
103     for(int i = 0; i < 4; i++){
104         if(regs[i].temp != NULL){
105             if(strcmp(regs[i].temp, str) == 0){
106                 return;
107             }
108         }
109     }
110     for(int i = 0; i < 4; i++){
111         if(regs[i].temp == NULL){
112             regs[i].temp = str;
113             return;
114         }
115     }
116     fprintf(stderr, "No available register.\n");
117 }
118
119 void freeRegs(livenessNode *node){
120     stringBuffer *in = node->in;
121     while(in != NULL){
122         if(!isPresent(node->out, in->buffer)){
123             unassignTemp(in->buffer);
124         }
125         in = in->next;
126     }
127 }
128
129 void unassignTemp(char *str){
130     for(int i = 0; i < 4; i++){
131         if(regs[i].temp != NULL){
132             if(strcmp(regs[i].temp, str) == 0){
133                 regs[i].temp = NULL;
134                 return;
135             }
136         }
137     }
```

```
138         fprintf(stderr, "No matching temporary.\n");
139     }
```