# CC, Spring 2018
# Exam project, part 3

| Group | 1 |
|---|---|



| Name | Mark Wolff Jervelund |
|---|---|
| Birthday | 280795 |
| Login | mjerv15@student.sdu.dk |
| Signature |  |

| Name | Troels Blicher Petersen |
|---|---|
| Birthday | 230896 |
| Login | trpet15@student.sdu.dk |
| Signature |  |

| Name | Morten Kristian Jæger |
|---|---|
| Birthday | 030895 |
| Login | mojae15@student.sdu.dk |
| Signature |  |



This report contains a total of ?? pages.

# 3. Assignment

Mark Jervelund, Troels Blicher Petersen & Morten Jæger
(Mjerv15, Trpet15, Mojae15)

Compiler (DM546)


IMADA

**March 16, 2018**

# Contents

# Introduction

In the third assignment of the Compiler project, we are tasked with implementing a weeder phase and a typechecking phase, with primary focus on the typechecking phase. This entire third assignment in made only in C.

## Build and execute

To build and execute first run either make, make all or make compiler.

```
make
```

All ways will build the binary called compiler.

If one wants debug info from the compiler, this can be added, by uncommenting the #define debugflag in debug.h.

```
#ifndef COMPILER_DEBUG_H

#define debugflag

#define COMPILER_DEBUG_H
```

By having it as a definition in the header file, it ensures that all debug related prints wont get compiled, when building the compiler for production use. However, it comes with the caveat, that the entire project has to be cleaned make clean if this is changed.

There are several ways to run the program, all of which can be found by executing the program compiler with the option "-h"

```
./compiler -h
```

The program accepts raw text input, in form of a program, as an argument. It also accepts files with the Shere Khan extension .src as an argument. Both ways will in the current state of the compiler return imformation that the compiler has gathered about the program to be compiled.

To remove all object files, run

```
make clean
```

To remove all object files and executable binaries, run

```
make clean-all
```

# Design

The design involves the weeder and the typechecker. The weeder is run before the typechecker, to weed out any errors and potentially find any expressions that can already be

## Weeder

Design of the weeder is similar to that of the parser. However, it differs in that it looks for expressions that can already be evaluated at compile time. This will slow down compilation time, however, it will also allows for some optimizations in the compiled program.
At this point our weeder is primarily focused on logical expressions. These are fairly straightforward to evaluate at compile time and also provide increases in performance, since comparing values is usually a "slow" process in processors due the operations itself and branch prediction.

Therefore, looking for expressions that evaluate true means, that there is no reason to do that comparison. This not only allows us to remove that if-statement, but it also allows us to remove other parts

of the program that cannot be run as a result of this. This could be an if-else-statement, because this code would never be able to run anyways.
Same can be said with if-statements that evaluates to false everytime. However, in that case, only the part that evaluates to false can be removed, as other parts of an if-else-statement might still be able to evaluate true.

The weeder can also look for virtual constants, that is, variables and expressions which are calculated in some way, but can be calculated already at compile time. This reduces the number of calculations to perform during execution of the program.

Lastly, but still very important, is the ability to check whether a function has any return calls. If not, the compiler should hault compiling and report an error, that a return is missing. This will be done using a stack, that keeps track of all returns inside and outside a function. A stack is smart, because it can be used in a way to describe all the contents of a function and scoping.

Adding a weeder will increase compile time, since it has to check for all of these things. However, there are many potential performance gains in the compiled program, which makes it a worthwile thing to do. One way to increase the speed of the weeder is to reuse the same tree as the one made by the parser. This way the entire program wont have to be read again and instead the datastructure describing the program can be used.

## Typechecker

The thought behind designing the typechecker, was to do this in three parts or phases. These three phases are the *"setup"*, *"pickup"* and "*"check"* phases.

### Setup Phase

In the setup phase, we go through the AST we have from parsing the program, and setup symbols and symbol tables for the different nodes in the AST.
The symbols we insert into our symboltables are f.x. the id of a function, the name of a variable.
To help with identifying what a symbols type is when we want to check it later, we have a structure called *"symbol_type"*, which contains information about the symbol it is in. If the symbol we insert is from a f.x. from a function, this *"symbol_type"* will also contain information about the functions, like its return type and such.
Since this phase is mostly just setting up symboltables and preparing for the "pickup" and "check" phases, this could possibly be used when we create the nodes themselves, and thus save a pass-through of the AST.

### Pickup Phase

In the pickup phase, we go through the AST again, but this time we try to resolve symbol that doesn't have a specific type yet. An example of this could be the following:

```
1 type t1 = t2;
2 type t2 = int;
```

When we first go through the AST in the setup phase, we do not know what type *t1* has yet, but when we go through the pickup phase, we resolve these problems. This phase is also used to resolve deeper recursively defined types, and resolve conflicts with these.

### Check Phase

In the check phase, we go through the AST and, since we should now know the type of everything in the given program, we can do the actual typechecking. The typechecking is done by checking the type we get from the *"symbol_type"* structure, with what we expect to actually occur. An example of this could be a comparison of two variables:

```
1 var a : int;
2 var b : int;
3 a = 5;
4 b = 4;
```

```
5 if (a > b) then
6 write 1;
```

In this case, there would be no problems with the typechecking, since we know that to say that a variable is larger than another variable, both of these variable must be integers in this language.
However if the program was as follows:

```
1 var a : int;
2 var b : bool;
3 a = 5;
4 b = true;
5 if (a > b) then
6 write 1;
```

We would get an error, since the two types are incompatible, according to our language.

# Implementation

## Weeder

```
1  if ((left_term->kind == term_NUM) && (right_term->kind == term_NUM)){
2            //We have an expression with two constants
3            switch(expression->kind){
4
5                case (exp_MULT):
6                    temp = make_Term_num(left_term->val.num * right_term->val.num);
7                    break;
```

Above is a small part of the weeding program, where we decide what to do if an expression consists of two numbers. In this case, we have a multiplication, which we can then resolve in compile time, instead of having to generate code for this calculation later.

```
1 if (expression->kind == exp_AND){
2
3                if ((left_term->kind == term_FALSE) || (right_term->kind == term_FALSE)){
4                    temp = make_Term_boolean(0);
5                }
6                if ((left_term->kind == term_TRUE) && (right_term->kind == term_TRUE)){
7                    temp = make_Term_boolean(1);
8                }
```

Above is a small part of the weeding program, where we decide what to do when we have and "AND" expression. Since we know this kind of boolean operations, we know that if either side of the "AND" expression is false, the whole expression will be false, and we can therefore set the expressions term to be false. This way, just like the other example, we don't have to calculate this later on.

```
1 if (stmt->val.ifthen.expression->kind == exp_TERM){
2                if (stmt->val.ifthen.expression->val.term->kind == term_FALSE){
3                    stmt = stmt->val.ifthen.statement2;
4                }
5                stmt = stmt->val.ifthen.statement1;
6            }
```

Above is a small part of the weeding program, where we decide what to do in a "if-else" expression. Here we check the type of the expression in the if statement, and based on it's type, we can decide what to do. Again, this is to weed out unnecessary code.
These methods of checking what type expressions and terms have is used throughout the weeder, to weed out the things we can in this phase.

## Typechecker

### Setup phase

```
symbol_table*nextTable;
    nextTable = scope_symbol_table(table);
    function->table = nextTable;
    function->tail->table = nextTable;
    setup_head(function->head, nextTable, table);
    setup_body(function->body, nextTable);
```

Above is a small part of the setup program, where we setup a function. To setup a function it, we need to give the function a new scope to work with. As seen in the code, we create a new scope, which is used when setting up the body of the function.

```
void setup_head(head *head, symbol_table *table, symbol_table *outer_scope){
    head->table = table;
    symbol_type *st;
    st = NEW(symbol_type);
    st->type = symbol_FUNCTION;
    put_symbol(outer_scope, head->id, 0, st);
```

Above is a small part of the setup program, where we setup the head of a function. Here it is shown how we make the function available for the rest of the program, by putting the "id" of the function into the symboltable "outer_scope", which it gets from the "setup_function" function, as seen earlier. As mentioned in the design section, since this is mostly setting up symboltables, this could possibly be put into an earlier pass-through of the AST.

### Pickup phase

```
case (type_ID):
    s = get_symbol(type->table, type->val.id);
    if (s == NULL || s->stype->type != symbol_ID){
        if (s == NULL){
    printf("Symbol is NULL\n");
    }
    if (s->stype->type != symbol_ID){
    printf("Symbol is not ID, it is of type: %d", s->stype->type);
    }
    print_error("Identifier error", 0, type->lineno);
    }
    struct type *temp;
    temp = resolve_recursive_type(s->stype->val.id_type);
    type->stype = temp->stype;
```

Above is a small part of the pickup program, where we try to find the type of an id. This happens when we f.x. assign a variables type to be that of an other variable. In this case, we would need to see if the id we are referring to exists, and if it does, we also check if it is a recursive definition of a type.

```
struct type *temp;
    temp = type;
        if (type->recursive_type == 1){
        print_error("Recursive type definition", 0, type->lineno);
    }
    type->recursive_type = 1;
    if (type->kind == type_ID){
        printf("Checking symbol table for symbol\n");
        SYMBOL *s;
        s = get_symbol(type->table, type->val.id);
        if (s == NULL || s->stype->type != symbol_ID){
            print_error("Undefined identifier", 0, type->lineno);
```

```
13          }
14          temp = resolve_recursive_type(s->stype->val.id_type);
15      }
16      type->recursive_type = 0;
17      return temp;
```

Above is a small part of the pickup program, where we check to see if a type is recursively defined. This is done by first setting a flag, "recursive_type", to 1, which will indicate that we have now seen this type in this check. Afterwards, we just checks its type recursively, until we find a definitive type.

### Check phase

```
1  case (exp_PLUS):
2      case (exp_MIN):
3      case (exp_MULT):
4      case (exp_DIV):
5          check_exp(exp->val.ops.left);
6          check_exp(exp->val.ops.right);
7          if (exp->val.ops.left->stype->type == symbol_INT && exp->val.ops.right->stype->type ==
               symbol_INT){
8
9              st = NEW(symbol_type);
10             st->type = symbol_INT;
11             exp->stype= st;
12
13         } else {
14             print_error("Operators in arithmetic expression are not integers", 0, exp->lineno);
15         }
16         break;
```

Above is a part of the check program, where we check the types in an expression, in this case an arithmetic expression. Since we know that the types of the kinds of expressions need to be integers, we check the "symbol_type" structure associated with each expression, and check the type of that. We also return an error message and exit the program if this is not the case.
In further extensions of the language and the compiler, "+" and "·" could be made to work with strings/chars, like they work in f.x. Java, where "Hi"+2 would result in the string "Hi2".

```
1  case (statement_RETURN):
2              check_exp(stmt->val.ret);
3              if (stmt->function->stype->val.func_type.ret_type->stype->type != stmt->val.ret->
                   stype->type){
4                  print_error("Wrong return type", 0, stmt->lineno);
5              }
6              break;
```

Above is a small part of the check program, where we check the return type of a function. This is done by comparing the type after the "return" statement, with the type of the function this statement belongs to. Again, we use the "symbol_type" structure to keep information about the function it belongs to. This was also mentioned in the design section, in the section about the "setup" phase.

## Testing

All the tests can be found in the appendix and in the tests// directory in the project directory with the same name as here. In this section we will give a short explanation as to what the different tests are meant to test. Furthermore, the testing i divided into two sections. The first section is weeder related and the second section is typechecker related.

### Weeder tests

Since the main use of the weeder is to weed expressions, the tests are mostly used to test that.

**test_arithmetic_multiply.src**

This test is used to check if the weeder can evaluate an arithmetic expression, which consists of two numbers, so that we don't have to generate code for this later.

**test_arithmetric_zero_division.src**

This test is used to check if we correctly catch a division by 0 error in an expression, and to see if we print the error correctly.

**test_if-else_only_boolean.src**

This test is used to check whether or not we can evaluate a boolean expression in a "if-else" statement, in such a way that we can remove the code in either the "if-then" or the "else" part.

**test_if-else_boolean_expression.src**

This test is used to check whether or not we can evaluate a boolean expression consisting of an expressions and a boolean, to see if we can reduce this to either the expression or the boolean, depending on what the boolean operator is.

**test_return_inside_outside.src**

This test is used to check whether or not we can check if there is a return statement outside of a function definition.

**test_no_return_if.src**

This test is about detecting return statements outside of functions. This is therefore also a test of the stack used for this purpose.

**test_no_return_multiple_functions.src**

This test is an amendment to the previous test (test_8.src). In this case there are two functions, the first where the return statement is inside the function and the second function has the return statement outside of its scope.

**test_return_not_enough.src**

This test is about detecting if there is enough return statements in a function, so that a return of some value is always guaranteed.

## Typechecker tests

The main tests of the typechecker will be of the "checker" program, but there will be a few tests of the "pickup" program.

**test_recursive_pickup.src**

This test is used to check whether or not we can check for a recursive type definition in the pickup phase.

**test_recursive_pickup_2.src**

This test is used to check whether or not we can check for a recursive type definition in the pickup phase. This is different from "test_6.src", because in this case there is a recursive type definition.

**test_function_return_type.src**

This test is used to check whether or not we can check the return type of a function, to see if what we return in the function is of the expected type.

#### test_function_wrong_return_type.src

This test is used to check whether or not we can check the return type of a function, to see if what we return in the function is of the expected type. This is different from "test12.src", because in this case a function returns the wrong type.

#### test_arithmetic_typecheck.src

This test is used to check whether or not we can check the types of values used in an arithmetic expression, to see if these evaluate to integers.

#### test_arithmetic_wrong_types.src

This test is used to check whether or not we can check the types of values used in an arithmetic expression, to see if the evaluate to integers. This is different from "test15.src", because in this case we do not use two integers in the arithmetic expression.

#### test_function_arguments.src

This test is used to check whether or not we can check the types of arguments given in a function call.

#### test_function_arguments_too_few.src

This test is used to check whether or not we can check the amount of arguments given in a function call. In this case there are too few arguments.

#### test_function_arguments_to_many.src

This test is used to check whether or not we can check the amount of arguments given in a function call. In this case there are too many arguments.

#### test_function_exists.src

This test is used to check whether or not we can check if a reference to a function actually exists.

#### test_array_index.src

This test is used to check whether or not we can check if the index of an array if an integer or not.

# Results

## Weeder

#### test_arithmetic_multiply.src

This test is successful, as it correctly identifies and calculates $a$ to be a constant with value 4.

```
1 var a : int;
2 a = 4 : int;
```

#### test_arithmetric_zero_division.src

This test is successful, as the compiler correctly identifies a division by zero.

```
1 Division by 0 error at line 3
```

### test_if-else_only_boolean.src

This test is successful. The compiler sees that the if-statement can be evaluated at compile time to be true, thus it can remove the else-part, because that part wont ever be reachable. It also correctly removes the if-statement, since it is not needed anymore.

```
write 1 : int;
```

### test_if-else_boolean_expression.src

This test is not successful, because we expected it to also evaluate the if-statement to true. This would mean that it should have removed the else-part and the if-statement itself, leaving only "write 1 : int;" to remain. The reason why it doesn't do that is, that the first part is an expression and the second part is a term. In the current state of the weeder, this is not evaluated, because the weeder doesn't evaluate the result of an expression and a term, yet.

```
var n : int;
n = 1 : int;
if (((n : int > 0 : int) : boolean || true : boolean) : boolean) then
    write 1 : int;
 else
    write 2 : int;
```

### test_return_inside_outside.src

This test is successful. The compiler correctly identifies that a return statement is left outside of a function. Furthermore, it also reports on which line this is found.

```
Return outside of function at line 6
```

### test_no_return.src

This test is successful since the function will not work without a return statement. However, it seg-faults, because of the parser.

```
syntax error before end
Segmentation fault (core dumped)
```

### test_no_return_if.src

This test is unsuccessful, because it should identify that there is no return statement present in the function. However, the if-statement seems to be the reason for this, which is further explored in test_return_not_enoug

```
func test(n : int) : int
    var n : int;
    if ((n : int == 0 : int) : boolean || (n : int == 1 : int) : boolean) : boolean) then
        n = 3 : int;
end test : function(n : int) : int
write test(1 : int) : int;
```

### test_no_return_multiple_functions.src

As the output shows, the weeder catches a return that does not belong to a function, which results in a error.

```
Return outside of function at line 16
```

### test_return_not_enough.src

As the output shows, the compiler does not detect that the single return statement is only reachable in some situations, where n greater than or equal to 0 and not 2.

```
1  func test(n : int) : int
2      var n : int;
3      var a : int;
4      var b : int;
5      if (n : int >= 0 : int) : boolean) then
6          {
7              if ((n : int == 2 : int) : boolean) then
8                  {
9                      b = 2 : int;
10                 }
11             else
12                 {
13                     n = (n : int+b : int) : int;
14                     return n : int;
15                 }
16         }
17 end test : function(n : int) : int
18 write test(1 : int) : int;
```

## Typechecker

### test_recursive_pickup.src

As the output shows, the pickup phase allows this program, since recursively defined types end up being a specific type.

```
1  Checking symbol table for symbol
2  Checking symbol table for symbol
3  Checking symbol table for symbol
4  Checking symbol table for symbol
5  type r1 = r2 : record of {x : int};
6  type r2 = r3 : record of {x : int};
7  type r3 = record of { x : int };
8  var v1 : r1 : record of {x : int};
9  var v2 : r2 : record of {x : int};
10 var v3 : r3 : record of {x : int};
11 write 42 : int;
```

### test_recursive_pickup_2.src

As the output shows, we have a recursively defined type that does not end up being a specific type ("int", "bool", etc."), which results in an error.

```
1  Recursive type definition at line 3
```

### test_function_return_type.src

As the output shows, the typechecker allows this program, since the return value of the function, is the same as the defined return type.

```
1  type recordType = record of { x : int, y : int };
2  func a(x : int, y : int) : recordType : record of {x : int, y : int}
3      var p2 : recordType : record of {x : int, y : int};
4      allocate p2;
5      p2.x = x : int;
6      p2.y = y : int;
```

```
7     return p2 : record of {x : int, y : int};
8 end a : function(x : int, y : int) : record of {x : int, y : int}
9 var p1 : recordType : record of {x : int, y : int};
10 p1 = a(10 : int, 2 : int) : record of {x : int, y : int};
11 write (p1.x : int/p1.y : int) : int;
```

### test_function_wrong_return_type.src

As the output shows, when we return the wrong type in a function, we output an error.

```
1 Wrong return type at line 13
```

### test_arithmetic_typecheck.src

As the output shows, the typechecker allows arithmetic expressions, when both of the terms used in the expression is an integer. The type of the "+" operator can also be seen here, which is of the type "int".

```
1 var a : int;
2 var b : int;
3 a = 1 : int;
4 b = 3 : int;
5 write (a : int+b : int) : int;
```

### test_arithmetic_wrong_types.src

As the output shows, some of the types used in an arithmetic expressions is not an integer, which results in an error.

```
1 Operators in arithmetic expression are not integers at line 7
```

### test_function_arguments.src

As the output shows, the type of a function argument is not of the expected type. An improvement of this would be to also tell the user what type they used, and what the expected type would be in the function.

```
1 Function argument type mismatch at line 20
```

### test_function_arguments_too_few.src

As the output shows, the amount of arguments to a function are too few.

```
1 Too few function arguments at line 20
```

### test_function_arguments_to_many.src

As the output shows, the amount of arguments to a function are too many.

```
1 Too many function arguments at line 20
```

### test_function_exists.src

As the output shows, when we reference a function that does not exists, we output an error.

```
1 Reference to function that does not exists at line 20
```

**test_array_index.src**

As the output shows, when we try to used a value that is now an integer to get the index from an array, we output an error.

```
1 Expression in [] not an integer at line 5
```

# Conclusion

From the tests we have run, we can conclude that the weeder works in most of the cases that we want it to work on, and that the typechecker functions properly.
From the printed output, we can also see that the types match what we would expect.

# Appendix

## Complete test output

```
+-------------+
|    Tests    |
+-------------+
+--------------------
| TEST: ./tests/typechecker/test_arithmetic_typecheck.src
+--------------------
var a : int;
var b : int;
a = 1 : int;
b = 3 : int;
write (a : int+b : int) : int;


+--------------------
| TEST: ./tests/typechecker/test_arithmetic_wrong_types.src
+--------------------
Operators in arithmetic expression are not integers at line 7

+--------------------
| TEST: ./tests/typechecker/test_array_index.src
+--------------------
Expression in [] not an integer at line 5

+--------------------
| TEST: ./tests/typechecker/test_function_arguments.src
+--------------------
Function argument type mismatch at line 20

+--------------------
| TEST: ./tests/typechecker/test_function_arguments_to_many.src
+--------------------
Too many function arguments at line 20

+--------------------
| TEST: ./tests/typechecker/test_function_arguments_too_few.src
+--------------------
Too few function arguments at line 20

+--------------------
| TEST: ./tests/typechecker/test_function_exists.src
+--------------------
Reference to function that does not exists at line 20

+--------------------
| TEST: ./tests/typechecker/test_function_return_type.src
+--------------------
type recordType = record of { x : int, y : int };
func a(x : int, y : int) : recordType : record of {x : int, y : int}
    var p2 : recordType : record of {x : int, y : int};
    allocate p2;
    p2.x = x : int;
    p2.y = y : int;
    return p2 : record of {x : int, y : int};
end a : function(x : int, y : int) : record of {x : int, y : int}
var p1 : recordType : record of {x : int, y : int};
p1 = a(10 : int, 2 : int) : record of {x : int, y : int};
write (p1.x : int/p1.y : int) : int;


```

```
+--------------------
| TEST: ./tests/typechecker/test_function_wrong_return_type.src
+--------------------
Wrong return type at line 13

+--------------------
| TEST: ./tests/typechecker/test_recursive_pickup_2.src
+--------------------
Checking symbol table for symbol
Checking symbol table for symbol
Checking symbol table for symbol
Recursive type definition at line 3

+--------------------
| TEST: ./tests/typechecker/test_recursive_pickup.src
+--------------------
Checking symbol table for symbol
Checking symbol table for symbol
Checking symbol table for symbol
Checking symbol table for symbol
type r1 = r2 : record of {x : int};
type r2 = r3 : record of {x : int};
type r3 = record of { x : int };
var v1 : r1 : record of {x : int};
var v2 : r2 : record of {x : int};
var v3 : r3 : record of {x : int};
write 42 : int;


+--------------------
| TEST: ./tests/weeder/test_arithmetic_multiply.src
+--------------------
var a : int;
a = 4 : int;


+--------------------
| TEST: ./tests/weeder/test_arithmetric_zero_division.src
+--------------------
Division by 0 error at line 3

+--------------------
| TEST: ./tests/weeder/test_if-else_boolean_expression.src
+--------------------
var n : int;
n = 1 : int;
if (((n : int > 0 : int) : boolean || true : boolean) : boolean) then
    write 1 : int;
 else
    write 2 : int;


+--------------------
| TEST: ./tests/weeder/test_if-else_only_boolean.src
+--------------------
write 1 : int;


+--------------------
| TEST: ./tests/weeder/test_no_return_if.src
+--------------------
func test(n : int) : int
    var n : int;
```

```
123     if ((n : int == 0 : int) : boolean || (n : int == 1 : int) : boolean) : boolean) then
124         n = 3 : int;
125 end test : function(n : int) : int
126 write test(1 : int) : int;
```

## Complete code listing

### main.h

```
1 #ifndef __main_h
2 #define __main_h
3
4
5 int main(int argc, char **argv);
6
7 #endif
```

### main.c

```
1  /**
2   * @brief Compiler program.
3   *
4   * @file main.c
5   * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
6   * @date 2018-03-09
7   */
8  #include <string.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <ctype.h>
12 #include <getopt.h>
13 #include "debug.h"
14
15 #include "main.h"
16 #include "auxiliary.h"
17 #include "symbol.h"
18 #include "tree.h"
19 #include "pretty.h"
20 #include "y.tab.h"
21 #include "weeder.h"
22 #include "typechecker.h"
23
24
25 int lineno;
26
27 body *theprogram;
28
29 int main(int argc, char **argv) {
30     int helpflag = 0;
31     int bflag = 0;
32     char *cvalue = NULL;
33     int index;
34     int c;
35
36     opterr = 0;
37
38     int files[argc];
39
40     while ((c = getopt(argc, argv, "hc:")) != -1) {
41         switch (c) {
```

```
42          case 'h':
43              helpflag = 1;
44              files[optind - 2] = 1;
45              break;
46          case 'c':
47              cvalue = optarg;
48              files[optind - 3] = 1;
49              files[optind - 2] = 1;
50              break;
51          case '?':
52              if (optopt == 'c')
53                  fprintf(stderr, "Option -%c requires an argument.\n", optopt);
54              else if (isprint(optopt))
55                  fprintf(stderr, "Unknown option `-%c'.\n", optopt);
56              else
57                  fprintf(stderr,
58                          "Unknown option character `\\x%x'.\n",
59                          optopt);
60              return 1;
61          default:
62              abort();
63          }
64      }
65      if (helpflag) {
66          system("man ./manual");
67          return 0;
68      }
69
70      if (optind < argc) {
71          for (int i = 1; i < argc; ++i) {
72              if (files[i] == 0) {
73                  if (ends_with(argv[i], ".src")) {
74                      freopen(argv[i], "r", stdin);
75                  }
76              }
77          }
78      }
79
80      lineno = 1;
81      yyparse();
82
83      weeder_init(theprogram);
84      types = 0;
85      //prettyProgram(theprogram);
86
87
88      //printf("\nStarting typechecking\n\n");
89
90  #if debugflag > 0
91      printf("\nStarting typechecking\n\n");
92  #endif
93      typecheck(theprogram);
94  #if debugflag > 0
95      printf("\nAfter typechecking\n\n");
96  #endif
97      types = 1;
98      prettyProgram(theprogram);
99
100
101     printf("\n");
102     return 1;
103 }
```

**exp.y**

```
1
2  //Comments
3  // String followed by : is detected as a decleartion evening when within a string, the code
       still works, but syntax highlighting is broken.
4
5  %{
6  #include <stdio.h>
7  #include "tree.h"
8  #include "y.tab.h"
9  extern char *yytext;
10 //extern EXP *theexpression;
11 extern body *theprogram;
12 void yyerror() {
13    printf("syntax error before %s\n",yytext);
14 }
15 %}
16
17 %union {
18     int intconst;
19     char *stringconst;
20     struct EXP *exp;
21     struct function *function;
22     struct head *head;
23     struct tail *tail;
24     struct type *type;
25     struct par_decl_list *par_decl_list;
26     struct var_decl_list *var_decl_list;
27     struct var_type *var_type;
28     struct body *body;
29     struct decl_list *decl_list;
30     struct declaration *declaration;
31     struct statement_list *statement_list;
32     struct statement *statement;
33     struct variable *variable;
34     struct expression *expression;
35     struct term *term;
36     struct act_list *act_list;
37     struct exp_list *exp_list;
38 }
39
40 %token <intconst> tINTCONST
41 %token <stringconst> tIDENTIFIER
42 %token EQ
43 %token NEQ
44 %token LEQ
45 %token GEQ
46 %token LT
47 %token GT
48 %token IF
49 %token ELSE
50 %token WHILE
51 %token RETURN
52 %token AND
53 %token OR
54 %token ASSIGN
55 %token TRUE
56 %token FALSE
57 %token _NULL
58 %token FUNC
59 %token END
60 %token INT
```

```
61  %token BOOL
62  %token ARRAY_OF
63  %token RECORD_OF
64  %token TYPE
65  %token VAR
66  %token OF_LENGTH
67  %token THEN
68  %token WRITE
69  %token ALLOCATE
70  %token DO
71
72
73
74  %type <function> function
75  %type <head> head
76  %type <tail> tail
77  %type <type> type
78  %type <par_decl_list> par_decl_list
79  %type <var_decl_list> var_decl_list
80  %type <var_type> var_type
81  %type <body> body
82  %type <decl_list> decl_list
83  %type <declaration> declaration
84  %type <statement_list> statement_list
85  %type <statement> statement
86  %type <expression> expression
87  %type <term> term
88  %type <act_list> act_list
89  %type <variable> variable
90  %type <exp_list> exp_list
91
92  %start program
93
94  %precedence NEG
95
96  %left AND '|'
97  %left EQ NEQ
98  %left GT LT GEQ LEQ
99  %left '+' '-'
100 %left '*' '/'
101 %nonassoc THEN
102 %nonassoc ELSE
103
104 %%
105 program:  body
106         { theprogram = $1;}
107 ;
108
109 function:  head body tail
110        {$$ = make_Func($1, $2, $3);
111        if (check_Func($1, $3) != 0){
112            fprintf(stderr, "Function name: %s, at line %i, does not match function name: %s, at
                   line %i\n ", $1->id, $1->lineno, $3->id, $3->lineno);
113            YYABORT;
114            }}
115 ;
116
117 head:  FUNC tIDENTIFIER '(' par_decl_list ')' ':' type
118        {$$ = make_Head($2, $4, $7);}
119 ;
120
121 tail:  END tIDENTIFIER
122        {$$ = make_Tail($2);}
```

```
123 ;
124
125 type:  tIDENTIFIER
126         {$$ = make_Type_id($1);}
127             | INT
128         {$$ = make_Type_int();}
129             | BOOL
130         {$$ = make_Type_bool();}
131             | ARRAY_OF type
132         {$$ = make_Type_array($2);}
133             | RECORD_OF '{' var_decl_list '}'
134         {$$ = make_Type_record($3);}
135 ;
136
137 par_decl_list: var_decl_list
138         {$$ = make_PDL_list($1);}
139             | /*empty*/
140         {$$ = make_PDL_empty();}
141 ;
142
143 var_decl_list: var_type ',' var_decl_list
144         {$$ = make_VDL_list($1, $3);}
145             | var_type
146         {$$ = make_VDL_type($1);}
147 ;
148
149 var_type:tIDENTIFIER ':' type
150         {$$ = make_VType_id($1, $3);}
151 ;
152
153 body:decl_list statement_list
154         {$$ = make_Body($1, $2);}
155
156 ;
157
158 decl_list:declaration decl_list
159         {$$ = make_DL_list($1, $2);}
160             | /*empty*/
161         {$$ = make_DL_empty();}
162 ;
163
164 declaration:TYPE tIDENTIFIER '=' type ';'
165         {$$ = make_Decl_type($2, $4);}
166             | function
167         {$$ = make_Decl_func($1);}
168             | VAR var_decl_list ';'
169         {$$ = make_Decl_list($2);}
170 ;
171
172 statement_list:statement
173         {$$ = make_SL_statement($1);}
174             | statement statement_list
175         {$$ = make_SL_list($1, $2);}
176 ;
177
178 statement:RETURN expression ';'
179         {$$ = make_STMT_ret($2);}
180             | WRITE expression ';'
181         {$$ = make_STMT_wrt($2);}
182             | ALLOCATE variable ';'
183         {$$ = make_STMT_allocate_var($2);}
184             | ALLOCATE variable OF_LENGTH expression ';'
185         {$$ = make_STMT_allocate_length($2, $4);}
```

```
186             | variable '=' expression ';'
187         {$$ = make_STMT_assign($1, $3);}
188             | IF expression THEN statement  %prec THEN
189         {$$ = make_STMT_if($2, $4);}
190             | IF expression THEN statement ELSE statement
191         {$$ = make_STMT_if_else($2, $4, $6);}
192             | WHILE expression DO statement
193         {$$ = make_STMT_while($2, $4);}
194             | '{' statement_list '}'
195         {$$ = make_STMT_list($2);}
196 ;
197
198 variable:tIDENTIFIER
199         {$$ = make_Var_id($1);}
200             | variable '[' expression ']'
201         {$$ = make_Var_exp($1, $3);}
202             | variable '.' tIDENTIFIER
203         {$$ = make_Var_record($1, $3);}
204 ;
205
206 expression:expression '+' expression
207         {$$ = make_EXP(exp_PLUS, $1, $3);}
208             | expression '-' expression
209         {$$ = make_EXP(exp_MIN, $1, $3);}
210             | expression '*' expression
211         {$$ = make_EXP(exp_MULT, $1, $3);}
212             | expression '/' expression
213         {$$ = make_EXP(exp_DIV, $1, $3);}
214             | '(' expression ')'
215         {$$ = $2;}
216             | expression EQ expression
217         {$$ = make_EXP(exp_EQ, $1, $3);}
218             | expression NEQ expression
219         {$$ = make_EXP(exp_NEQ, $1, $3);}
220             | expression GT expression
221         {$$ = make_EXP(exp_GT, $1, $3);}
222             | expression LT expression
223         {$$ = make_EXP(exp_LT, $1, $3);}
224             | expression GEQ expression
225         {$$ = make_EXP(exp_GEQ, $1, $3);}
226             | expression LEQ expression
227         {$$ = make_EXP(exp_LEQ, $1, $3);}
228             | expression AND expression
229         {$$ = make_EXP(exp_AND, $1, $3);}
230             | expression '|''|' expression
231         {$$ = make_EXP(exp_OR, $1, $4);}
232             | '-' expression %prec NEG
233         {$$ = make_EXP_neg($2);}
234             | term
235         {$$ = make_EXP_term($1);}
236 ;
237
238 term:       tINTCONST
239         {$$ = make_Term_num($1);}
240             | '(' expression ')'
241         {$$ = make_Term_par($2);}
242             | '!' term
243         {$$ = make_Term_not($2);}
244             | '|' expression '|'
245         {$$ = make_Term_abs($2);}
246             | TRUE
247         {$$ = make_Term_boolean(1);}
248             | FALSE
```

```
249         {$$ = make_Term_boolean(0);}
250             | _NULL
251         {$$ = make_Term_null();}
252             | variable
253         {$$ = make_Term_variable($1);}
254             | tIDENTIFIER '(' act_list ')'
255         {$$ = make_Term_list($1, $3);}
256 ;
257
258 act_list:exp_list
259         {$$ = make_Act_list($1);}
260             | /*empty*/
261         {$$ = make_Act_empty();}
262 ;
263
264 exp_list:expression
265         {$$ = make_ExpL_exp($1);}
266             | expression ',' exp_list
267         {$$ = make_ExpL_list($1, $3);}
268 ;
269
270 %%
```

**exp.l**

```
1  %{
2  #include "y.tab.h"
3  #include <string.h>
4
5  extern int lineno;
6  extern int fileno();
7  int nested_comment = 0;
8
9  %}
10
11 %x COMMENT_SINGLE
12 %x COMMENT_MULTI
13
14 %option noyywrap nounput noinput
15
16 /* abbreviation of symbols we match on, TO BE EXPANDED */
17 SYMBOLS [+\-*\/\(\)\[\]{}!\|,\.=;:]
18
19 %%
20 [ \t]+       /* ignore */;
21 \n             lineno++;
22
23 {SYMBOLS}     return yytext[0];
24
25 "=="          return EQ;
26 "!="          return NEQ;
27 "<"           return LT;
28 "<="          return LEQ;
29 ">"           return GT;
30 ">="          return GEQ;
31 "if"          return IF;
32 "else"        return ELSE;
33 "while"       return WHILE;
34 "return"      return RETURN;
35 "&&"          return AND;
36 "true"        return TRUE;
```

```
"false"         return FALSE;
"null"          return _NULL;

"func"          return FUNC;
"end"           return END;

"int"           return INT;
"bool"          return BOOL;
"array of"      return ARRAY_OF;
"record of"     return RECORD_OF;
"type"          return TYPE;
"var"           return VAR;
"write"         return WRITE;
"allocate"      return ALLOCATE;
"of length"     return OF_LENGTH;
"then"          return THEN;
"do"            return DO;

0|([1-9][0-9]*)      { yylval.intconst = atoi(yytext);
                       return tINTCONST; }

[a-zA-Z_][a-zA-Z0-9_]* { yylval.stringconst = (char *)malloc(strlen(yytext)+1);
                         sprintf(yylval.stringconst,"%s",yytext);
                         return tIDENTIFIER; }

"#"             BEGIN(COMMENT_SINGLE);
"(*"            nested_comment++; BEGIN(COMMENT_MULTI);

<COMMENT_SINGLE>{

\n              lineno++; BEGIN(0);
.               /* ignore */

}

<COMMENT_MULTI>{

\n              lineno++;
"(*"            nested_comment++;
"*)"            {   nested_comment--;
                    if (nested_comment == 0){
                        BEGIN(0);
                    }
                }
.               /* ignore */
<<EOF>>         fprintf(stderr, "Comment not closed at the end of the file. Found at line: %i\n",
     lineno); exit(1);

}


.               fprintf(stderr, "Unrecognized symbol. Found at line: %i\n", lineno); exit(1);

%%
```

**symbol.h**

```
#ifndef __symbol_h
#define __symbol_h

#include "tree.h"
```

```
5
6  #define HashSize 317
7  /* SYMBOL will be extended later.
8  Function calls will take more parameters later.
9  */
10
11 typedef enum { symbol_ID,
12               symbol_INT,
13               symbol_BOOL,
14               symbol_RECORD,
15               symbol_ARRAY,
16               symbol_NULL,
17               symbol_FUNCTION,
18               symbol_UNKNOWN } SYMBOL_type;
19
20 typedef struct SYMBOL {
21     char *name;
22     int value;
23     struct SYMBOL *next;
24     struct symbol_type *stype;
25 } SYMBOL;
26
27 typedef struct symbol_table {
28     SYMBOL *table[HashSize];
29     struct symbol_table *next;
30 } symbol_table;
31
32
33 typedef struct symbol_type {
34     SYMBOL_type type;
35     union {
36         struct type *array_type;
37         struct var_decl_list *record_type;
38         struct type *id_type;
39         struct {
40             struct type *ret_type;
41             struct par_decl_list *pdl;
42             struct function *func;
43         } func_type;
44     } val;
45     int printed;
46 } symbol_type;
47
48 int hash(char *str);
49
50 struct symbol_table *init_symbol_table();
51
52 struct symbol_table *scope_symbol_table(symbol_table *t);
53
54 SYMBOL *put_symbol(symbol_table *t, char *name, int value, symbol_type *st);
55
56 SYMBOL *get_symbol(symbol_table *t, char *name);
57
58 void dump_symbol_table(symbol_table *t);
59
60 SYMBOL *check_local(symbol_table *t, char *name);
61
62 void print_symbol(SYMBOL *symbol);
63
64 #endif
```

**symbol.c**

```c
/**
 * @brief Symbol table using hashing.
 *
 * @file symbol.c
 * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
 * @date 2018-03-09
 */

#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include "symbol.h"
#include "memory.h"

/**
 * @brief Computes the hash function as seen below.
 *
 * @param str
 * @return int
 */
int hash(char *str) {
    int length;
    length = (unsigned)strlen(str);
    int k = (int)str[0];
    int i;
    int pointer = 1;

    while (pointer < length) {
        k = k << 1;
        i = (int)str[pointer];
        k = i + k;
        pointer++;
    }
    return (k % HashSize);
}

/**
 * @brief Initializes the symbol table
 *
 * @return symbol_table* Returns a pointer to a new initialized hash table (of
 * type SymbolTable)
 */
symbol_table *init_symbol_table() {

    int i = 0;
    symbol_table *table = Malloc(sizeof(SYMBOL) * HashSize);
    table->next = NULL;
    while (i < HashSize) {
        table->table[i] = NULL;
        i++;
    }
    return table;
}

/**
 * @brief
 *
 * @param t Pointer to a hash table
 * @return symbol_table* Returns a new hash table pointing to t.
 */
symbol_table *scope_symbol_table(symbol_table *t) {
```

```
62      symbol_table *newTable = init_symbol_table();
63      newTable->next = t;
64      return newTable;
65  }
66
67  /**
68   * @brief put_symbol takes a hash table and a string, name, as arguments and
69   * inserts name into the hash table together with the associated value value.
70   * A pointer to the SYMBOL value which stores name is returned.
71   *
72   * @param t Pointer to hash table.
73   * @param name
74   * @param value
75   * @param st
76   * @return SYMBOL*
77   */
78  SYMBOL *put_symbol(symbol_table *t, char *name, int value, symbol_type *st) {
79      if (t == NULL) {
80          return NULL;
81      }
82      SYMBOL *localCheck = check_local(t, name);
83      //Symbol already exists
84      if (localCheck != NULL) {
85          return localCheck;
86      } else {
87          int hashValue = hash(name);
88          //pretty("Putting symbol with name: %s, value: %d, type: %d, table: %p, hash: %d\n",
                name, value, st->type, t, hashValue);
89
90          SYMBOL *symbol = Malloc(sizeof(SYMBOL));
91          symbol->name = name;
92          symbol->value = value;
93          symbol->stype = st;
94          symbol->next = Malloc(sizeof(SYMBOL));
95
96          //Placed in front of the list
97          symbol->next = t->table[hashValue];
98          t->table[hashValue] = symbol;
99          return symbol;
100     }
101 }
102
103 /*
104     getSymbol takes a hash table and a string name as arguments and searches for
105     name in the following manner: First search for name in the hash table which
106     is one of the arguments of the function call. If name is not there, continue the
107     search in the next hash table. This process is repeatedly recursively. If name has
108     not been found after the root of the tree (see Fig. 1) has been checked, the result
109     NULL is returned. If name is found, return a pointer to the SYMBOL value in
110     which name is stored
111     */
112 SYMBOL *get_symbol(symbol_table *t, char *name) {
113     //   First check if t is null
114     //pretty("Getting symbol %s, from table %p\n", name, t);
115     if (t == NULL) {
116         //pretty("Table is null\n");
117         return NULL;
118     }
119
120     SYMBOL *localCheck = check_local(t, name);
121
122     //Symbol in local table?
123     if (localCheck != NULL) {
```

```
124         //pretty("Trying local check, type: %d\n", localCheck->stype->type);
125         return localCheck;
126     }
127 
128     if (t->next != NULL) {
129         //pretty("Checking next\n");
130         return get_symbol(t->next, name);
131     }
132 
133     //Symbol does not exists
134     return NULL;
135 }
136 
137 /*
138  * dumpSymbolTable takes a pointer to a hash table t as argument and prints all
139  * the (name, value) pairs that are found in the hash tables from t up to the root.
140  * Hash tables are printed one at a time. The printing should be formatted in a nice
141  * way and is intended to be used for debugging (of other parts of the compiler).
142 */
143 void dump_symbol_table(symbol_table *t) {
144     if (t == NULL) {
145         return;
146     }
147 
148     printf("Printing symbol table:\n\n");
149 
150     for (int i = 0; i < HashSize; i++) {
151         if (t->table[i] != NULL) {
152             print_symbol(t->table[i]);
153             printf("\n");
154         }
155     }
156     printf("\n");
157 
158     dump_symbol_table(t->next);
159 }
160 
161 /*
162  * Check the current table we are in for a value
163  */
164 SYMBOL *check_local(symbol_table *t, char *name) {
165     int hashValue = hash(name);
166 
167     SYMBOL *symbol = t->table[hashValue];
168     if (symbol == NULL) {
169         //pretty("Local symbol is null\n");
170         return NULL;
171     } else {
172         while (symbol != NULL) {
173             if (strcmp(name, symbol->name) == 0) {
174                 //pretty("Compared %s and %s, success\n", name, symbol->name);
175                 return symbol;
176             }
177             symbol = symbol->next;
178         }
179     }
180 
181     //Hash value for the symbol exists, but the symbol is not in the table
182     return NULL;
183 }
184 
185 void print_symbol(SYMBOL *symbol) {
186     printf("(%s, %i)", symbol->name, symbol->value);
```

```
187 }
```

**tree.h**

```
1  #ifndef __tree_h
2  #define __tree_h
3
4  #include "kind.h"
5  #include "symbol.h"
6
7  typedef struct function {
8      struct symbol_table*table;
9      struct symbol_type *stype;
10     int lineno;
11     struct head *head;
12     struct body *body;
13     struct tail *tail;
14
15 } function;
16
17 typedef struct head {
18     struct symbol_table*table;
19     struct symbol_type *stype;
20     int lineno;
21     char *id;
22     struct par_decl_list *list;
23     struct type *type;
24     int args;
25
26 } head;
27
28 typedef struct tail {
29     struct symbol_table*table;
30     int lineno;
31     char *id;
32 } tail;
33
34 typedef struct type {
35     struct symbol_table*table;
36     struct symbol_type *stype;
37     int recursive_type;
38     int lineno;
39     TYPE_kind kind;
40     union {
41         char *id;
42         struct type *type;
43         struct var_decl_list *list;
44     } val;
45 } type;
46
47 typedef struct par_decl_list {
48     struct symbol_table*table;
49     int lineno;
50     PDL_kind kind;
51     struct var_decl_list *list;
52 } par_decl_list;
53
54 typedef struct var_decl_list {
55     struct symbol_table*table;
56     int lineno;
57     VDL_kind kind;
```

```
58      struct var_decl_list *list;
59      struct var_type *vartype;
60  } var_decl_list;
61
62  typedef struct var_type {
63      struct symbol_table*table;
64      struct SYMBOL *symbol;
65      int lineno;
66      char *id;
67      struct type *type;
68  } var_type;
69
70  typedef struct body {
71      struct symbol_table*table;
72      int lineno;
73      struct decl_list *d_list;
74      struct statement_list *s_list;
75  } body;
76
77  typedef struct decl_list {
78      struct symbol_table*table;
79      int lineno;
80      DL_kind kind;
81      struct declaration *decl;
82      struct decl_list *list;
83  } decl_list;
84
85  typedef struct declaration {
86      struct symbol_table*table;
87      int lineno;
88      DECL_kind kind;
89      union {
90          struct {
91              char *id;
92              struct type *type;
93          } type;
94          struct function *function;
95          var_decl_list *list;
96      } val;
97
98  } declaration;
99
100 typedef struct statement_list {
101     struct symbol_table*table;
102     int lineno;
103     SL_kind kind;
104     struct statement *statement;
105     struct statement_list *list;
106
107 } statement_list;
108
109 typedef struct statement {
110     struct symbol_table*table;
111     int lineno;
112     STATEMENT_kind kind;
113     struct function *function;
114     int contains_ret;
115     union {
116         struct expression *ret;
117         struct expression *wrt;
118         struct {
119             struct variable *variable;
120             struct expression *length;
```

```
121         } allocate;
122
123         struct {
124             struct variable *variable;
125             struct expression *expression;
126         } assignment;
127
128         struct {
129             struct expression *expression;
130             struct statement *statement1;
131             struct statement *statement2;
132         } ifthen;
133
134         struct {
135             struct expression *expression;
136             struct statement *statement;
137         } loop;
138
139         struct statement_list *list;
140     } val;
141
142 } statement;
143
144 typedef struct variable {
145     struct symbol_table*table;
146     struct symbol_type *stype;
147     int lineno;
148     char *id;
149     Var_kind kind;
150     union {
151         struct {
152             struct variable *var;
153             struct expression *exp;
154         } exp;
155         struct {
156             struct variable *var;
157             char *id;
158         } record;
159     } val;
160 } variable;
161
162 typedef struct expression {
163     struct symbol_table*table;
164     struct symbol_type *stype;
165     int lineno;
166     EXP_kind kind;
167     union {
168         struct {
169             struct expression *left;
170             struct expression *right;
171         } ops;
172         struct term *term;
173         struct expression *neg;
174     } val;
175
176 } expression;
177
178 typedef struct term {
179     struct symbol_table*table;
180     struct symbol_type *stype;
181     int lineno;
182     TERM_kind kind;
183     union {
```

```
184         int num;
185         struct expression *expression;
186         struct term *term_not;
187         struct variable *variable;
188         struct {
189             char *id;
190             struct act_list *list;
191         } list;
192     } val;
193 } term;
194 
195 typedef struct act_list {
196     struct symbol_table*table;
197     int lineno;
198     AL_kind kind;
199     struct exp_list *list;
200 } act_list;
201 
202 typedef struct exp_list {
203     struct symbol_table *table;
204     int lineno;
205     EL_kind kind;
206     struct expression *expression;
207     struct exp_list *list;
208 } exp_list;
209 
210 function *make_Func(head *h, body *b, tail *t);
211 
212 head *make_Head(char *id, par_decl_list *pdl, type *t);
213 
214 tail *make_Tail(char *id);
215 
216 type *make_Type_id(char *id);
217 type *make_Type_int();
218 type *make_Type_bool();
219 type *make_Type_array(type *t);
220 type *make_Type_record(var_decl_list *vdl);
221 
222 par_decl_list *make_PDL_list(var_decl_list *vdl);
223 par_decl_list *make_PDL_empty();
224 
225 var_decl_list *make_VDL_list(var_type *vt, var_decl_list *vdl);
226 var_decl_list *make_VDL_type(var_type *vt);
227 
228 var_type *make_VType_id(char *id, type *t);
229 
230 body *make_Body(decl_list *dl, statement_list *sl);
231 
232 decl_list *make_DL_list(declaration *d, decl_list *dl);
233 decl_list *make_DL_empty();
234 
235 declaration *make_Decl_type(char *id, type *t);
236 declaration *make_Decl_func(function *f);
237 declaration *make_Decl_list(var_decl_list *vdl);
238 
239 statement_list *make_SL_statement(statement *s);
240 statement_list *make_SL_list(statement *s, statement_list *sl);
241 
242 statement *make_STMT_ret(expression *e);
243 statement *make_STMT_wrt(expression *e);
244 statement *make_STMT_allocate_var(variable *v);
245 statement *make_STMT_allocate_length(variable *v, expression *e);
246 statement *make_STMT_assign(variable *v, expression *e);
```

```
247 statement *make_STMT_if(expression *e, statement *s);
248 statement *make_STMT_if_else(expression *e, statement *s1, statement *s2);
249 statement *make_STMT_while(expression *e, statement *s);
250 statement *make_STMT_list(statement_list *sl);
251
252 variable *make_Var_id(char *id);
253 variable *make_Var_exp(variable *var, expression *expression);
254 variable *make_Var_record(variable *var, char *id);
255
256 expression *make_EXP(EXP_kind kind, expression *left, expression *right);
257 expression *make_EXP_term(term *term);
258 expression *make_EXP_neg(expression *neg);
259
260 term *make_Term_num(int intconst);
261 term *make_Term_par(expression *expression);
262 term *make_Term_not(term *term);
263 term *make_Term_abs(expression *expression);
264 term *make_Term_boolean(int bool);
265 term *make_Term_null();
266 term *make_Term_variable(variable *var);
267 term *make_Term_list(char *id, act_list *list);
268
269 act_list *make_Act_list(exp_list *list);
270 act_list *make_Act_empty();
271
272 exp_list *make_ExpL_exp(expression *expression);
273 exp_list *make_ExpL_list(expression *expression, exp_list *list);
274
275 int check_Func(head *head, tail *tail);
276
277 #endif
```

**tree.c**

```
1  /**
2   * @brief
3   *
4   * @file tree.c
5   * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
6   * @date 2018-03-09
7   */
8  #include "memory.h"
9  #include "tree.h"
10 #include <stdio.h>
11
12 extern int lineno;
13
14 function *make_Func(head *h, body *b, tail *t) {
15     function *f;
16     f = NEW(function);
17     f->lineno = lineno;
18     f->head = h;
19     f->body = b;
20     f->tail = t;
21     return f;
22 }
23
24 head *make_Head(char *id, par_decl_list *pdl, type *t) {
25     head *h;
26     h = NEW(head);
27     h->lineno = lineno;
```

```
28      h->id = id;
29      h->list = pdl;
30      h->type = t;
31      return h;
32  }
33
34  tail *make_Tail(char *id) {
35      tail *t;
36      t = NEW(tail);
37      t->lineno = lineno;
38      t->id = id;
39      return t;
40  }
41
42  type *make_Type_id(char *id) {
43      type *t;
44      t = NEW(type);
45      t->lineno = lineno;
46      t->kind = type_ID;
47      t->val.id = id;
48      return t;
49  }
50
51  type *make_Type_int() {
52      type *t;
53      t = NEW(type);
54      t->lineno = lineno;
55      t->kind = type_INT;
56      return t;
57  }
58
59  type *make_Type_bool() {
60      type *t;
61      t = NEW(type);
62      t->lineno = lineno;
63      t->kind = type_BOOl;
64      return t;
65  }
66
67  type *make_Type_array(type *t1) {
68      type *t;
69      t = NEW(type);
70      t->lineno = lineno;
71      t->kind = type_ARRAY;
72      t->val.type = t1;
73      return t;
74  }
75
76  type *make_Type_record(var_decl_list *vdl) {
77      type *t;
78      t = NEW(type);
79      t->lineno = lineno;
80      t->kind = type_RECORD;
81      t->val.list = vdl;
82      return t;
83  }
84
85  par_decl_list *make_PDL_list(var_decl_list *vdl) {
86      par_decl_list *pdl;
87      pdl = NEW(par_decl_list);
88      pdl->lineno = lineno;
89      pdl->kind = pdl_LIST;
90      pdl->list = vdl;
```

```
91      return pdl;
92  }
93
94  par_decl_list *make_PDL_empty() {
95      par_decl_list *pdl;
96      pdl = NEW(par_decl_list);
97      pdl->lineno = lineno;
98      pdl->kind = pdl_EMPTY;
99      pdl->list = NULL;
100     return pdl;
101 }
102
103 var_decl_list *make_VDL_list(var_type *vt, var_decl_list *vdl1) {
104     var_decl_list *vdl;
105     vdl = NEW(var_decl_list);
106     vdl->lineno = lineno;
107     vdl->kind = vdl_LIST;
108     vdl->list = vdl1;
109     vdl->vartype = vt;
110     return vdl;
111 }
112
113 var_decl_list *make_VDL_type(var_type *vt) {
114     var_decl_list *vdl;
115     vdl = NEW(var_decl_list);
116     vdl->lineno = lineno;
117     vdl->kind = vdl_TYPE;
118     vdl->list = NULL;
119     vdl->vartype = vt;
120     return vdl;
121 }
122
123 var_type *make_VType_id(char *id, type *t) {
124     var_type *vt;
125     vt = NEW(var_type);
126     vt->lineno = lineno;
127     vt->id = id;
128     vt->type = t;
129     return vt;
130 }
131
132 body *make_Body(decl_list *dl, statement_list *sl) {
133     body *b;
134     b = NEW(body);
135     b->lineno = lineno;
136     b->d_list = dl;
137     b->s_list = sl;
138     return b;
139 }
140
141 decl_list *make_DL_list(declaration *d, decl_list *dl1) {
142     decl_list *dl;
143     dl = NEW(decl_list);
144     dl->lineno = lineno;
145     dl->kind = dl_LIST;
146     dl->decl = d;
147     dl->list = dl1;
148     return dl;
149 }
150
151 decl_list *make_DL_empty() {
152     decl_list *dl;
153     dl = NEW(decl_list);
```

```
154     dl->lineno = lineno;
155     dl->kind = dl_EMPTY;
156     dl->decl = NULL;
157     dl->list = NULL;
158     return dl;
159 }
160 
161 declaration *make_Decl_type(char *id, type *t) {
162     declaration *d;
163     d = NEW(declaration);
164     d->lineno = lineno;
165     d->kind = decl_TYPE;
166     d->val.type.id = id;
167     d->val.type.type = t;
168     return d;
169 }
170 
171 declaration *make_Decl_func(function *f) {
172     declaration *d;
173     d = NEW(declaration);
174     d->lineno = lineno;
175     d->kind = decl_FUNC;
176     d->val.function = f;
177     return d;
178 }
179 
180 declaration *make_Decl_list(var_decl_list *vdl) {
181     declaration *d;
182     d = NEW(declaration);
183     d->lineno = lineno;
184     d->kind = decl_VAR;
185     d->val.list = vdl;
186     return d;
187 }
188 
189 statement_list *make_SL_statement(statement *s) {
190     statement_list *sl;
191     sl = NEW(statement_list);
192     sl->lineno = lineno;
193     sl->kind = sl_STATEMENT;
194     sl->statement = s;
195     sl->list = NULL;
196     return sl;
197 }
198 
199 statement_list *make_SL_list(statement *s, statement_list *sl1) {
200     statement_list *sl;
201     sl = NEW(statement_list);
202     sl->lineno = lineno;
203     sl->kind = sl_LIST;
204     sl->statement = s;
205     sl->list = sl1;
206     return sl;
207 }
208 
209 statement *make_STMT_ret(expression *e) {
210     statement *s;
211     s = NEW(statement);
212     s->lineno = lineno;
213     s->kind = statement_RETURN;
214     s->val.ret = e;
215     return s;
216 }
```

```
217
218 statement *make_STMT_wrt(expression *e) {
219     statement *s;
220     s = NEW(statement);
221     s->lineno = lineno;
222     s->kind = statement_WRITE;
223     s->val.wrt = e;
224     return s;
225 }
226
227 statement *make_STMT_allocate_var(variable *v) {
228     statement *s;
229     s = NEW(statement);
230     s->lineno = lineno;
231     s->kind = statement_ALLOCATE;
232     s->val.allocate.variable = v;
233     s->val.allocate.length = NULL;
234     return s;
235 }
236
237 statement *make_STMT_allocate_length(variable *v, expression *e) {
238     statement *s;
239     s = NEW(statement);
240     s->lineno = lineno;
241     s->kind = statement_ALLOCATE_LENGTH;
242     s->val.allocate.variable = v;
243     s->val.allocate.length = e;
244     return s;
245 }
246
247 statement *make_STMT_assign(variable *v, expression *e) {
248     statement *s;
249     s = NEW(statement);
250     s->lineno = lineno;
251     s->kind = statement_ASSIGNMENT;
252     s->val.assignment.variable = v;
253     s->val.assignment.expression = e;
254     return s;
255 }
256
257 statement *make_STMT_if(expression *e, statement *s1) {
258     statement *s;
259     s = NEW(statement);
260     s->lineno = lineno;
261     s->kind = statement_IF;
262     s->val.ifthen.expression = e;
263     s->val.ifthen.statement1 = s1;
264     s->val.ifthen.statement2 = NULL;
265     return s;
266 }
267
268 statement *make_STMT_if_else(expression *e, statement *s1, statement *s2) {
269     statement *s;
270     s = NEW(statement);
271     s->lineno = lineno;
272     s->kind = statement_IF_ELSE;
273     s->val.ifthen.expression = e;
274     s->val.ifthen.statement1 = s1;
275     s->val.ifthen.statement2 = s2;
276     return s;
277 }
278
279 statement *make_STMT_while(expression *e, statement *s1) {
```

```
280     statement *s;
281     s = NEW(statement);
282     s->lineno = lineno;
283     s->kind = statement_WHILE;
284     s->val.loop.expression = e;
285     s->val.loop.statement = s1;
286     return s;
287 }
288 
289 statement *make_STMT_list(statement_list *sl) {
290     statement *s;
291     s = NEW(statement);
292     s->lineno = lineno;
293     s->kind = statement_LIST;
294     s->val.list = sl;
295     return s;
296 }
297 
298 expression *make_EXP(EXP_kind kind, expression *left, expression *right) {
299     expression *e;
300     e = NEW(expression);
301     e->lineno = lineno;
302     e->kind = kind;
303     e->val.ops.left = left;
304     e->val.ops.right = right;
305     return e;
306 }
307 
308 expression *make_EXP_term(term *term) {
309     expression *e;
310     e = NEW(expression);
311     e->lineno = lineno;
312     e->kind = exp_TERM;
313     e->val.term = term;
314     return e;
315 }
316 
317 //Negation of x is the same as 0-x, so we make a minus node
318 expression *make_EXP_neg(expression *neg) {
319 
320     expression *zero = make_EXP_term(make_Term_num(0));
321 
322     expression *minus = make_EXP(exp_MIN, zero, neg);
323 
324     return minus;
325 }
326 
327 term *make_Term_num(int intconst) {
328     term *t;
329     t = NEW(term);
330     t->lineno = lineno;
331     t->kind = term_NUM;
332     t->val.num = intconst;
333     return t;
334 }
335 
336 term *make_Term_par(expression *expression) {
337     term *t;
338     t = NEW(term);
339     t->lineno = lineno;
340     t->kind = term_PAR;
341     t->val.expression = expression;
342     return t;
```

```
343 }
344
345 term *make_Term_not(term *term) {
346     struct term *t;
347     t = NEW(struct term);
348     t->lineno = lineno;
349     t->kind = term_NOT;
350     t->val.term_not = term;
351     return t;
352 }
353
354 term *make_Term_abs(expression *expression) {
355     term *t;
356     t = NEW(term);
357     t->lineno = lineno;
358     t->kind = term_ABS;
359     t->val.expression = expression;
360     return t;
361 }
362
363 term *make_Term_boolean(int bool) {
364     term *t;
365     t = NEW(term);
366     t->lineno = lineno;
367     if (bool == 1) {
368         t->kind = term_TRUE;
369         return t;
370     }
371     t->kind = term_FALSE;
372     return t;
373 }
374
375 term *make_Term_null() {
376     term *t;
377     t = NEW(term);
378     t->lineno = lineno;
379     t->kind = term_NULL;
380     return t;
381 }
382
383 term *make_Term_variable(variable *var) {
384     term *t;
385     t = NEW(term);
386     t->lineno = lineno;
387     t->kind = term_VAR;
388     t->val.variable = var;
389     return t;
390 }
391
392 term *make_Term_list(char *id, act_list *list) {
393     term *t;
394     t = NEW(term);
395     t->lineno = lineno;
396     t->kind = term_LIST;
397     t->val.list.id = id;
398     t->val.list.list = list;
399     return t;
400 }
401
402 act_list *make_Act_list(exp_list *list) {
403     act_list *al;
404     al = NEW(act_list);
405     al->lineno = lineno;
```

```
406     al->kind = al_LIST;
407     al->list = list;
408     return al;
409 }
410 act_list *make_Act_empty() {
411     act_list *al;
412     al = NEW(act_list);
413     al->lineno = lineno;
414     al->kind = al_EMPTY;
415     al->list = NULL;
416     return al;
417 }
418 
419 exp_list *make_ExpL_exp(expression *expression) {
420     exp_list *el;
421     el = NEW(exp_list);
422     el->lineno = lineno;
423     el->kind = el_EXP;
424     el->expression = expression;
425     el->list = NULL;
426     return el;
427 }
428 
429 exp_list *make_ExpL_list(expression *expression, exp_list *list) {
430     exp_list *el;
431     el = NEW(exp_list);
432     el->lineno = lineno;
433     el->kind = el_LIST;
434     el->expression = expression;
435     el->list = list;
436     return el;
437 }
438 
439 variable *make_Var_id(char *id) {
440     variable *v;
441     v = NEW(variable);
442     v->lineno = lineno;
443     v->kind = var_ID;
444     v->id = id;
445     return v;
446 }
447 
448 variable *make_Var_exp(variable *var, expression *expression) {
449     variable *v;
450     v = NEW(variable);
451     v->lineno = lineno;
452     v->kind = var_EXP;
453     v->val.exp.var = var;
454     v->val.exp.exp = expression;
455     return v;
456 }
457 
458 variable *make_Var_record(variable *var, char *id) {
459     variable *v;
460     v = NEW(variable);
461     v->lineno = lineno;
462     v->kind = var_RECORD;
463     v->val.record.var = var;
464     v->val.record.id = id;
465     return v;
466 }
467 
468 int check_Func(head *head, tail *tail) {
```

```
469     if (strcmp(head->id, tail->id) == 0) {
470         return 0;
471     }
472     return 1;
473 }
```

**pretty.h**

```
1  #ifndef __pretty_h
2  #define __pretty_h
3
4  #include "tree.h"
5
6  extern int types;
7
8  void prettyProgram(body *body);
9
10 void prettyFunc(function *f);
11
12 void prettyHead(head *h);
13
14 void prettyTail(tail *t);
15
16 void prettyType(type *t);
17
18 void prettyPDL(par_decl_list *pdl);
19
20 void prettyVDL(var_decl_list *vdl);
21
22 void prettyVT(var_type *vt);
23
24 void prettyBody(body *b);
25
26 void prettyDL(decl_list *dl);
27
28 void prettyDecl(declaration *d);
29
30 void prettySL(statement_list *sl);
31
32 void prettySTMT(statement *s);
33
34 void prettyVar(variable *v);
35
36 void prettyEXP(expression *e);
37
38 void prettyTerm(term *t);
39
40 void prettyAL(act_list *al);
41
42 void prettyEL(exp_list *el);
43
44 void indent();
45
46 void prettySymbol(symbol_table *table, char *id, int line);
47
48 void prettyStype(symbol_type *stype, int line);
49
50 #endif
```

**pretty.c**

```c
/**
 * @brief
 *
 * @file pretty.c
 * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
 * @date 2018-03-09
 */
#include <stdio.h>
#include "pretty.h"
#include "tree.h"
#include "symbol.h"
#include "error.h"

int indent_depth;
int exp_depth;
int types;
int inside_par;

void prettyProgram(body *body){
    indent_depth = 0;
    exp_depth = 0;
    inside_par = 0;
    prettyBody(body);
}

void prettyFunc(function *f) {
    prettyHead(f->head);
    indent_depth++;
    prettyBody(f->body);
    indent_depth--;
    prettyTail(f->tail);
}

void prettyHead(head *h) {
    printf("func %s(", h->id);
    prettyPDL(h->list);
    printf(") : ");
    prettyType(h->type);
    printf("\n");
}

void prettyTail(tail *t) {
    indent();
    printf("end %s", t->id);
    if(types){
        prettySymbol(t->table, t->id, t->lineno);
    }
    printf("\n");
}

void prettyType(type *t) {
    switch (t->kind) {
    case type_ID:
        printf("%s : ", t->val.id);
        if (types){
            prettyStype(t->stype, t->lineno);
        }
        break;

    case type_INT:
        printf("int");
```

```
62          break;
63
64      case type_BOOl:
65          printf("bool");
66          break;
67
68      case type_ARRAY:
69          printf("array of ");
70          prettyType(t->val.type);
71          break;
72
73      case type_RECORD:
74          printf("record of { ");
75          prettyVDL(t->val.list);
76          printf(" }");
77          break;
78      }
79  }
80
81  void prettyPDL(par_decl_list *pdl) {
82
83      if (pdl->kind == pdl_LIST){
84          prettyVDL(pdl->list);
85      }
86  }
87
88  void prettyVDL(var_decl_list *vdl) {
89      switch (vdl->kind) {
90          case vdl_LIST:
91              prettyVT(vdl->vartype);
92              printf(", ");
93              prettyVDL(vdl->list);
94              break;
95
96          case vdl_TYPE:
97              prettyVT(vdl->vartype);
98              break;
99      }
100 }
101
102 void prettyVT(var_type *vt) {
103     printf("%s : ", vt->id);
104     prettyType(vt->type);
105 }
106
107 void prettyBody(body *b) {
108     prettyDL(b->d_list);
109     prettySL(b->s_list);
110 }
111
112 void prettyDL(decl_list *dl) {
113     switch (dl->kind) {
114     case dl_LIST:
115         prettyDecl(dl->decl);
116         prettyDL(dl->list);
117         break;
118
119     case dl_EMPTY:
120         break;
121     }
122 }
123
124 void prettyDecl(declaration *d) {
```

```
125     indent();
126     switch (d->kind) {
127     case decl_TYPE:
128         printf("type %s = ", d->val.type.id);
129         prettyType(d->val.type.type);
130         printf(";\n");
131         break;
132 
133     case decl_FUNC:
134         prettyFunc(d->val.function);
135         break;
136 
137     case decl_VAR:
138         printf("var ");
139         prettyVDL(d->val.list);
140         printf(";\n");
141         break;
142     }
143 }
144 
145 void prettySL(statement_list *sl) {
146     switch (sl->kind) {
147     case sl_STATEMENT:
148         prettySTMT(sl->statement);
149         break;
150 
151     case sl_LIST:
152         prettySTMT(sl->statement);
153         prettySL(sl->list);
154         break;
155     }
156 }
157 
158 void prettySTMT(statement *s) {
159 
160     if (s->kind != statement_LIST) {
161         indent();
162     }
163 
164     switch (s->kind) {
165     case statement_RETURN:
166         printf("return ");
167         prettyEXP(s->val.ret);
168         printf(";\n");
169         break;
170 
171     case statement_WRITE:
172         printf("write ");
173         prettyEXP(s->val.wrt);
174         printf(";\n");
175         break;
176 
177     case statement_ALLOCATE:
178         printf("allocate ");
179         prettyVar(s->val.allocate.variable);
180         printf(";\n");
181         break;
182 
183     case statement_ALLOCATE_LENGTH:
184         printf("allocate ");
185         prettyVar(s->val.allocate.variable);
186         printf(" of length ");
187         prettyEXP(s->val.allocate.length);
```

```
188         printf(";\n");
189         break;
190 
191     case statement_ASSIGNMENT:
192         prettyVar(s->val.assignment.variable);
193         printf(" = ");
194         prettyEXP(s->val.assignment.expression);
195         printf(";\n");
196         break;
197 
198     case statement_IF:
199         printf("if (");
200         prettyEXP(s->val.ifthen.expression);
201         printf(") then\n");
202         indent_depth++;
203         prettySTMT(s->val.ifthen.statement1);
204         indent_depth--;
205         break;
206 
207     case statement_IF_ELSE:
208         printf("if (");
209         prettyEXP(s->val.ifthen.expression);
210         printf(") then\n");
211         indent_depth++;
212         prettySTMT(s->val.ifthen.statement1);
213         indent_depth--;
214         indent();
215         printf(" else\n");
216         indent_depth++;
217         prettySTMT(s->val.ifthen.statement2);
218         indent_depth--;
219         break;
220 
221     case statement_WHILE:
222         printf("while (");
223         prettyEXP(s->val.loop.expression);
224         printf(") do\n");
225         prettySTMT(s->val.loop.statement);
226         break;
227 
228     case statement_LIST:
229         indent();
230         printf("{\n");
231         indent_depth++;
232         prettySL(s->val.list);
233         indent_depth--;
234         indent();
235         printf("}\n");
236         break;
237     }
238 }
239 
240 void prettyVar(variable *v) {
241     switch (v->kind) {
242 
243     case var_ID:
244         printf("%s", v->id);
245         break;
246 
247     case var_EXP:
248         prettyVar(v->val.exp.var);
249         printf("[");
250         prettyEXP(v->val.exp.exp);
```

```
251         printf("]");
252         break;
253
254     case var_RECORD:
255         prettyVar(v->val.record.var);
256         printf(".%s", v->val.record.id);
257         break;
258     }
259 }
260
261 void prettyEXP(expression *e) {
262     exp_depth++;
263
264     if (e->kind == exp_TERM){
265         prettyTerm(e->val.term);
266         return;
267     }
268
269     if(exp_depth > 1 && inside_par == 0){
270         printf("(");
271     }
272     switch (e->kind) {
273
274     case exp_MULT:
275         prettyEXP(e->val.ops.left);
276         printf("*");
277         prettyEXP(e->val.ops.right);
278         break;
279
280     case exp_DIV:
281         prettyEXP(e->val.ops.left);
282         printf("/");
283         prettyEXP(e->val.ops.right);
284         break;
285
286     case exp_PLUS:
287         prettyEXP(e->val.ops.left);
288         printf("+");
289         prettyEXP(e->val.ops.right);
290         break;
291
292     case exp_MIN:
293         prettyEXP(e->val.ops.left);
294         printf("-");
295         prettyEXP(e->val.ops.right);
296         break;
297
298     case exp_EQ:
299         prettyEXP(e->val.ops.left);
300         printf(" == ");
301         prettyEXP(e->val.ops.right);
302         break;
303
304     case exp_NEQ:
305         prettyEXP(e->val.ops.left);
306         printf(" != ");
307         prettyEXP(e->val.ops.right);
308         break;
309
310     case exp_GT:
311         prettyEXP(e->val.ops.left);
312         printf(" > ");
313         prettyEXP(e->val.ops.right);
```

```
314         break;
315
316     case exp_LT:
317         prettyEXP(e->val.ops.left);
318         printf(" < ");
319         prettyEXP(e->val.ops.right);
320         break;
321
322     case exp_GEQ:
323         prettyEXP(e->val.ops.left);
324         printf(" >= ");
325         prettyEXP(e->val.ops.right);
326         break;
327
328     case exp_LEQ:
329         prettyEXP(e->val.ops.left);
330         printf(" <= ");
331         prettyEXP(e->val.ops.right);
332         break;
333
334     case exp_AND:
335         prettyEXP(e->val.ops.left);
336         printf(" && ");
337         prettyEXP(e->val.ops.right);
338         break;
339
340     case exp_OR:
341         prettyEXP(e->val.ops.left);
342         printf(" || ");
343         prettyEXP(e->val.ops.right);
344         break;
345
346     }
347     if(exp_depth > 1 && inside_par == 0){
348         printf(")");
349
350     }
351     exp_depth--;
352
353     if (types){
354        // printf("\nCalling printStype in expression");
355         printf( " : ");
356         prettyStype(e->stype, e->lineno);
357     }
358
359 }
360
361 void prettyTerm(term *t) {
362     switch (t->kind) {
363
364     case term_VAR:
365         prettyVar(t->val.variable);
366         break;
367
368     case term_LIST:
369         printf("%s(", t->val.list.id);
370         prettyAL(t->val.list.list);
371         printf(")");
372         break;
373
374     case term_PAR:
375         if (exp_depth > 1){
376             printf("(");
```

```
377         }
378         inside_par = 1;
379         prettyEXP(t->val.expression);
380         inside_par = 0;
381 
382         if (exp_depth > 1){
383             printf("(");
384         }
385         break;
386 
387     case term_NOT:
388         printf("!");
389         prettyTerm(t->val.term_not);
390         break;
391 
392     case term_ABS:
393         printf("|");
394         prettyEXP(t->val.expression);
395         printf("|");
396         break;
397 
398     case term_TRUE:
399         printf("true");
400         break;
401 
402     case term_FALSE:
403         printf("false");
404         break;
405 
406     case term_NULL:
407         printf("null");
408         break;
409 
410     case term_NUM:
411         printf("%i", t->val.num);
412         break;
413     }
414 
415     if (types){
416         //printf("\nCalling printStype in term");
417                 // Haps
418         printf(" : ");
419         prettyStype(t->stype, t->lineno);
420     }
421 
422 }
423 
424 void prettyAL(act_list *al) {
425     switch (al->kind) {
426 
427     case al_LIST:
428         prettyEL(al->list);
429         break;
430 
431     case al_EMPTY:
432         break;
433     }
434 }
435 
436 void prettyEL(exp_list *el) {
437     switch (el->kind) {
438 
439     case el_EXP:
```

```
440         prettyEXP(el->expression);
441         break;
442 
443     case el_LIST:
444         prettyEXP(el->expression);
445         printf(", ");
446         prettyEL(el->list);
447         break;
448     }
449 }
450 
451 void indent() {
452 
453     int spaces = 0;
454     while (spaces < (indent_depth * 4)) {
455         printf(" ");
456         spaces++;
457     }
458 }
459 
460 void prettySymbol(symbol_table *table, char *id, int line){
461 
462     SYMBOL *s;
463     s = get_symbol(table, id);
464     if (s == NULL || s->stype == NULL){
465         print_error("Symbol is not recognized", 0, line);
466     }
467     printf(" : ");
468     prettyStype(s->stype, line);
469 }
470 
471 void prettyStype(symbol_type *stype, int line){
472    // printf("\nPrintSType of type: %d: ", stype->type);
473     if (stype->printed){
474         return;
475     }
476     stype->printed = 1;
477     switch(stype->type){
478 
479         case (symbol_ID):
480             prettyType(stype->val.id_type);
481             break;
482 
483         case (symbol_INT):
484             printf("int");
485             break;
486 
487         case (symbol_BOOL):
488             printf("boolean");
489             break;
490 
491         case (symbol_RECORD):
492             printf("record of {");
493             prettyVDL(stype->val.record_type);
494             printf("}");
495             break;
496 
497         case (symbol_ARRAY):
498             printf("array[");
499             prettyStype(stype->val.array_type->stype, line);
500             printf("]");
501             break;
502 
```

```
503         case (symbol_FUNCTION):
504             printf("function(");
505             prettyPDL(stype->val.func_type.pdl);
506             printf(") : ");
507             prettyStype(stype->val.func_type.ret_type->stype, line);
508             break;
509 
510         case (symbol_NULL):
511             printf("NULL");
512             break;
513 
514         // Should never happen
515         case (symbol_UNKNOWN):
516             printf("unknown");
517             print_error("Unknown symbol type", 0, line);
518             break;
519 
520     }
521     stype->printed = 0;
522 
523 }
```

### typechecker.h

```
1  #ifndef __typechecker_h
2  #define __typechecker_h
3  #include "tree.h"
4  
5  
6  int typecheck(body *program);
7  
8  
9  
10 
11 #endif
```

### typechecker.c

```
1  /**
2   * @brief
3   *
4   * @file typechecker.c
5   * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
6   * @date 2018-03-09
7   */
8  
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include "debug.h"
12 #include "check.h"
13 #include "typechecker.h"
14 #include "error.h"
15 #include "symbol.h"
16 #include "setup.h"
17 #include "pickup.h"
18 
19 int typecheck(body *program) {
20 
21     symbol_table *table;
22     table = init_symbol_table();
```

```
#if debugflag > 1
    printf("Starting Setup\n");
#endif
    setup_body(program, table);
#if debugflag > 1
    printf("Starting pickup\n");
#endif
    pickup_body(program);
#if debugflag > 1
    printf("Starting check\n");
#endif
    check_body(program);

    return 0;
}
```

**weeder.h**

```
#ifndef __weeder_h
#define __weeder_h

#include "tree.h"

body *weeder_init(body *program);

body *weeder(body *body);

body *weed_body(body *body);

function *weed_function(function *func);

head *weed_head(head *head);

tail *weed_tail(tail *tail);

type *weed_type(type *type);

par_decl_list *weed_pdl(par_decl_list *pdl);

var_decl_list *weed_vdl(var_decl_list *vdl);

var_type *weed_vtype(var_type *vtype);

decl_list *weed_dlist(decl_list *dlist);

declaration *weed_decl(declaration *decl);

statement_list *weed_slist(statement_list *slist);

statement *weed_stmt(statement *stmt);

variable *weed_variable(variable *variable);

expression *weed_expression(expression *expression);

term *weed_term(term *term);

act_list *weed_alist(act_list *alist);

exp_list *weed_elist(exp_list *elist);

```

```
44
45
46
47 #endif
```

**weeder.c**

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include "tree.h"
5  #include "weeder.h"
6  #include "error.h"
7  #include "stack.h"
8
9  body *theprogram;
10
11 struct stack *function_stack;
12
13 /**
14  * What we want to weed (TO BE EXPANDED):
15  *      [IF] statements:
16  *          true && term   - should be term
17  *          false && term  - should be false
18  *          true || term   - should be true
19  *          false || term  - should be true
20  *          break/continue - error (for now, break could possibly be implemented)
21  *
22  *      [CONSTANTS]
23  *          F.x. 5*2       - should be 10, can be calculated during compile time
24  *
25  *      [FUNCTIONS]
26  *          no return      - error
27  *
28  *
29  */
30
31
32 body *weeder_init(body *program){
33
34     function_stack = init_stack();
35     program = weed_body(program);
36     return program;
37
38
39 }
40
41 body *weed_body(body *body){
42     ////printf("Weeding body\n");
43     body->d_list = weed_dlist(body->d_list);
44     body->s_list = weed_slist(body->s_list);
45     return body;
46 }
47
48 function *weed_function(function *func){
49
50     stack_push(function_stack, func);
51
52     func->body = weed_body(func->body);
53     func->head = weed_head(func->head);
54
```

```
55      stack_pop(function_stack);
56
57      return func;
58
59  }
60
61  head *weed_head(head *head){
62      ////printf("Weeding head\n");
63      head->list = weed_pdl(head->list);
64      head->type = weed_type(head->type);
65      return head;
66
67  }
68
69  type *weed_type(type *type){
70
71      ////printf("Weeding type\n");
72      switch (type->kind){
73          case (type_ARRAY):
74              type->val.type = weed_type(type->val.type);
75              break;
76
77          case (type_RECORD):
78              type->val.list = weed_vdl(type->val.list);
79              break;
80
81          default:
82              break;
83      }
84      return type;
85
86  }
87
88  par_decl_list *weed_pdl(par_decl_list *pdl){
89      ////printf("Weeding pdl\n");
90      if (pdl->kind == pdl_EMPTY){
91          return pdl;
92      }
93
94      pdl->list = weed_vdl(pdl->list);
95      return pdl;
96  }
97
98  var_decl_list *weed_vdl(var_decl_list *vdl){
99      switch (vdl->kind){
100
101         case (vdl_TYPE):
102             vdl->vartype = weed_vtype(vdl->vartype);
103             break;
104
105         case (vdl_LIST):
106             vdl->list = weed_vdl(vdl->list);
107             break;
108
109     }
110     return vdl;
111
112 }
113
114 var_type *weed_vtype(var_type *vtype){
115     ////printf("Weeding vtype\n");
116
117     vtype->type = weed_type(vtype->type);
```

```
118     return vtype;
119 
120 }
121 
122 decl_list *weed_dlist(decl_list *dlist){
123     ////printf("Weeding dlist\n");
124     if (dlist->kind == dl_EMPTY){
125         //Nothing to do
126         return dlist;
127     }
128     dlist->list = weed_dlist(dlist->list);
129     dlist->decl = weed_decl(dlist->decl);
130     return dlist;
131 
132 }
133 
134 declaration *weed_decl(declaration *decl){
135     ////printf("Weeding decl\n");
136     switch (decl->kind){
137 
138         case (decl_TYPE):
139             decl->val.type.type = weed_type(decl->val.type.type);
140             break;
141 
142         case (decl_FUNC):
143             decl->val.function = weed_function(decl->val.function);
144             break;
145 
146         case (decl_VAR):
147             decl->val.list = weed_vdl(decl->val.list);
148             break;
149 
150     }
151     return decl;
152 
153 }
154 
155 statement_list *weed_slist(statement_list *slist){
156     //printf("Weeding slist\n");
157 
158     if (slist == NULL){
159         return NULL;
160     }
161 
162     slist->statement = weed_stmt(slist->statement);
163     if (slist->kind == sl_LIST){
164         slist->list = weed_slist(slist->list);
165     }
166 
167     if (slist->statement == NULL){
168         return slist->list;
169     }
170 
171 
172 
173     return slist;
174 
175 }
176 
177 statement *weed_stmt(statement *stmt){
178     //printf("Weeding statement, kind: %d\n", stmt->kind);
179 
180     struct function *f;
```

```
181
182     switch(stmt->kind){
183
184         case (statement_RETURN):
185             //printf("Statement return\n");
186             //printf("Before weeding expression:\n");
187             //prettyEXP(stmt->val.ret);
188             //printf("\n\n");
189             stmt->val.ret = weed_expression(stmt->val.ret);
190             //printf("After weeding expression:\n");
191             //prettyEXP(stmt->val.ret);
192             //printf("\n\n");
193
194             f = stack_read(function_stack);
195             //printf("Function this return belongs to: %s", f->head->id);
196             if (f == NULL){
197                 print_error("Return outside of function", 0, stmt->lineno);
198             }
199             stmt->function = f;
200             stmt->contains_ret = 1;
201             break;
202
203         case (statement_WRITE):
204             //printf("Statement write\n");
205             stmt->val.wrt = weed_expression(stmt->val.wrt);
206             break;
207
208         case (statement_ALLOCATE):
209             //printf("Statement allocate\n");
210             stmt->val.allocate.variable = weed_variable(stmt->val.allocate.variable);
211             break;
212
213         case (statement_ALLOCATE_LENGTH):
214             //printf("Statement allocate length\n");
215             stmt->val.allocate.variable = weed_variable(stmt->val.allocate.variable);
216             stmt->val.allocate.length = weed_expression(stmt->val.allocate.length);
217             break;
218
219         case (statement_ASSIGNMENT):
220             //printf("Statement assignment\n");
221             stmt->val.assignment.variable = weed_variable(stmt->val.assignment.variable);
222             stmt->val.assignment.expression = weed_expression(stmt->val.assignment.expression);
223             break;
224
225         case (statement_IF):
226             //printf("Statement if\n");
227             stmt->val.ifthen.expression = weed_expression(stmt->val.ifthen.expression);
228             stmt->val.ifthen.statement1 = weed_stmt(stmt->val.ifthen.statement1);
229
230             // Check if expression is always true/false
231             if (stmt->val.ifthen.expression->kind == exp_TERM){
232                 if (stmt->val.ifthen.expression->val.term->kind == term_FALSE){
233                     return NULL;
234                 }
235                 stmt = stmt->val.ifthen.statement1;
236             }
237             break;
238
239         case (statement_IF_ELSE):
240             //printf("Statement if else\n");
241             stmt->val.ifthen.expression = weed_expression(stmt->val.ifthen.expression);
242             stmt->val.ifthen.statement1 = weed_stmt(stmt->val.ifthen.statement1);
243             //printf("Weeding statement 2\n");
```

```
244             stmt->val.ifthen.statement2 = weed_stmt(stmt->val.ifthen.statement2);
245             //printf("Done weeding statement 2\n");
246 
247             if (stmt->val.ifthen.statement1->contains_ret && stmt->val.ifthen.statement2->
                    contains_ret){
248                 stmt->contains_ret = 1;
249             }
250 
251             // Check if expression is always true/false
252             if (stmt->val.ifthen.expression->kind == exp_TERM){
253                 if (stmt->val.ifthen.expression->val.term->kind == term_FALSE){
254                     stmt = stmt->val.ifthen.statement2;
255                 }
256                 stmt = stmt->val.ifthen.statement1;
257             }
258 
259             break;
260 
261         case (statement_WHILE):
262             //printf("Statement while\n");
263             stmt->val.loop.expression = weed_expression(stmt->val.loop.expression);
264             stmt->val.loop.statement = weed_stmt(stmt->val.loop.statement);
265             if (stmt->val.loop.statement == NULL){
266                 return NULL;
267             }
268             break;
269 
270         case (statement_LIST):
271             //printf("Statement list\n");
272             stmt->val.list = weed_slist(stmt->val.list);
273             if (stmt->val.list == NULL){
274                 return NULL;
275             }
276             stmt->contains_ret = stmt->val.list->statement->contains_ret;
277             break;
278     }
279 
280     //printf("\n");
281     //prettySTMT(stmt);
282     //printf("\n\n");
283 
284     return stmt;
285 
286 }
287 
288 variable *weed_variable(variable *variable){
289     //printf("Weeding variable, kind: %d\n", variable->kind);
290     switch (variable->kind){
291 
292         case (var_ID):
293             //printf("ID of variable: %s\n", variable->id);
294             break;
295 
296 
297         case (var_EXP):
298             variable->val.exp.exp = weed_expression(variable->val.exp.exp);
299             variable->val.exp.var = weed_variable(variable->val.exp.var);
300             break;
301 
302         case (var_RECORD):
303             variable->val.record.var = weed_variable(variable->val.exp.var);
304             break;
305 
```

```
306         default:
307             break;
308 
309     }
310     return variable;
311 
312 
313 }
314 
315 expression *weed_expression(expression *expression){
316     //printf("Weeding expression, kind: %d\n", expression->kind);
317 
318     struct expression *left_exp;
319     struct expression *right_exp;
320 
321     struct term *left_term;
322     struct term *right_term;
323     struct term *temp;
324 
325     temp = NULL;
326 
327 
328     if (expression->kind == exp_TERM){
329         //printf("Weeding single term of kind: %d, expression kind: %d\n", expression->val.term
                ->kind, expression->kind);
330         expression->val.term = weed_term(expression->val.term);
331         //printf("New term kind: %d\n", expression->val.term->kind);
332         return expression;
333     }
334 
335     expression->val.ops.left = weed_expression(expression->val.ops.left);
336     expression->val.ops.right = weed_expression(expression->val.ops.right);
337 
338     left_exp = expression->val.ops.left;
339     right_exp = expression->val.ops.right;
340 
341     if ((left_exp->kind == exp_TERM) && (right_exp->kind == exp_TERM)){
342         left_term = left_exp->val.term;
343         right_term = right_exp->val.term;
344 
345         if ((left_term->kind == term_NUM) && (right_term->kind == term_NUM)){
346             //We have an expression with two constants
347             switch(expression->kind){
348 
349                 case (exp_MULT):
350                     temp = make_Term_num(left_term->val.num * right_term->val.num);
351                     break;
352 
353                 case (exp_DIV):
354                     if(right_term->val.num == 0){
355                         print_error("Division by 0 error", 1, right_term->lineno);
356                     }
357                     temp = make_Term_num(left_term->val.num / right_term->val.num);
358                     break;
359 
360                 case (exp_PLUS):
361                     temp = make_Term_num(left_term->val.num + right_term->val.num);
362                     break;
363 
364                 case (exp_MIN):
365                     temp = make_Term_num(left_term->val.num - right_term->val.num);
366                     break;
367 
```

```
368                 case (exp_EQ):
369                     if (left_term->val.num == right_term->val.num){
370                         temp = make_Term_boolean(1);
371                     } else {
372                         temp = make_Term_boolean(0);
373                     }
374                     break;
375 
376                 case (exp_NEQ):
377                     if (left_term->val.num != right_term->val.num){
378                         temp = make_Term_boolean(1);
379                     } else {
380                         temp = make_Term_boolean(0);
381                     }
382                     break;
383 
384                 case (exp_GT):
385                     if (left_term->val.num > right_term->val.num){
386                         temp = make_Term_boolean(1);
387                     } else {
388                         temp = make_Term_boolean(0);
389                     }
390                     break;
391 
392                 case (exp_LT):
393                     if (left_term->val.num < right_term->val.num){
394                         temp = make_Term_boolean(1);
395                     } else {
396                         temp = make_Term_boolean(0);
397                     }
398                     break;
399 
400                 case (exp_GEQ):
401                     if (left_term->val.num >= right_term->val.num){
402                         temp = make_Term_boolean(1);
403                     } else {
404                         temp = make_Term_boolean(0);
405                     }
406                     break;
407 
408                 case (exp_LEQ):
409                     if (left_term->val.num <= right_term->val.num){
410                         temp = make_Term_boolean(1);
411                     } else {
412                         temp = make_Term_boolean(0);
413                     }
414                     break;
415 
416             }
417         }
418 
419         //Check for boolean expression
420         switch(expression->kind){
421 
422             case (exp_AND):
423                 if ((left_term->kind == term_FALSE) || (right_term->kind == term_FALSE)){
424                     temp = make_Term_boolean(0);
425                 }
426 
427                 if ((left_term->kind == term_TRUE) && (right_term->kind == term_TRUE)){
428                     temp = make_Term_boolean(1);
429                 }
430 
```

```
431                 break;
432
433             case (exp_OR):
434                 if ((left_term->kind == term_TRUE) || (right_term->kind == term_TRUE)){
435                     temp = make_Term_boolean(1);
436                 }
437
438                 if ((left_term->kind == term_FALSE) && (right_term->kind == term_FALSE)){
439                     temp = make_Term_boolean(0);
440                 }
441                 break;
442
443             default:
444                 break;
445         }
446
447         //TODO Optimize this please, to many comparisons I think, or maybe put advanced patterns
                into a function for itself?
448         if (temp == NULL){
449
450             if (expression->kind == exp_AND){
451
452                 //Advanced patterns
453
454                 if (left_exp->kind == exp_TERM){
455                     if (left_exp->val.term->kind == term_TRUE){
456                         return expression->val.ops.right;
457                     }
458                     if (left_exp->val.term->kind == term_FALSE){
459                         temp = make_Term_boolean(1);
460                     }
461                 }
462
463                 if (right_exp->kind == exp_TERM){
464                     if (right_exp->val.term->kind == term_TRUE){
465                         return expression->val.ops.left;
466                     }
467                     if (right_exp->val.term->kind == term_FALSE){
468                         temp = make_Term_boolean(1);
469                     }
470                 }
471
472             }
473
474             if (expression->kind == exp_OR){
475
476                 //Advanced patterns
477
478                 if (left_exp->kind == exp_TERM){
479                     if (left_exp->val.term->kind == term_FALSE){
480                         return expression->val.ops.right;
481                     }
482                     if (left_exp->val.term->kind == term_TRUE){
483                         temp = make_Term_boolean(1);
484                     }
485                 }
486
487                 if (right_exp->kind == exp_TERM){
488                     if (right_exp->val.term->kind == term_FALSE){
489                         return expression->val.ops.left;
490                     }
491                     if (right_exp->val.term->kind == term_TRUE){
492                         temp = make_Term_boolean(1);
```

```
493                     }
494                 }
495             }
496         }
497     }
498
499     if (temp != NULL){
500         //We reduced something
501         //printf("Reduced something\n");
502         expression->kind = exp_TERM;
503         expression->val.term = temp;
504         //printf("Done with expression, new kind: %d\n", expression->val.term->kind);
505     }
506
507     return expression;
508 }
509
510 term *weed_term(term *term){
511     //printf("Weeding term, kind: %d\n", term->kind);
512     struct expression *e;
513
514     switch(term->kind){
515         case (term_VAR):
516             term->val.variable = weed_variable(term->val.variable);
517             break;
518
519         case (term_LIST):
520             term->val.list.list = weed_alist(term->val.list.list);
521             break;
522
523         case (term_PAR):
524             term->val.expression = weed_expression(term->val.expression);
525             break;
526
527         case (term_NOT):
528             term->val.term_not = weed_term(term->val.term_not);
529             if (term->val.term_not->kind == term_TRUE){
530                 term->kind = term_FALSE;
531                 break;
532             }
533
534             if (term->val.term_not->kind == term_FALSE){
535                 term->kind = term_TRUE;
536                 break;
537             }
538
539             if (term->val.term_not->kind == term_NOT){
540                 term = term->val.term_not->val.term_not;
541                 break;
542             }
543
544             if (term->val.term_not->kind == term_PAR){
545                 e = term->val.term_not->val.expression;
546                 switch(e->kind){
547
548                     case (exp_EQ):
549                         e->kind = exp_NEQ;
550                         term = term->val.term_not;
551                         break;
552
553                     case (exp_NEQ):
554                         e->kind = exp_EQ;
555                         term = term->val.term_not;
```

```
556                         break;
557
558                     case (exp_LT):
559                         e->kind = exp_GEQ;
560                         term = term->val.term_not;
561                         break;
562
563                     case (exp_GT):
564                         e->kind = exp_LEQ;
565                         term = term->val.term_not;
566                         break;
567
568                     case (exp_LEQ):
569                         e->kind = exp_GT;
570                         term = term->val.term_not;
571                         break;
572
573                     case (exp_GEQ):
574                         e->kind = exp_LT;
575                         term = term->val.term_not;
576                         break;
577
578                     default:
579                         break;
580
581                 }
582             }
583             break;
584
585         case (term_ABS):
586             term->val.expression = weed_expression(term->val.expression);
587             if ((term->val.expression->kind == exp_TERM) && (term->val.expression->val.term->kind
                      == term_NUM)){
588                 term->kind = term_NUM;
589                 term->val.num = abs(term->val.expression->val.term->val.num); // Should probably
                        use a temp value instead of such a long value
590             }
591             break;
592
593         default:
594             break;
595
596     }
597
598     return term;
599
600 }
601
602 act_list *weed_alist(act_list *alist){
603     //printf("Weeding alist\n");
604     if (alist->kind == al_EMPTY){
605         return alist;
606     }
607     alist->list = weed_elist(alist->list);
608     return alist;
609
610 }
611
612 exp_list *weed_elist(exp_list *elist){
613     //printf("Weeding elist\n");
614
615     elist->expression = weed_expression(elist->expression);
616
```

```
617     if (elist->kind == el_LIST){
618         elist->list = weed_elist(elist->list);
619     }
620 
621     return elist;
622 
623 }
```

**setup.h**

```
1  #ifndef __setup_h
2  #define __setup_h
3  #include "tree.h"
4  #include "symbol.h"
5  
6  void setup_body(body *body, symbol_table *table);
7  
8  void setup_function(function *function, symbol_table *table);
9  
10 void setup_head(head *head, symbol_table *table, symbol_table *outer_scope);
11 
12 void setup_type(type *type, symbol_table *table);
13 
14 int setup_pdl(par_decl_list *pdl, symbol_table *table);
15 
16 int setup_vdl(var_decl_list *vdl, symbol_table *table);
17 
18 void setup_vtype(var_type *vtype, symbol_table*table);
19 
20 void setup_dlist(decl_list *dlist, symbol_table*table);
21 
22 void setup_decl(declaration *decl, symbol_table *table);
23 
24 void setup_slist(statement_list *slist, symbol_table *table);
25 
26 void setup_stmt(statement *stmt, symbol_table *table);
27 
28 void setup_var(variable *var, symbol_table *table);
29 
30 void setup_exp(expression *exp, symbol_table *table);
31 
32 void setup_term(term *term, symbol_table *table);
33 
34 void setup_alist(act_list *alist, symbol_table *table);
35 
36 void setup_elist(exp_list *elist, symbol_table *table);
37 
38 
39 #endif
```

**setup.c**

```
1  /**
2   * @brief
3   *
4   * @file setup.c
5   * @author Morten Jæger
6   * @date 2018-03-09
7   */
8  
```

```c
#include <stdio.h>
#include <stdlib.h>
#include "setup.h"
#include "symbol.h"
#include "memory.h"
#include "error.h"

/**
 *
 * TODO All of this can possibly be put in the "tree.c", when we create a node.
 * This means that we do not have to run through the AST again to set up the symbol table
 *
 */


void setup_body(body *body, symbol_table *table){

    body->table = table;

    //printf("Setting up declaration list\n");
    setup_dlist(body->d_list, table);

    //printf("Setting up statement list\n");
    setup_slist(body->s_list, table);

}



void setup_function(function *function, symbol_table *table){

    //printf("Setting up function\n");
    symbol_table*nextTable;
    nextTable = scope_symbol_table(table);
    function->table = nextTable;
    function->tail->table = nextTable;
    setup_head(function->head, nextTable, table);
    setup_body(function->body, nextTable);

    SYMBOL *s;
    s = get_symbol(table, function->head->id);
    if (s == NULL || s->stype->type != symbol_FUNCTION){
        print_error("Function does not exist", 0, function->lineno);
    }
    s->stype->val.func_type.func = function;


}

void setup_head(head *head, symbol_table *table, symbol_table *outer_scope){

    //printf("Setting up head\n");
    head->table = table;
    symbol_type *st;
    st = NEW(symbol_type);
    st->type = symbol_FUNCTION;
    put_symbol(outer_scope, head->id, 0, st);
    head->args = setup_pdl(head->list, table);
    head->stype = st;

    //printf("Number of args for function %s: %d\n", head->id, head->args);

    setup_type(head->type, outer_scope);
```

```
72      st->val.func_type.pdl = head->list;
73
74  }
75
76  void setup_type(type *type, symbol_table*table){
77      type->table = table;
78      symbol_type *st;
79
80      st = NEW(symbol_type);
81      //printf("Setting up type\n");
82
83      switch(type->kind){
84
85          case (type_ID):
86              st->type = symbol_ID;
87              type->stype = st;
88              break;
89
90          case (type_INT):
91              st->type = symbol_INT;
92              type->stype = st;
93              break;
94
95          case (type_BOOl):
96              st->type = symbol_BOOL;
97              type->stype = st;
98              break;
99
100         case (type_ARRAY):
101             st->type = symbol_ARRAY;
102             type->stype = st;
103             setup_type(type->val.type, table);
104             break;
105
106         case (type_RECORD):
107             st->type = symbol_RECORD;
108             type->stype = st;
109             setup_vdl(type->val.list, scope_symbol_table(table));
110             break;
111     }
112
113
114 }
115
116 int setup_pdl(par_decl_list *pdl, symbol_table*table){
117     //printf("Setting up pdl\n");
118     pdl->table = table;
119     int args;
120     args = 0;
121     if (pdl->kind != pdl_EMPTY){
122         args = args + setup_vdl(pdl->list, table);
123     }
124     return args;
125 }
126
127 int setup_vdl(var_decl_list *vdl, symbol_table *table){
128
129     //printf("Setting up vdl\n");
130     vdl->table = table;
131     int args;
132     args = 1;
133     setup_vtype(vdl->vartype, table);
134     if (vdl->kind == vdl_LIST){
```

```
135         args = args + setup_vdl(vdl->list, table);
136     }
137     return args;
138 }
139 
140 void setup_vtype(var_type *vtype, symbol_table*table){
141 
142     //printf("Setting up var_type\n");
143     vtype->table = table;
144     symbol_type *st;
145     st = NEW(symbol_type);
146     st->type = symbol_UNKNOWN; // Sikkert ikke rigtigt
147 
148     SYMBOL *s;
149     s = put_symbol(table, vtype->id, 0, st);
150     vtype->symbol = s;
151 
152     setup_type(vtype->type, table);
153 
154 
155 }
156 
157 void setup_dlist(decl_list *dlist, symbol_table *table){
158 
159     //printf("Setting up dlist\n");
160     if (dlist->kind != dl_EMPTY){
161         setup_decl(dlist->decl, table);
162         setup_dlist(dlist->list, table);
163     }
164 
165 }
166 
167 void setup_decl(declaration *decl, symbol_table *table){
168 
169     //printf("Setting up declaration\n");
170     decl->table = table;
171 
172     symbol_type *st;
173     switch (decl->kind){
174 
175         case (decl_TYPE):
176             st = NEW(symbol_type);
177             st->type = symbol_ID;
178             put_symbol(table, decl->val.type.id, 0, st);
179             setup_type(decl->val.type.type, table);
180             st->val.id_type = decl->val.type.type;
181             break;
182 
183         case (decl_FUNC):
184             setup_function(decl->val.function, table);
185             break;
186 
187         case (decl_VAR):
188             setup_vdl(decl->val.list, table);
189             break;
190 
191     }
192 
193 }
194 
195 void setup_slist(statement_list *slist, symbol_table *table){
196 
197     //printf("Setting up slist\n");
```

```
198     slist->table = table;
199     setup_stmt(slist->statement, table);
200     if (slist->list != NULL){
201         setup_slist(slist->list, table);
202     }
203 
204 }
205 
206 void setup_stmt(statement *stmt, symbol_table*table){
207 
208     //printf("Setting up statement\n");
209     stmt->table = table;
210     switch(stmt->kind){
211 
212         case (statement_RETURN):
213 
214             //printf("\tStatement return\n");
215             setup_exp(stmt->val.ret, table);
216             break;
217 
218         case (statement_WRITE):
219             //printf("\tStatement write\n");
220             setup_exp(stmt->val.wrt, table);
221             break;
222 
223         case (statement_ALLOCATE):
224             //printf("\tStatement allocate\n");
225             setup_var(stmt->val.allocate.variable, table);
226             break;
227 
228         case (statement_ALLOCATE_LENGTH):
229             //printf("\tStatement allocate length\n");
230             setup_var(stmt->val.allocate.variable, table);
231             setup_exp(stmt->val.allocate.length, table);
232             break;
233 
234         case (statement_ASSIGNMENT):
235             //printf("\tStatement assignment\n");
236             setup_var(stmt->val.assignment.variable, table);
237             setup_exp(stmt->val.assignment.expression, table);
238             break;
239 
240         case (statement_IF):
241             //printf("\tStatement if\n");
242             setup_exp(stmt->val.ifthen.expression, table);
243             setup_stmt(stmt->val.ifthen.statement1, table);
244             break;
245 
246         case (statement_IF_ELSE):
247             //printf("\tStatement if else\n");
248             setup_exp(stmt->val.ifthen.expression, table);
249             setup_stmt(stmt->val.ifthen.statement1, table);
250             setup_stmt(stmt->val.ifthen.statement2, table);
251             break;
252 
253         case (statement_WHILE):
254             //printf("\tStatement while\n");
255             setup_exp(stmt->val.loop.expression, table);
256             setup_stmt(stmt->val.loop.statement, table);
257             break;
258 
259         case (statement_LIST):
260             //printf("\tStatement list\n");
```

```
261             setup_slist(stmt->val.list, table);
262             break;
263     }
264 
265 }
266 
267 void setup_var(variable *var, symbol_table*table){
268 
269     //printf("Setting up variable\n");
270     var->table = table;
271 
272     switch (var->kind){
273 
274         case (var_ID):
275             //printf("ID: %s\n", var->id);
276             break;
277 
278         case (var_EXP):
279             setup_var(var->val.exp.var, table);
280             setup_exp(var->val.exp.exp, table);
281             break;
282 
283         case (var_RECORD):
284             setup_var(var->val.record.var, table);
285             break;
286 
287     }
288 
289 }
290 
291 void setup_exp(expression *exp, symbol_table *table){
292 
293     //printf("Setting up expression\n");
294     exp->table = table;
295     //printf("Expression kind: %d\n", exp->kind);
296     if (exp->kind == exp_TERM){
297         setup_term(exp->val.term, table);
298     } else {
299         setup_exp(exp->val.ops.left, table);
300         setup_exp(exp->val.ops.right, table);
301     }
302 
303 }
304 
305 void setup_term(term *term, symbol_table *table){
306 
307     //printf("Setting up term\n");
308     term->table = table;
309     switch(term->kind){
310 
311         case (term_VAR):
312             setup_var(term->val.variable, table);
313             break;
314 
315         case (term_LIST):
316             setup_alist(term->val.list.list, table);
317             break;
318 
319         case (term_PAR):
320             setup_exp(term->val.expression, table);
321             break;
322 
323         case (term_NOT):
```

```
324             setup_term(term->val.term_not, table);
325             break;
326 
327         case (term_ABS):
328             setup_exp(term->val.expression, table);
329             break;
330 
331         default:
332             break;
333 
334     }
335 
336 }
337 
338 void setup_alist(act_list *alist, symbol_table *table){
339 
340     //printf("Setting up alist\n");
341     alist->table = table;
342     if (alist->kind == al_LIST){
343         setup_elist(alist->list, table);
344     }
345 
346 }
347 
348 void setup_elist(exp_list *elist, symbol_table*table){
349     //printf("Setting up elist\n");
350     elist->table = table;
351 
352     switch(elist->kind){
353 
354         case (el_EXP):
355             setup_exp(elist->expression, table);
356             break;
357 
358         case (el_LIST):
359             setup_exp(elist->expression, table);
360             setup_elist(elist->list, table);
361             break;
362     }
363 }
```

**pickup.h**

```
1  #ifndef __pickup_h
2  #define __pickup_h
3  #include "tree.h"
4  
5  void pickup_body(body *body);
6  
7  void pickup_function(function *function);
8  
9  void pickup_head(head *head);
10 
11 void pickup_pdl(par_decl_list *pdl);
12 
13 void pickup_vdl(var_decl_list *vdl);
14 
15 void pickup_vtype(var_type *vtype);
16 
17 void pickup_dlist(decl_list *dlist);
18 
```

```
19 void pickup_declaration(declaration *decl);
20
21 void pickup_type(type *type);
22
23 type *resolve_recursive_type(type *type);
24
25 #endif
```

**pickup.c**

```
1  #include <stdio.h>
2  #include "debug.h"
3  #include "pickup.h"
4  #include "tree.h"
5  #include "symbol.h"
6  #include "error.h"
7
8
9
10 void pickup_body(body *body){
11 #if debugflag > 2
12     printf("Picking up body\n");
13 #endif
14     pickup_dlist(body->d_list);
15
16 }
17
18 void pickup_function(function *function){
19 #if debugflag > 2
20     printf("Picking up function\n");
21 #endif
22     pickup_head(function->head);
23     pickup_body(function->body);
24     function->stype = function->head->stype;
25 }
26
27 void pickup_head(head *head){
28 #if debugflag > 2
29     printf("Picking up head\n");
30 #endif
31     pickup_pdl(head->list);
32     pickup_type(head->type);
33 #if debugflag > 3
34     printf("Picked up Type in Head, Head type: %d, Symbol type: %d\n", head->type->kind, head->
           stype->type);
35 #endif
36     head->stype->val.func_type.ret_type = head->type;
37 #if debugflag > 2
38     printf("Assigned ret type\n");
39 #endif
40 }
41
42 void pickup_pdl(par_decl_list *pdl){
43 #if debugflag > 2
44     printf("Picking up pdl\n");
45 #endif
46     if (pdl->kind == pdl_LIST){
47         pickup_vdl(pdl->list);
48     }
49
50 }
```

```
51
52  void pickup_vdl(var_decl_list *vdl){
53  #if debugflag > 2
54      printf("Picking up vdl\n");
55  #endif
56      pickup_vtype(vdl->vartype);
57      if (vdl->kind == vdl_LIST){
58          pickup_vdl(vdl->list);
59      }
60
61  }
62
63  void pickup_vtype(var_type *vtype){
64  #if debugflag > 3
65      printf("Picking up vtype, id: %s\n", vtype->id);
66  #endif
67      pickup_type(vtype->type);
68  #if debugflag > 2
69      printf("Picked up Type in VType\n");
70  #endif
71      vtype->symbol->stype = vtype->type->stype;
72
73  }
74
75  void pickup_dlist(decl_list *dlist){
76  #if debugflag > 2
77      printf("Picking up dlist\n");
78  #endif
79      if (dlist->kind == dl_LIST){
80          pickup_declaration(dlist->decl);
81          pickup_dlist(dlist->list);
82      }
83  }
84
85  void pickup_declaration(declaration *decl){
86  #if debugflag > 2
87      printf("Picking up declaration\n");
88  #endif
89
90      switch (decl->kind){
91
92          case (decl_FUNC):
93              pickup_function(decl->val.function);
94              break;
95
96          case (decl_TYPE):
97              pickup_type(decl->val.type.type);
98              break;
99
100         case (decl_VAR):
101             pickup_vdl(decl->val.list);
102             break;
103     }
104
105 }
106
107 void pickup_type(type *type){
108 #if debugflag > 2
109     printf("Picking up type, kind: %d\n", type->kind);
110 #endif
111
112     SYMBOL *s;
113     switch (type->kind){
```

```
114
115         case (type_ARRAY):
116             pickup_type(type->val.type);
117             type->stype->val.array_type = type->val.type;
118             break;
119
120         case (type_BOOl):
121             break;
122
123         case (type_INT):
124             break;
125
126         case (type_RECORD):
127             pickup_vdl(type->val.list);
128             type->stype->val.record_type = type->val.list;
129             break;
130
131         case (type_ID):
132 #if debugflag > 3
133             printf("ID we are looking for: %s, in table: %p\n", type->val.id, type->table);
134 #endif
135             s = get_symbol(type->table, type->val.id);
136             if (s == NULL || s->stype->type != symbol_ID){
137                 if (s == NULL){
138                     printf("Symbol is NULL\n");
139                 }
140                 if (s->stype->type != symbol_ID){
141                     printf("Symbol is not ID, it is of type: %d\n", s->stype->type);
142                 }
143                 print_error("Identifier error", 0, type->lineno);
144             }
145             struct type *temp;
146             temp = resolve_recursive_type(s->stype->val.id_type);
147 #if debugflag > 3
148             printf("After recursive check\n");
149 #endif
150             type->stype = temp->stype;
151 #if debugflag > 3
152             printf("After assignment\n");
153 #endif
154     }
155 #if debugflag > 3
156     printf("After switch\n");
157 #endif
158
159 }
160
161 type *resolve_recursive_type(type *type){
162 #if debugflag > 2
163     printf("Resolving recursive conflict\n");
164 #endif
165     struct type *temp;
166     temp = type;
167     if (type->recursive_type == 1){
168         print_error("Recursive type definition", 0, type->lineno);
169     }
170     type->recursive_type = 1;
171 #if debugflag > 3
172     printf("Type kind: %d\n", type->kind);
173 #endif
174     if (type->kind == type_ID){
175         printf("Checking symbol table for symbol\n");
176         SYMBOL *s;
```

```
177         s = get_symbol(type->table, type->val.id);
178         if (s == NULL || s->stype->type != symbol_ID){
179             print_error("Undefined identifier", 0, type->lineno);
180         }
181         temp = resolve_recursive_type(s->stype->val.id_type);
182     }
183 #if debugflag > 3
184     printf("Checked recursively\n");
185 #endif
186     type->recursive_type = 0;
187     return temp;
188 }
```

**check.h**

```
1  #ifndef __check_h
2  #define __check_h
3  #include "tree.h"
4  #include "symbol.h"
5
6
7  void check_body(body *body);
8
9  void check_function(function *function);
10
11 void check_dlist(decl_list *dlist);
12
13 void check_decl(declaration *decl);
14
15 void check_slist(statement_list *slist);
16
17 void check_stmt(statement *stmt);
18
19 void check_var(variable *var);
20
21 void check_exp(expression *exp);
22
23 void check_term(term *term);
24
25 void check_alist(act_list *alist);
26
27 void check_elist(exp_list *elist);
28
29 int check_function_args(par_decl_list *pdl, act_list *alist);
30
31 int compare_stype(symbol_type *stype1, symbol_type *stype2);
32
33 int compare_record(symbol_type *stype1, symbol_type *stype2);
34
35
36
37 #endif
```

**check.c**

```
1  #include <stdio.h>
2  #include "debug.h"
3  #include "tree.h"
4  #include "check.h"
5  #include "memory.h"
```

```
6  #include "error.h"
7  #include "symbol.h"
8  #include "pickup.h"
9
10
11 void check_body(body *body){
12
13     check_dlist(body->d_list);
14     check_slist(body->s_list);
15
16 }
17
18 void check_function(function *function){
19
20     check_body(function->body);
21
22 }
23
24 void check_dlist(decl_list *dlist){
25
26     if (dlist->kind == dl_LIST){
27         check_dlist(dlist->list);
28         check_decl(dlist->decl);
29     }
30
31 }
32
33 void check_decl(declaration *decl){
34
35     if (decl->kind == decl_FUNC){
36         check_function(decl->val.function);
37     }
38
39 }
40
41 void check_slist(statement_list *slist){
42
43     check_stmt(slist->statement);
44     if (slist->kind == sl_LIST){
45         check_slist(slist->list);
46     }
47
48 }
49
50 void check_stmt(statement *stmt){
51 #if debugflag > 2
52     printf("Checking statement, kind: %d\n", stmt->kind);
53 #endif
54
55
56     switch(stmt->kind){
57
58         case (statement_RETURN):
59             check_exp(stmt->val.ret);
60             if (stmt->function->stype->val.func_type.ret_type->stype->type != stmt->val.ret->
                    stype->type){
61                 print_error("Wrong return type", 0, stmt->lineno);
62             }
63             break;
64
65         case (statement_WRITE):
66             check_exp(stmt->val.wrt);
```

```
67            if ((stmt->val.wrt->stype->type != symbol_INT) && (stmt->val.wrt->stype->type !=
                  symbol_BOOL) && (stmt->val.wrt->stype->type != symbol_ID)){
68               print_error("Wrong write type", 0, stmt->lineno);
69            }
70            break;
71
72        case (statement_ALLOCATE):
73            check_var(stmt->val.allocate.variable);
74 #if debugflag > 3
75            printf("Allocating type: %d\n", stmt->val.allocate.variable->stype->type);
76 #endif
77            if ( (stmt->val.allocate.variable->stype->type != symbol_ARRAY) && (stmt->val.
                  allocate.variable->stype->type != symbol_RECORD)){
78               print_error("Wrong allocate type", 0, stmt->lineno);
79            }
80            break;
81
82        case (statement_ALLOCATE_LENGTH):
83            check_var(stmt->val.allocate.variable);
84            check_exp(stmt->val.allocate.length);
85            if ( (stmt->val.allocate.variable->stype->type != symbol_ARRAY) && (stmt->val.
                  allocate.variable->stype->type != symbol_RECORD)){
86               print_error("Wrong allocate type", 0, stmt->lineno);
87            }
88            if (stmt->val.allocate.length->stype->type != symbol_INT){
89               print_error("Allocate length must be integer", 0, stmt->lineno);
90            }
91            break;
92
93        case (statement_ASSIGNMENT):
94            check_var(stmt->val.assignment.variable);
95            check_exp(stmt->val.assignment.expression);
96            if (stmt->val.assignment.expression->stype->type == symbol_NULL){
97               if ( (stmt->val.assignment.variable->stype->type != symbol_ARRAY) && (stmt->val.
                     assignment.variable->stype->type != symbol_RECORD)){
98                  print_error("Can only assign array and record to NULL", 0 , stmt->lineno);
99               }
100              break;
101           }
102           if (!compare_stype(stmt->val.assignment.variable->stype, stmt->val.assignment.
                 expression->stype)){
103              print_error("Incompatible type assignment", 0, stmt->lineno);
104           }
105           break;
106
107       case (statement_IF):
108           check_exp(stmt->val.ifthen.expression);
109           check_stmt(stmt->val.ifthen.statement1);
110           if (stmt->val.ifthen.expression->stype->type != symbol_BOOL){
111              print_error("If condition is not a boolean", 0, stmt->lineno);
112           }
113           break;
114
115       case (statement_IF_ELSE):
116           check_exp(stmt->val.ifthen.expression);
117           check_stmt(stmt->val.ifthen.statement1);
118           check_stmt(stmt->val.ifthen.statement2);
119           if (stmt->val.ifthen.expression->stype->type != symbol_BOOL){
120              print_error("If condition is not a boolean", 0, stmt->lineno);
121           }
122           break;
123
124       case (statement_WHILE):
```

```
125             check_exp(stmt->val.loop.expression);
126             check_stmt(stmt->val.loop.statement);
127             break;
128 
129         case (statement_LIST):
130             check_slist(stmt->val.list);
131             break;
132 
133     }
134 
135 }
136 
137 void check_var(variable *var){
138 #if debugflag > 2
139     printf("Checking variable, kind: %d\n", var->kind);
140 #endif
141 
142     SYMBOL *s;
143     switch (var->kind){
144 
145         case (var_ID):
146             s = get_symbol(var->table, var ->id);
147             if (s == NULL){
148                 print_error("Symbol not defined", 0, var->lineno);
149             }
150             var->stype = s->stype;
151             break;
152 
153         case (var_EXP):
154             check_var(var->val.exp.var);
155             check_exp(var->val.exp.exp);
156             if (var->val.exp.exp->stype->type != symbol_INT){
157                 print_error("Expression in [] not an integer", 0, var->lineno);
158             }
159             if (var->val.exp.var->stype->type != symbol_ARRAY){
160                 print_error("Variable is not an array", 0, var->lineno);
161             }
162             var->stype = var->val.exp.var->stype->val.array_type->stype;
163             break;
164 
165         case (var_RECORD):
166             check_var(var->val.record.var);
167             if (var->val.record.var->stype->type != symbol_RECORD){
168                 print_error("Variable is not an record", 0, var->lineno);
169             }
170             s = get_symbol(var->val.record.var->stype->val.record_type->table, var->val.record.id
                    );
171             if (s == NULL){
172                 print_error("Record entry does not exist", 0, var->lineno);
173             }
174             var->stype = s->stype;
175             break;
176     }
177 
178 }
179 
180 void check_exp(expression *exp){
181 #if debugflag > 2
182     printf("Checking expression, kind %d\n", exp->kind);
183 #endif
184     symbol_type *st;
185 
186     switch(exp->kind){
```

```
187
188         case (exp_PLUS):      // Subject to change, could be made to work with strings, f.x. "Hi
                "+2 could be: "Hi2"
189         case (exp_MIN):
190         case (exp_MULT):      // Subject to change, could be made to work with strings, f.x. "Hi
                "*2 could be: "HiHi"
191         case (exp_DIV):
192 #if debugflag > 3
193             printf("Checking left expression, Arithmetic\n");
194 #endif
195             check_exp(exp->val.ops.left);
196 #if debugflag > 3
197             printf("Checking right expression, Arithmetic\n");
198 #endif
199             check_exp(exp->val.ops.right);
200             if (exp->val.ops.left->stype->type == symbol_INT && exp->val.ops.right->stype->type
                    == symbol_INT){
201
202                 st = NEW(symbol_type);
203                 st->type = symbol_INT;
204                 exp->stype= st;
205
206             } else {
207                 print_error("Operators in arithmetic expression are not integers", 0, exp->lineno
                        );
208             }
209             break;
210
211         case (exp_EQ):
212         case (exp_NEQ):
213 #if debugflag > 2
214             printf("Checking left expression, EQ/NEQ\n");
215 #endif
216             check_exp(exp->val.ops.left);
217 #if debugflag > 2
218             printf("Checking right expression, EQ/NEQ\n");
219 #endif
220             check_exp(exp->val.ops.right);
221
222             // Should check stype->type->val.func_type.ret_type if comparing a function
223             if ((exp->val.ops.left->stype->type == symbol_RECORD) && (exp->val.ops.right->stype->
                    type == symbol_NULL)){
224 #if debugflag > 3
225                 printf("Left type is record, and right type is NULL\n");
226 #endif
227                 st = NEW(symbol_type);
228                 st->type = symbol_BOOL;
229                 exp->stype = st;
230                 break;
231             }
232
233             if ((exp->val.ops.left->stype->type == symbol_NULL) && (exp->val.ops.right->stype->
                    type == symbol_RECORD)){
234 #if debugflag > 3
235                 printf("Left type is NULL, and right type is record\n");
236 #endif
237                 st = NEW(symbol_type);
238                 st->type = symbol_BOOL;
239                 exp->stype = st;
240                 break;
241             }
242 #if debugflag > 3
```

```
243             printf("Left sides type: %d, Right sides type: %d\n", exp->val.ops.left->stype->type,
                    exp->val.ops.right->stype->type);
244 #endif
245             if (exp->val.ops.left->stype->type == exp->val.ops.right->stype->type){
246 #if debugflag > 3
247                 printf("Checked if type is the same\n");
248 #endif
249                 st = NEW(symbol_type);
250                 st->type = symbol_BOOL;
251                 exp->stype = st;
252 
253             } else {
254                 print_error("Operators in EQ or NEQ not the same", 0, exp->lineno);
255             }
256             break;
257 
258         case (exp_GEQ):
259         case (exp_LEQ):
260         case (exp_LT):
261         case (exp_GT):
262 #if debugflag > 2
263             printf("Checking left expression, GEQ/LEQ/LT/GT\n");
264 #endif
265             check_exp(exp->val.ops.left);
266 #if debugflag > 2
267             printf("Checking right expression, GEQ/LEQ/LT/GT\n");
268 #endif
269             check_exp(exp->val.ops.right);
270             if (exp->val.ops.left->stype->type == symbol_INT && exp->val.ops.right->stype->type
                    == symbol_INT){
271 
272                 st = NEW(symbol_type);
273                 st->type = symbol_BOOL;
274                 exp->stype = st;
275 
276             } else {
277                 print_error("Operators used in GEQ, LEQ, LT or GT not integers", 0, exp->lineno);
278             }
279             break;
280 
281         case (exp_AND):
282         case (exp_OR):
283 #if debugflag > 2
284             printf("Checking left expression, AND/OR\n");
285 #endif
286             check_exp(exp->val.ops.left);
287 #if debugflag > 2
288             printf("Checking right expression, AND/OR\n");
289 #endif
290             check_exp(exp->val.ops.right);
291 
292             if (exp->val.ops.left->stype->type == symbol_BOOL && exp->val.ops.right->stype->type
                    == symbol_BOOL){
293 
294                 st = NEW(symbol_type);
295                 st->type = symbol_BOOL;
296                 exp->stype = st;
297 
298             } else {
299                 print_error("Operators used in AND or OR not boolean", 0, exp->lineno);
300             }
301             break;
302 
```

```
303         case (exp_TERM):
304             check_term(exp->val.term);
305             exp->stype = exp->val.term->stype;
306             break;
307 
308     }
309 
310 }
311 
312 void check_term(term *term){
313 #if debugflag > 2
314     printf("Checking term, kind: %d\n", term->kind);
315 #endif
316 
317     SYMBOL *s;
318     symbol_type *st;
319 
320     switch(term->kind){
321 
322         case (term_VAR):
323             check_var(term->val.variable);
324             term->stype = term->val.variable->stype;
325 #if debugflag > 3
326             printf("Term type: %d\n", term->stype->type);
327 #endif
328             break;
329 
330         case (term_LIST):
331             check_alist(term->val.list.list);
332             s = get_symbol(term->table, term->val.list.id);
333             if (s == NULL || s->stype->type != symbol_FUNCTION){
334                 print_error("Reference to function that does not exists", 0, term->lineno);
335             }
336             check_function_args(s->stype->val.func_type.pdl, term->val.list.list);
337             term->stype = s->stype->val.func_type.ret_type->stype;
338             break;
339 
340         case (term_PAR):
341             check_exp(term->val.expression);
342             term->stype->type = term->val.expression->stype->type;
343             break;
344 
345         case (term_NOT):
346             check_term(term->val.term_not);
347             if (term->val.term_not->stype->type != symbol_BOOL){
348                 print_error("Cannot negate non-boolean", 0, term->lineno);
349 
350             }
351             term->stype->type = term->val.term_not->stype->type;
352             break;
353 
354         case (term_ABS):
355             check_exp(term->val.expression);
356 #if debugflag > 3
357             printf("Type of expression: %d\n", term->val.expression->stype->type);
358 #endif
359             if ((term->val.expression->stype->type != symbol_INT) && (term->val.expression->stype
                 ->type != symbol_ARRAY)){
360                 print_error("Absolute value must be used on integer or array", 0 , term->lineno);
361             }
362             st = NEW(symbol_type);
363             st->type = symbol_INT;
364             term->stype = st;
```

```
365             break;
366
367         case (term_NUM):
368             st = NEW(symbol_type);
369             st->type = symbol_INT;
370             term->stype = st;
371             break;
372
373         case (term_TRUE):
374         case (term_FALSE):
375             st = NEW(symbol_type);
376             st->type = symbol_BOOL;
377             term->stype = st;
378             break;
379
380         case (term_NULL):
381             st = NEW(symbol_type);
382             st->type = symbol_NULL;
383             term->stype = st;
384             break;
385
386     }
387
388 }
389
390 void check_alist(act_list *alist){
391
392     if (alist->kind == al_LIST){
393         check_elist(alist->list);
394     }
395
396 }
397
398
399 void check_elist(exp_list *elist){
400     check_exp(elist->expression);
401     if (elist->kind == el_LIST){
402         check_elist(elist->list);
403     }
404
405
406 }
407
408 int check_function_args(par_decl_list *pdl, act_list *alist){
409
410     struct exp_list *elist;
411     struct var_decl_list *vdl;
412     struct type *temp;
413     struct symbol_type *st1;
414     struct symbol_type *st2;
415
416     if (pdl->kind == pdl_EMPTY){
417 #if debugflag > 3
418         printf("PDL is empty\n");
419 #endif
420         if (alist->kind == al_EMPTY){
421             return 1;
422         } else {
423             print_error("Too many function arguments", 0, alist->lineno);
424         }
425     } else {
426         if (alist->kind == al_EMPTY){
427             print_error("Too few function arguments", 0, alist->lineno);
```

```
428         }
429     }
430 
431     elist = alist->list;
432     vdl = pdl->list;
433 
434     while ((vdl != NULL) && (elist != NULL)){
435         //In case of recursive definition
436         temp = resolve_recursive_type(vdl->vartype->type);
437 
438         st1 = temp->stype;
439         st2 = elist->expression->stype;
440 #if debugflag > 2
441         printf("ST1s type: %d, ST2s type: %d\n", st1->type, st2->type);
442 #endif
443 
444         if (!compare_stype(st1, st2)){
445             print_error("Function argument type mismatch", 0, alist->lineno);
446         }
447         vdl = vdl->list;
448         elist = elist->list;
449     }
450 
451     if ((vdl == NULL) && (elist == NULL)){
452         return 1;
453     }
454     if (vdl == NULL){
455         print_error("Too many function arguments", 0, alist->lineno);
456     }
457     if (elist == NULL){
458         print_error("Too few function arguments", 0, alist->lineno);
459     }
460 
461     return 0;
462 
463 }
464 
465 int compare_stype(symbol_type *stype1, symbol_type *stype2){
466 
467     if (stype1 == NULL || stype2 == NULL){
468 #if debugflag > 3
469         printf("Both NULL, return 0\n");
470 #endif
471         return 0;
472     }
473 
474     if (stype1->type == stype2->type){
475 #if debugflag > 3
476         printf("Both equal, return 1\n");
477 #endif
478         return 1;
479     }
480 
481     if (stype1->type == symbol_ARRAY && stype2->type == symbol_ARRAY){
482         return compare_stype(stype1->val.array_type->stype, stype2->val.array_type->stype);
483     }
484 
485     if (stype1->type == symbol_RECORD && stype2->type == symbol_RECORD){
486         return compare_record(stype1, stype2);
487     }
488 #if debugflag > 3
489     printf("Not equal, return 0\n");
490 #endif
```

```
491
492     return 0;
493
494 }
495
496 int compare_record(symbol_type *stype1, symbol_type *stype2){
497
498     var_decl_list *vdl1;
499     var_decl_list *vdl2;
500
501     symbol_table *table1;
502     symbol_table *table2;
503
504     SYMBOL *s1;
505     SYMBOL *s2;
506
507     vdl1 = stype1->val.record_type;
508     vdl2 = stype2->val.record_type;
509
510     table1 = stype1->val.record_type->table;
511     table2 = stype2->val.record_type->table;
512
513     while ((vdl1 != NULL) && (vdl2 != NULL)){
514
515         s1 = get_symbol(table1, vdl1->vartype->id);
516         s2 = get_symbol(table2, vdl2->vartype->id);
517
518         if (s1 == NULL || s2 == NULL){
519             return 0;
520         }
521
522         if (!compare_stype(s1->stype, s2->stype)){
523             return 0;
524         }
525         vdl1 = vdl1->list;
526         vdl2 = vdl2->list;
527     }
528
529     if ((vdl1 == NULL) && (vdl2 == NULL)){
530         return 1;
531     }
532
533     return 0;
534
535 }
```

**debug.h**

```
1  /**
2   * @brief
3   *
4   * @file debug.h
5   * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
6   * @date 2018-03-16
7   */
8  #ifndef COMPILER_DEBUG_H
9
10
11 //0 none,
12 // 1 basic debug levels, eg. are the modules called correctly.
13 // 2 advanced debug levels, eg. is the program transversing the functions as expected.
```

```
14 // 3 all the debug things. print all debug information within functions.
15
16 #define debugflag 0
17
18 #define COMPILER_DEBUG_H
19
20 #endif //COMPILER_DEBUG_H
```

**kind.h**

```
1  #ifndef __compiler_kind_h
2  #define __compiler_kind_h
3
4  typedef enum { exp_PLUS,
5                 exp_MIN,
6                 exp_MULT,
7                 exp_DIV,
8                 exp_EQ,
9                 exp_NEQ,
10                exp_GT,
11                exp_LT,
12                exp_GEQ,
13                exp_LEQ,
14                exp_AND,
15                exp_OR,
16                exp_TERM } EXP_kind;
17
18 typedef enum { term_VAR,
19                term_LIST,
20                term_PAR,
21                term_NOT,
22                term_ABS,
23                term_NUM,
24                term_TRUE,
25                term_FALSE,
26                term_NULL } TERM_kind;
27
28 typedef enum { type_ID,
29                type_INT,
30                type_BOOl,
31                type_ARRAY,
32                type_RECORD } TYPE_kind;
33
34 typedef enum { pdl_LIST,
35                pdl_EMPTY } PDL_kind;
36
37 typedef enum { vdl_LIST,
38                vdl_TYPE } VDL_kind;
39
40 typedef enum { dl_LIST,
41                dl_EMPTY } DL_kind;
42
43 typedef enum { decl_TYPE,
44                decl_FUNC,
45                decl_VAR } DECL_kind;
46
47 typedef enum { sl_STATEMENT,
48                sl_LIST } SL_kind;
49
50 typedef enum { statement_RETURN,
51                statement_WRITE,
```

```
52              statement_ALLOCATE,
53              statement_ALLOCATE_LENGTH,
54              statement_ASSIGNMENT,
55              statement_IF,
56              statement_IF_ELSE,
57              statement_WHILE,
58              statement_LIST } STATEMENT_kind;
59
60  typedef enum { var_ID,
61              var_EXP,
62              var_RECORD } Var_kind;
63
64  typedef enum { al_LIST,
65              al_EMPTY } AL_kind;
66
67  typedef enum { el_EXP,
68              el_LIST } EL_kind;
69
70  typedef enum { jmp, //unconditional jump
71              je, //jump equal
72              jne, //Jump not equal
73              jg, //jump greater
74              jge, //jump greater or equal
75              jl, //Jump less
76              jle, //Jump less or equal
77
78              push,
79              pop,
80
81              orl, //or less ?
82
83              add,
84              sub,
85              mul,
86              mov,
87              movl,
88              setne,
89              cmpl,
90              andb,
91              cmp
92  } ASM_kind;
93
94  #endif //COMPILER_KIND_H
```

**memory.h**

```
1  #ifndef __memory_h
2  #define __memory_h
3
4  void *Malloc(unsigned n);
5
6  #define NEW(type) (type *)Malloc(sizeof(type))
7
8  #endif
```

**memory.c**

```
1  #include <stdio.h>
2  #include <malloc.h>
3  #include <stdlib.h>
```

```
4
5  void *Malloc(unsigned n) {
6      void *p;
7      if (!(p = malloc(n))) {
8          fprintf(stderr, "Malloc(%d) failed.\n", n);
9          fflush(stderr);
10         abort();
11     }
12     return p;
13 }
```

**error.h**

```
1  #ifndef __error_h
2  #define __error_h
3
4  void print_error(char *error, int code, int line);
5
6
7  #endif
```

**error.c**

```
1  /**
2   * @brief Error printing.
3   *
4   * @file error.c
5   * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
6   * @date 2018-03-09
7   */
8  #include <stdio.h>
9  #include <string.h>
10 #include <stdlib.h>
11 #include "error.h"
12
13 void print_error(char *error, int code, int line){
14     fprintf(stderr, "%s at line %d\n", error, line);
15     exit(code);
16 }
```

**stack.h**

```
1  #ifndef __stack_h
2  #define __stack_h
3
4  typedef struct stack_node{
5      void *val;
6      struct stack_node *next;
7  } stack_node;
8
9  typedef struct stack{
10     struct stack_node *top;
11 } stack;
12
13 stack *init_stack();
14
15 void stack_push(stack *stack, void *val);
16
```

```
17 void *stack_pop(stack *stack);
18
19 void *stack_read(stack *stack);
20
21 #endif
```

**stack.c**

```
1  #include <stdio.h>
2  #include "stack.h"
3  #include "memory.h"
4
5  stack *init_stack(){
6
7      stack *s;
8      s = NEW(stack);
9      s->top = NULL;
10     return s;
11
12 }
13
14 void stack_push(stack *stack, void *val){
15
16     stack_node *sn;
17     sn = NEW(stack_node);
18     sn->val = val;
19     sn->next = stack->top;
20     stack->top = sn;
21
22 }
23
24 void *stack_pop(stack *stack){
25
26     stack_node *top;
27     void *val;
28
29     top = stack->top;
30     val = top->val;
31     stack->top = top->next;
32     free(top);
33     return val;
34
35 }
36
37 void *stack_read(stack *stack){
38
39     if (stack->top == NULL){
40         return NULL;
41     }
42     return stack->top->val;
43 }
```