# CC, Spring 2018
# Exam project, part 3

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

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

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

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

This report contains a total of ?? pages.

# 3. Assignment

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

Compiler (DM546)

UNIVERSITY OF
SOUTHERN DENMARK

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:

```
type t1 = t2;
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:

```
var a : int;
var b : int;
a = 5;
b = 4;
```

```
if (a > b) then
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:

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

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

# Implementation

## Weeder

```
 if ((left_term->kind == term_NUM) && (right_term->kind == term_NUM)){
          //We have an expression with two constants
          switch(expression->kind){

              case (exp_MULT):
                  temp = make_Term_num(left_term->val.num * right_term->val.num);
                  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.

```
if (expression->kind == exp_AND){

              if ((left_term->kind == term_FALSE) || (right_term->kind == term_FALSE)){
                  temp = make_Term_boolean(0);
              }
              if ((left_term->kind == term_TRUE) && (right_term->kind == term_TRUE)){
                  temp = make_Term_boolean(1);
              }
```

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.

```
if (stmt->val.ifthen.expression->kind == exp_TERM){
              if (stmt->val.ifthen.expression->val.term->kind == term_FALSE){
                  stmt = stmt->val.ifthen.statement2;
              }
              stmt = stmt->val.ifthen.statement1;
          }
```

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

```c
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.

```c
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

```c
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.

```c
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);
```

```
        }
        temp = resolve_recursive_type(s->stype->val.id_type);
    }
    type->recursive_type = 0;
    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

```
case (exp_PLUS):
    case (exp_MIN):
    case (exp_MULT):
    case (exp_DIV):
        check_exp(exp->val.ops.left);
        check_exp(exp->val.ops.right);
        if (exp->val.ops.left->stype->type == symbol_INT && exp->val.ops.right->stype->type ==
            symbol_INT){

            st = NEW(symbol_type);
            st->type = symbol_INT;
            exp->stype= st;

        } else {
            print_error("Operators in arithmetic expression are not integers", 0, exp->lineno);
        }
        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".

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

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

#### test_arithmetric_zero_division.src

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

```
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 segfaults, 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.

```
func test(n : int) : int
    var n : int;
    var a : int;
    var b : int;
    if (n : int >= 0 : int) : boolean) then
        {
            if ((n : int == 2 : int) : boolean) then
                {
                    b = 2 : int;
                }
             else
                {
                    n = (n : int+b : int) : int;
                    return n : int;
                }
        }
end test : function(n : int) : int
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.

```
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_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.

```
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.

```
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_function_wrong_return_type.src

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

```
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".

```
var a : int;
var b : int;
a = 1 : int;
b = 3 : int;
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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;
    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;
```

## Complete code listing

### main.h

```
#ifndef __main_h
#define __main_h


int main(int argc, char **argv);

#endif
```

### main.c

```
/**
 * @brief Compiler program.
 *
 * @file main.c
 * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
 * @date 2018-03-09
 */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <getopt.h>
#include "debug.h"

#include "main.h"
#include "auxiliary.h"
#include "symbol.h"
#include "tree.h"
#include "pretty.h"
#include "y.tab.h"
#include "weeder.h"
#include "typechecker.h"


int lineno;

body *theprogram;

int main(int argc, char **argv) {
    int helpflag = 0;
    int bflag = 0;
    char *cvalue = NULL;
    int index;
    int c;

    opterr = 0;

    int files[argc];

    while ((c = getopt(argc, argv, "hc:")) != -1) {
        switch (c) {
```

```
        case ’h’:
            helpflag = 1;
            files[optind - 2] = 1;
            break;
        case ’c’:
            cvalue = optarg;
            files[optind - 3] = 1;
            files[optind - 2] = 1;
            break;
        case ’?’:
            if (optopt == ’c’)
                fprintf(stderr, "Option -%c requires an argument.\n", optopt);
            else if (isprint(optopt))
                fprintf(stderr, "Unknown option ‘-%c’.\n", optopt);
            else
                fprintf(stderr,
                        "Unknown option character ‘\\x%x’.\n",
                        optopt);
            return 1;
        default:
            abort();
        }
    }
    if (helpflag) {
        system("man ./manual");
        return 0;
    }

    if (optind < argc) {
        for (int i = 1; i < argc; ++i) {
            if (files[i] == 0) {
                if (ends_with(argv[i], ".src")) {
                    freopen(argv[i], "r", stdin);
                }
            }
        }
    }

    lineno = 1;
    yyparse();

    weeder_init(theprogram);
    types = 0;
    //prettyProgram(theprogram);


    //printf("\nStarting typechecking\n\n");

#if debugflag > 0
    printf("\nStarting typechecking\n\n");
#endif
    typecheck(theprogram);
#if debugflag > 0
    printf("\nAfter typechecking\n\n");
#endif
    types = 1;
    prettyProgram(theprogram);


    printf("\n");
    return 1;
}
```

### exp.y

```

//Comments
// String followed by : is detected as a decleartion evening when within a string, the code still works, but syntax highlighting is broken.

%{
#include <stdio.h>
#include "tree.h"
#include "y.tab.h"
extern char *yytext;
//extern EXP *theexpression;
extern body *theprogram;
void yyerror() {
  printf("syntax error before %s\n",yytext);
}
%}

%union {
    int intconst;
    char *stringconst;
    struct EXP *exp;
    struct function *function;
    struct head *head;
    struct tail *tail;
    struct type *type;
    struct par_decl_list *par_decl_list;
    struct var_decl_list *var_decl_list;
    struct var_type *var_type;
    struct body *body;
    struct decl_list *decl_list;
    struct declaration *declaration;
    struct statement_list *statement_list;
    struct statement *statement;
    struct variable *variable;
    struct expression *expression;
    struct term *term;
    struct act_list *act_list;
    struct exp_list *exp_list;
}

%token <intconst> tINTCONST
%token <stringconst> tIDENTIFIER
%token EQ
%token NEQ
%token LEQ
%token GEQ
%token LT
%token GT
%token IF
%token ELSE
%token WHILE
%token RETURN
%token AND
%token OR
%token ASSIGN
%token TRUE
%token FALSE
%token _NULL
%token FUNC
%token END
%token INT
```

```
%token BOOL
%token ARRAY_OF
%token RECORD_OF
%token TYPE
%token VAR
%token OF_LENGTH
%token THEN
%token WRITE
%token ALLOCATE
%token DO



%type <function> function
%type <head> head
%type <tail> tail
%type <type> type
%type <par_decl_list> par_decl_list
%type <var_decl_list> var_decl_list
%type <var_type> var_type
%type <body> body
%type <decl_list> decl_list
%type <declaration> declaration
%type <statement_list> statement_list
%type <statement> statement
%type <expression> expression
%type <term> term
%type <act_list> act_list
%type <variable> variable
%type <exp_list> exp_list

%start program

%precedence NEG

%left AND '|'
%left EQ NEQ
%left GT LT GEQ LEQ
%left '+' '-'
%left '*' '/'
%nonassoc THEN
%nonassoc ELSE

%%
program:  body
        { theprogram = $1;}
;

function:  head body tail
       {$$ = make_Func($1, $2, $3);
       if (check_Func($1, $3) != 0){
           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);
           YYABORT;
           }}
;

head:  FUNC tIDENTIFIER '(' par_decl_list ')' ':' type
       {$$ = make_Head($2, $4, $7);}
;

tail:  END tIDENTIFIER
       {$$ = make_Tail($2);}
```

```
;

type:  tIDENTIFIER
       {$$ = make_Type_id($1);}
           | INT
       {$$ = make_Type_int();}
           | BOOL
       {$$ = make_Type_bool();}
           | ARRAY_OF type
       {$$ = make_Type_array($2);}
           | RECORD_OF '{' var_decl_list '}'
       {$$ = make_Type_record($3);}
;

par_decl_list: var_decl_list
       {$$ = make_PDL_list($1);}
           | /*empty*/
       {$$ = make_PDL_empty();}
;

var_decl_list: var_type ',' var_decl_list
       {$$ = make_VDL_list($1, $3);}
           | var_type
       {$$ = make_VDL_type($1);}
;

var_type:tIDENTIFIER ':' type
       {$$ = make_VType_id($1, $3);}
;

body:decl_list statement_list
       {$$ = make_Body($1, $2);}

;

decl_list:declaration decl_list
       {$$ = make_DL_list($1, $2);}
           | /*empty*/
       {$$ = make_DL_empty();}
;

declaration:TYPE tIDENTIFIER '=' type ';'
       {$$ = make_Decl_type($2, $4);}
           | function
       {$$ = make_Decl_func($1);}
           | VAR var_decl_list ';'
       {$$ = make_Decl_list($2);}
;

statement_list:statement
       {$$ = make_SL_statement($1);}
           | statement statement_list
       {$$ = make_SL_list($1, $2);}
;

statement:RETURN expression ';'
       {$$ = make_STMT_ret($2);}
           | WRITE expression ';'
       {$$ = make_STMT_wrt($2);}
           | ALLOCATE variable ';'
       {$$ = make_STMT_allocate_var($2);}
           | ALLOCATE variable OF_LENGTH expression ';'
       {$$ = make_STMT_allocate_length($2, $4);}
```

```
           | variable ’=’ expression ’;’
       {$$ = make_STMT_assign($1, $3);}
           | IF expression THEN statement  %prec THEN
       {$$ = make_STMT_if($2, $4);}
           | IF expression THEN statement ELSE statement
       {$$ = make_STMT_if_else($2, $4, $6);}
           | WHILE expression DO statement
       {$$ = make_STMT_while($2, $4);}
           | ’{’ statement_list ’}’
       {$$ = make_STMT_list($2);}
;

variable:tIDENTIFIER
       {$$ = make_Var_id($1);}
           | variable ’[’ expression ’]’
       {$$ = make_Var_exp($1, $3);}
           | variable ’.’ tIDENTIFIER
       {$$ = make_Var_record($1, $3);}
;

expression:expression ’+’ expression
       {$$ = make_EXP(exp_PLUS, $1, $3);}
           | expression ’-’ expression
       {$$ = make_EXP(exp_MIN, $1, $3);}
           | expression ’*’ expression
       {$$ = make_EXP(exp_MULT, $1, $3);}
           | expression ’/’ expression
       {$$ = make_EXP(exp_DIV, $1, $3);}
           | ’(’ expression ’)’
       {$$ = $2;}
           | expression EQ expression
       {$$ = make_EXP(exp_EQ, $1, $3);}
           | expression NEQ expression
       {$$ = make_EXP(exp_NEQ, $1, $3);}
           | expression GT expression
       {$$ = make_EXP(exp_GT, $1, $3);}
           | expression LT expression
       {$$ = make_EXP(exp_LT, $1, $3);}
           | expression GEQ expression
       {$$ = make_EXP(exp_GEQ, $1, $3);}
           | expression LEQ expression
       {$$ = make_EXP(exp_LEQ, $1, $3);}
           | expression AND expression
       {$$ = make_EXP(exp_AND, $1, $3);}
           | expression ’|’’|’ expression
       {$$ = make_EXP(exp_OR, $1, $4);}
           | ’-’ expression %prec NEG
       {$$ = make_EXP_neg($2);}
           | term
       {$$ = make_EXP_term($1);}
;

term:      tINTCONST
       {$$ = make_Term_num($1);}
           | ’(’ expression ’)’
       {$$ = make_Term_par($2);}
           | ’!’ term
       {$$ = make_Term_not($2);}
           | ’|’ expression ’|’
       {$$ = make_Term_abs($2);}
           | TRUE
       {$$ = make_Term_boolean(1);}
           | FALSE
       {$$ = make_Term_boolean(0);}
           | _NULL
       {$$ = make_Term_null();}
           | variable
       {$$ = make_Term_variable($1);}
           | tIDENTIFIER '(' act_list ')'
       {$$ = make_Term_list($1, $3);}
;

act_list:exp_list
       {$$ = make_Act_list($1);}
           | /*empty*/
       {$$ = make_Act_empty();}
;

exp_list:expression
       {$$ = make_ExpL_exp($1);}
           | expression ',' exp_list
       {$$ = make_ExpL_list($1, $3);}
;

%%
```

### exp.l

```
%{
#include "y.tab.h"
#include <string.h>

extern int lineno;
extern int fileno();
int nested_comment = 0;

%}

%x COMMENT_SINGLE
%x COMMENT_MULTI

%option noyywrap nounput noinput

/* abbreviation of symbols we match on, TO BE EXPANDED */
SYMBOLS [+\-*\/\(\)\[\]{}!\|,\.=;:]

%%
[ \t]+       /* ignore */;
\n            lineno++;

{SYMBOLS}     return yytext[0];

"=="          return EQ;
"!="          return NEQ;
"<"           return LT;
"<="          return LEQ;
">"           return GT;
">="          return GEQ;
"if"          return IF;
"else"        return ELSE;
"while"       return WHILE;
"return"      return RETURN;
"&&"          return AND;
"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"
```

```c

#define HashSize 317
/* SYMBOL will be extended later.
Function calls will take more parameters later.
*/

typedef enum { symbol_ID,
              symbol_INT,
              symbol_BOOL,
              symbol_RECORD,
              symbol_ARRAY,
              symbol_NULL,
              symbol_FUNCTION,
              symbol_UNKNOWN } SYMBOL_type;

typedef struct SYMBOL {
    char *name;
    int value;
    struct SYMBOL *next;
    struct symbol_type *stype;
} SYMBOL;

typedef struct symbol_table {
    SYMBOL *table[HashSize];
    struct symbol_table *next;
} symbol_table;


typedef struct symbol_type {
    SYMBOL_type type;
    union {
        struct type *array_type;
        struct var_decl_list *record_type;
        struct type *id_type;
        struct {
            struct type *ret_type;
            struct par_decl_list *pdl;
            struct function *func;
        } func_type;
    } val;
    int printed;
} symbol_type;

int hash(char *str);

struct symbol_table *init_symbol_table();

struct symbol_table *scope_symbol_table(symbol_table *t);

SYMBOL *put_symbol(symbol_table *t, char *name, int value, symbol_type *st);

SYMBOL *get_symbol(symbol_table *t, char *name);

void dump_symbol_table(symbol_table *t);

SYMBOL *check_local(symbol_table *t, char *name);

void print_symbol(SYMBOL *symbol);

#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) {
    symbol_table *newTable = init_symbol_table();
    newTable->next = t;
    return newTable;
}

/**
 * @brief put_symbol takes a hash table and a string, name, as arguments and
 * inserts name into the hash table together with the associated value value.
 * A pointer to the SYMBOL value which stores name is returned.
 *
 * @param t Pointer to hash table.
 * @param name
 * @param value
 * @param st
 * @return SYMBOL*
 */
SYMBOL *put_symbol(symbol_table *t, char *name, int value, symbol_type *st) {
    if (t == NULL) {
        return NULL;
    }
    SYMBOL *localCheck = check_local(t, name);
    //Symbol already exists
    if (localCheck != NULL) {
        return localCheck;
    } else {
        int hashValue = hash(name);
        //pretty("Putting symbol with name: %s, value: %d, type: %d, table: %p, hash: %d\n",
            name, value, st->type, t, hashValue);

        SYMBOL *symbol = Malloc(sizeof(SYMBOL));
        symbol->name = name;
        symbol->value = value;
        symbol->stype = st;
        symbol->next = Malloc(sizeof(SYMBOL));

        //Placed in front of the list
        symbol->next = t->table[hashValue];
        t->table[hashValue] = symbol;
        return symbol;
    }
}

/*
    getSymbol takes a hash table and a string name as arguments and searches for
    name in the following manner: First search for name in the hash table which
    is one of the arguments of the function call. If name is not there, continue the
    search in the next hash table. This process is repeatedly recursively. If name has
    not been found after the root of the tree (see Fig. 1) has been checked, the result
    NULL is returned. If name is found, return a pointer to the SYMBOL value in
    which name is stored
    */
SYMBOL *get_symbol(symbol_table *t, char *name) {
    //   First check if t is null
    //pretty("Getting symbol %s, from table %p\n", name, t);
    if (t == NULL) {
        //pretty("Table is null\n");
        return NULL;
    }

    SYMBOL *localCheck = check_local(t, name);

    //Symbol in local table?
    if (localCheck != NULL) {
        //pretty("Trying local check, type: %d\n", localCheck->stype->type);
        return localCheck;
    }

    if (t->next != NULL) {
        //pretty("Checking next\n");
        return get_symbol(t->next, name);
    }

    //Symbol does not exists
    return NULL;
}

/*
 * dumpSymbolTable takes a pointer to a hash table t as argument and prints all
 * the (name, value) pairs that are found in the hash tables from t up to the root.
 * Hash tables are printed one at a time. The printing should be formatted in a nice
 * way and is intended to be used for debugging (of other parts of the compiler).
*/
void dump_symbol_table(symbol_table *t) {
    if (t == NULL) {
        return;
    }

    printf("Printing symbol table:\n\n");

    for (int i = 0; i < HashSize; i++) {
        if (t->table[i] != NULL) {
            print_symbol(t->table[i]);
            printf("\n");
        }
    }
    printf("\n");

    dump_symbol_table(t->next);
}

/*
 * Check the current table we are in for a value
 */
SYMBOL *check_local(symbol_table *t, char *name) {
    int hashValue = hash(name);

    SYMBOL *symbol = t->table[hashValue];
    if (symbol == NULL) {
        //pretty("Local symbol is null\n");
        return NULL;
    } else {
        while (symbol != NULL) {
            if (strcmp(name, symbol->name) == 0) {
                //pretty("Compared %s and %s, success\n", name, symbol->name);
                return symbol;
            }
            symbol = symbol->next;
        }
    }

    //Hash value for the symbol exists, but the symbol is not in the table
    return NULL;
}

void print_symbol(SYMBOL *symbol) {
    printf("(%s, %i)", symbol->name, symbol->value);
```

```
}
```

### tree.h

```
#ifndef __tree_h
#define __tree_h

#include "kind.h"
#include "symbol.h"

typedef struct function {
    struct symbol_table*table;
    struct symbol_type *stype;
    int lineno;
    struct head *head;
    struct body *body;
    struct tail *tail;

} function;

typedef struct head {
    struct symbol_table*table;
    struct symbol_type *stype;
    int lineno;
    char *id;
    struct par_decl_list *list;
    struct type *type;
    int args;

} head;

typedef struct tail {
    struct symbol_table*table;
    int lineno;
    char *id;
} tail;

typedef struct type {
    struct symbol_table*table;
    struct symbol_type *stype;
    int recursive_type;
    int lineno;
    TYPE_kind kind;
    union {
        char *id;
        struct type *type;
        struct var_decl_list *list;
    } val;
} type;

typedef struct par_decl_list {
    struct symbol_table*table;
    int lineno;
    PDL_kind kind;
    struct var_decl_list *list;
} par_decl_list;

typedef struct var_decl_list {
    struct symbol_table*table;
    int lineno;
    VDL_kind kind;
    struct var_decl_list *list;
    struct var_type *vartype;
} var_decl_list;

typedef struct var_type {
    struct symbol_table*table;
    struct SYMBOL *symbol;
    int lineno;
    char *id;
    struct type *type;
} var_type;

typedef struct body {
    struct symbol_table*table;
    int lineno;
    struct decl_list *d_list;
    struct statement_list *s_list;
} body;

typedef struct decl_list {
    struct symbol_table*table;
    int lineno;
    DL_kind kind;
    struct declaration *decl;
    struct decl_list *list;
} decl_list;

typedef struct declaration {
    struct symbol_table*table;
    int lineno;
    DECL_kind kind;
    union {
        struct {
            char *id;
            struct type *type;
        } type;
        struct function *function;
        var_decl_list *list;
    } val;

} declaration;

typedef struct statement_list {
    struct symbol_table*table;
    int lineno;
    SL_kind kind;
    struct statement *statement;
    struct statement_list *list;

} statement_list;

typedef struct statement {
    struct symbol_table*table;
    int lineno;
    STATEMENT_kind kind;
    struct function *function;
    int contains_ret;
    union {
        struct expression *ret;
        struct expression *wrt;
        struct {
            struct variable *variable;
            struct expression *length;
        } allocate;

        struct {
            struct variable *variable;
            struct expression *expression;
        } assignment;

        struct {
            struct expression *expression;
            struct statement *statement1;
            struct statement *statement2;
        } ifthen;

        struct {
            struct expression *expression;
            struct statement *statement;
        } loop;

        struct statement_list *list;
    } val;

} statement;

typedef struct variable {
    struct symbol_table*table;
    struct symbol_type *stype;
    int lineno;
    char *id;
    Var_kind kind;
    union {
        struct {
            struct variable *var;
            struct expression *exp;
        } exp;
        struct {
            struct variable *var;
            char *id;
        } record;
    } val;
} variable;

typedef struct expression {
    struct symbol_table*table;
    struct symbol_type *stype;
    int lineno;
    EXP_kind kind;
    union {
        struct {
            struct expression *left;
            struct expression *right;
        } ops;
        struct term *term;
        struct expression *neg;
    } val;

} expression;

typedef struct term {
    struct symbol_table*table;
    struct symbol_type *stype;
    int lineno;
    TERM_kind kind;
    union {
        int num;
        struct expression *expression;
        struct term *term_not;
        struct variable *variable;
        struct {
            char *id;
            struct act_list *list;
        } list;
    } val;
} term;

typedef struct act_list {
    struct symbol_table*table;
    int lineno;
    AL_kind kind;
    struct exp_list *list;
} act_list;

typedef struct exp_list {
    struct symbol_table *table;
    int lineno;
    EL_kind kind;
    struct expression *expression;
    struct exp_list *list;
} exp_list;

function *make_Func(head *h, body *b, tail *t);

head *make_Head(char *id, par_decl_list *pdl, type *t);

tail *make_Tail(char *id);

type *make_Type_id(char *id);
type *make_Type_int();
type *make_Type_bool();
type *make_Type_array(type *t);
type *make_Type_record(var_decl_list *vdl);

par_decl_list *make_PDL_list(var_decl_list *vdl);
par_decl_list *make_PDL_empty();

var_decl_list *make_VDL_list(var_type *vt, var_decl_list *vdl);
var_decl_list *make_VDL_type(var_type *vt);

var_type *make_VType_id(char *id, type *t);

body *make_Body(decl_list *dl, statement_list *sl);

decl_list *make_DL_list(declaration *d, decl_list *dl);
decl_list *make_DL_empty();

declaration *make_Decl_type(char *id, type *t);
declaration *make_Decl_func(function *f);
declaration *make_Decl_list(var_decl_list *vdl);

statement_list *make_SL_statement(statement *s);
statement_list *make_SL_list(statement *s, statement_list *sl);

statement *make_STMT_ret(expression *e);
statement *make_STMT_wrt(expression *e);
statement *make_STMT_allocate_var(variable *v);
statement *make_STMT_allocate_length(variable *v, expression *e);
statement *make_STMT_assign(variable *v, expression *e);
statement *make_STMT_if(expression *e, statement *s);
statement *make_STMT_if_else(expression *e, statement *s1, statement *s2);
statement *make_STMT_while(expression *e, statement *s);
statement *make_STMT_list(statement_list *sl);

variable *make_Var_id(char *id);
variable *make_Var_exp(variable *var, expression *expression);
variable *make_Var_record(variable *var, char *id);

expression *make_EXP(EXP_kind kind, expression *left, expression *right);
expression *make_EXP_term(term *term);
expression *make_EXP_neg(expression *neg);

term *make_Term_num(int intconst);
term *make_Term_par(expression *expression);
term *make_Term_not(term *term);
term *make_Term_abs(expression *expression);
term *make_Term_boolean(int bool);
term *make_Term_null();
term *make_Term_variable(variable *var);
term *make_Term_list(char *id, act_list *list);

act_list *make_Act_list(exp_list *list);
act_list *make_Act_empty();

exp_list *make_ExpL_exp(expression *expression);
exp_list *make_ExpL_list(expression *expression, exp_list *list);

int check_Func(head *head, tail *tail);

#endif
```

### tree.c

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

extern int lineno;

function *make_Func(head *h, body *b, tail *t) {
    function *f;
    f = NEW(function);
    f->lineno = lineno;
    f->head = h;
    f->body = b;
    f->tail = t;
    return f;
}

head *make_Head(char *id, par_decl_list *pdl, type *t) {
    head *h;
    h = NEW(head);
    h->lineno = lineno;
    h->id = id;
    h->list = pdl;
    h->type = t;
    return h;
}

tail *make_Tail(char *id) {
    tail *t;
    t = NEW(tail);
    t->lineno = lineno;
    t->id = id;
    return t;
}

type *make_Type_id(char *id) {
    type *t;
    t = NEW(type);
    t->lineno = lineno;
    t->kind = type_ID;
    t->val.id = id;
    return t;
}

type *make_Type_int() {
    type *t;
    t = NEW(type);
    t->lineno = lineno;
    t->kind = type_INT;
    return t;
}

type *make_Type_bool() {
    type *t;
    t = NEW(type);
    t->lineno = lineno;
    t->kind = type_BOOl;
    return t;
}

type *make_Type_array(type *t1) {
    type *t;
    t = NEW(type);
    t->lineno = lineno;
    t->kind = type_ARRAY;
    t->val.type = t1;
    return t;
}

type *make_Type_record(var_decl_list *vdl) {
    type *t;
    t = NEW(type);
    t->lineno = lineno;
    t->kind = type_RECORD;
    t->val.list = vdl;
    return t;
}

par_decl_list *make_PDL_list(var_decl_list *vdl) {
    par_decl_list *pdl;
    pdl = NEW(par_decl_list);
    pdl->lineno = lineno;
    pdl->kind = pdl_LIST;
    pdl->list = vdl;
    return pdl;
}

par_decl_list *make_PDL_empty() {
    par_decl_list *pdl;
    pdl = NEW(par_decl_list);
    pdl->lineno = lineno;
    pdl->kind = pdl_EMPTY;
    pdl->list = NULL;
    return pdl;
}

var_decl_list *make_VDL_list(var_type *vt, var_decl_list *vdl1) {
    var_decl_list *vdl;
    vdl = NEW(var_decl_list);
    vdl->lineno = lineno;
    vdl->kind = vdl_LIST;
    vdl->list = vdl1;
    vdl->vartype = vt;
    return vdl;
}

var_decl_list *make_VDL_type(var_type *vt) {
    var_decl_list *vdl;
    vdl = NEW(var_decl_list);
    vdl->lineno = lineno;
    vdl->kind = vdl_TYPE;
    vdl->list = NULL;
    vdl->vartype = vt;
    return vdl;
}

var_type *make_VType_id(char *id, type *t) {
    var_type *vt;
    vt = NEW(var_type);
    vt->lineno = lineno;
    vt->id = id;
    vt->type = t;
    return vt;
}

body *make_Body(decl_list *dl, statement_list *sl) {
    body *b;
    b = NEW(body);
    b->lineno = lineno;
    b->d_list = dl;
    b->s_list = sl;
    return b;
}

decl_list *make_DL_list(declaration *d, decl_list *dl1) {
    decl_list *dl;
    dl = NEW(decl_list);
    dl->lineno = lineno;
    dl->kind = dl_LIST;
    dl->decl = d;
    dl->list = dl1;
    return dl;
}

decl_list *make_DL_empty() {
    decl_list *dl;
    dl = NEW(decl_list);
    dl->lineno = lineno;
    dl->kind = dl_EMPTY;
    dl->decl = NULL;
    dl->list = NULL;
    return dl;
}

declaration *make_Decl_type(char *id, type *t) {
    declaration *d;
    d = NEW(declaration);
    d->lineno = lineno;
    d->kind = decl_TYPE;
    d->val.type.id = id;
    d->val.type.type = t;
    return d;
}

declaration *make_Decl_func(function *f) {
    declaration *d;
    d = NEW(declaration);
    d->lineno = lineno;
    d->kind = decl_FUNC;
    d->val.function = f;
    return d;
}

declaration *make_Decl_list(var_decl_list *vdl) {
    declaration *d;
    d = NEW(declaration);
    d->lineno = lineno;
    d->kind = decl_VAR;
    d->val.list = vdl;
    return d;
}

statement_list *make_SL_statement(statement *s) {
    statement_list *sl;
    sl = NEW(statement_list);
    sl->lineno = lineno;
    sl->kind = sl_STATEMENT;
    sl->statement = s;
    sl->list = NULL;
    return sl;
}

statement_list *make_SL_list(statement *s, statement_list *sl1) {
    statement_list *sl;
    sl = NEW(statement_list);
    sl->lineno = lineno;
    sl->kind = sl_LIST;
    sl->statement = s;
    sl->list = sl1;
    return sl;
}

statement *make_STMT_ret(expression *e) {
    statement *s;
    s = NEW(statement);
    s->lineno = lineno;
    s->kind = statement_RETURN;
    s->val.ret = e;
    return s;
}
```

```c

statement *make_STMT_wrt(expression *e) {
    statement *s;
    s = NEW(statement);
    s->lineno = lineno;
    s->kind = statement_WRITE;
    s->val.wrt = e;
    return s;
}

statement *make_STMT_allocate_var(variable *v) {
    statement *s;
    s = NEW(statement);
    s->lineno = lineno;
    s->kind = statement_ALLOCATE;
    s->val.allocate.variable = v;
    s->val.allocate.length = NULL;
    return s;
}

statement *make_STMT_allocate_length(variable *v, expression *e) {
    statement *s;
    s = NEW(statement);
    s->lineno = lineno;
    s->kind = statement_ALLOCATE_LENGTH;
    s->val.allocate.variable = v;
    s->val.allocate.length = e;
    return s;
}

statement *make_STMT_assign(variable *v, expression *e) {
    statement *s;
    s = NEW(statement);
    s->lineno = lineno;
    s->kind = statement_ASSIGNMENT;
    s->val.assignment.variable = v;
    s->val.assignment.expression = e;
    return s;
}

statement *make_STMT_if(expression *e, statement *s1) {
    statement *s;
    s = NEW(statement);
    s->lineno = lineno;
    s->kind = statement_IF;
    s->val.ifthen.expression = e;
    s->val.ifthen.statement1 = s1;
    s->val.ifthen.statement2 = NULL;
    return s;
}

statement *make_STMT_if_else(expression *e, statement *s1, statement *s2) {
    statement *s;
    s = NEW(statement);
    s->lineno = lineno;
    s->kind = statement_IF_ELSE;
    s->val.ifthen.expression = e;
    s->val.ifthen.statement1 = s1;
    s->val.ifthen.statement2 = s2;
    return s;
}

statement *make_STMT_while(expression *e, statement *s1) {
    statement *s;
    s = NEW(statement);
    s->lineno = lineno;
    s->kind = statement_WHILE;
    s->val.loop.expression = e;
    s->val.loop.statement = s1;
    return s;
}

statement *make_STMT_list(statement_list *sl) {
    statement *s;
    s = NEW(statement);
    s->lineno = lineno;
    s->kind = statement_LIST;
    s->val.list = sl;
    return s;
}

expression *make_EXP(EXP_kind kind, expression *left, expression *right) {
    expression *e;
    e = NEW(expression);
    e->lineno = lineno;
    e->kind = kind;
    e->val.ops.left = left;
    e->val.ops.right = right;
    return e;
}

expression *make_EXP_term(term *term) {
    expression *e;
    e = NEW(expression);
    e->lineno = lineno;
    e->kind = exp_TERM;
    e->val.term = term;
    return e;
}

//Negation of x is the same as 0-x, so we make a minus node
expression *make_EXP_neg(expression *neg) {

    expression *zero = make_EXP_term(make_Term_num(0));

    expression *minus = make_EXP(exp_MIN, zero, neg);

    return minus;
}

term *make_Term_num(int intconst) {
    term *t;
    t = NEW(term);
    t->lineno = lineno;
    t->kind = term_NUM;
    t->val.num = intconst;
    return t;
}

term *make_Term_par(expression *expression) {
    term *t;
    t = NEW(term);
    t->lineno = lineno;
    t->kind = term_PAR;
    t->val.expression = expression;
    return t;
```

```
}

term *make_Term_not(term *term) {
    struct term *t;
    t = NEW(struct term);
    t->lineno = lineno;
    t->kind = term_NOT;
    t->val.term_not = term;
    return t;
}

term *make_Term_abs(expression *expression) {
    term *t;
    t = NEW(term);
    t->lineno = lineno;
    t->kind = term_ABS;
    t->val.expression = expression;
    return t;
}

term *make_Term_boolean(int bool) {
    term *t;
    t = NEW(term);
    t->lineno = lineno;
    if (bool == 1) {
        t->kind = term_TRUE;
        return t;
    }
    t->kind = term_FALSE;
    return t;
}

term *make_Term_null() {
    term *t;
    t = NEW(term);
    t->lineno = lineno;
    t->kind = term_NULL;
    return t;
}

term *make_Term_variable(variable *var) {
    term *t;
    t = NEW(term);
    t->lineno = lineno;
    t->kind = term_VAR;
    t->val.variable = var;
    return t;
}

term *make_Term_list(char *id, act_list *list) {
    term *t;
    t = NEW(term);
    t->lineno = lineno;
    t->kind = term_LIST;
    t->val.list.id = id;
    t->val.list.list = list;
    return t;
}

act_list *make_Act_list(exp_list *list) {
    act_list *al;
    al = NEW(act_list);
    al->lineno = lineno;
    al->kind = al_LIST;
    al->list = list;
    return al;
}
act_list *make_Act_empty() {
    act_list *al;
    al = NEW(act_list);
    al->lineno = lineno;
    al->kind = al_EMPTY;
    al->list = NULL;
    return al;
}

exp_list *make_ExpL_exp(expression *expression) {
    exp_list *el;
    el = NEW(exp_list);
    el->lineno = lineno;
    el->kind = el_EXP;
    el->expression = expression;
    el->list = NULL;
    return el;
}

exp_list *make_ExpL_list(expression *expression, exp_list *list) {
    exp_list *el;
    el = NEW(exp_list);
    el->lineno = lineno;
    el->kind = el_LIST;
    el->expression = expression;
    el->list = list;
    return el;
}

variable *make_Var_id(char *id) {
    variable *v;
    v = NEW(variable);
    v->lineno = lineno;
    v->kind = var_ID;
    v->id = id;
    return v;
}

variable *make_Var_exp(variable *var, expression *expression) {
    variable *v;
    v = NEW(variable);
    v->lineno = lineno;
    v->kind = var_EXP;
    v->val.exp.var = var;
    v->val.exp.exp = expression;
    return v;
}

variable *make_Var_record(variable *var, char *id) {
    variable *v;
    v = NEW(variable);
    v->lineno = lineno;
    v->kind = var_RECORD;
    v->val.record.var = var;
    v->val.record.id = id;
    return v;
}

int check_Func(head *head, tail *tail) {
    if (strcmp(head->id, tail->id) == 0) {
        return 0;
    }
    return 1;
}
```

### pretty.h

```
#ifndef __pretty_h
#define __pretty_h

#include "tree.h"

extern int types;

void prettyProgram(body *body);

void prettyFunc(function *f);

void prettyHead(head *h);

void prettyTail(tail *t);

void prettyType(type *t);

void prettyPDL(par_decl_list *pdl);

void prettyVDL(var_decl_list *vdl);

void prettyVT(var_type *vt);

void prettyBody(body *b);

void prettyDL(decl_list *dl);

void prettyDecl(declaration *d);

void prettySL(statement_list *sl);

void prettySTMT(statement *s);

void prettyVar(variable *v);

void prettyEXP(expression *e);

void prettyTerm(term *t);

void prettyAL(act_list *al);

void prettyEL(exp_list *el);

void indent();

void prettySymbol(symbol_table *table, char *id, int line);

void prettyStype(symbol_type *stype, int line);

#endif
```

### pretty.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");
        break;

    case type_BOOl:
        printf("bool");
        break;

    case type_ARRAY:
        printf("array of ");
        prettyType(t->val.type);
        break;

    case type_RECORD:
        printf("record of { ");
        prettyVDL(t->val.list);
        printf(" }");
        break;
    }
}

void prettyPDL(par_decl_list *pdl) {

    if (pdl->kind == pdl_LIST){
        prettyVDL(pdl->list);
    }
}

void prettyVDL(var_decl_list *vdl) {
    switch (vdl->kind) {
        case vdl_LIST:
            prettyVT(vdl->vartype);
            printf(", ");
            prettyVDL(vdl->list);
            break;

        case vdl_TYPE:
            prettyVT(vdl->vartype);
            break;
    }
}

void prettyVT(var_type *vt) {
    printf("%s : ", vt->id);
    prettyType(vt->type);
}

void prettyBody(body *b) {
    prettyDL(b->d_list);
    prettySL(b->s_list);
}

void prettyDL(decl_list *dl) {
    switch (dl->kind) {
    case dl_LIST:
        prettyDecl(dl->decl);
        prettyDL(dl->list);
        break;

    case dl_EMPTY:
        break;
    }
}

void prettyDecl(declaration *d) {
    indent();
    switch (d->kind) {
    case decl_TYPE:
        printf("type %s = ", d->val.type.id);
        prettyType(d->val.type.type);
        printf(";\n");
        break;

    case decl_FUNC:
        prettyFunc(d->val.function);
        break;

    case decl_VAR:
        printf("var ");
        prettyVDL(d->val.list);
        printf(";\n");
        break;
    }
}

void prettySL(statement_list *sl) {
    switch (sl->kind) {
    case sl_STATEMENT:
        prettySTMT(sl->statement);
        break;

    case sl_LIST:
        prettySTMT(sl->statement);
        prettySL(sl->list);
        break;
    }
}

void prettySTMT(statement *s) {

    if (s->kind != statement_LIST) {
        indent();
    }

    switch (s->kind) {
    case statement_RETURN:
        printf("return ");
        prettyEXP(s->val.ret);
        printf(";\n");
        break;

    case statement_WRITE:
        printf("write ");
        prettyEXP(s->val.wrt);
        printf(";\n");
        break;

    case statement_ALLOCATE:
        printf("allocate ");
        prettyVar(s->val.allocate.variable);
        printf(";\n");
        break;

    case statement_ALLOCATE_LENGTH:
        printf("allocate ");
        prettyVar(s->val.allocate.variable);
        printf(" of length ");
        prettyEXP(s->val.allocate.length);
        printf(";\n");
        break;

    case statement_ASSIGNMENT:
        prettyVar(s->val.assignment.variable);
        printf(" = ");
        prettyEXP(s->val.assignment.expression);
        printf(";\n");
        break;

    case statement_IF:
        printf("if (");
        prettyEXP(s->val.ifthen.expression);
        printf(") then\n");
        indent_depth++;
        prettySTMT(s->val.ifthen.statement1);
        indent_depth--;
        break;

    case statement_IF_ELSE:
        printf("if (");
        prettyEXP(s->val.ifthen.expression);
        printf(") then\n");
        indent_depth++;
        prettySTMT(s->val.ifthen.statement1);
        indent_depth--;
        indent();
        printf(" else\n");
        indent_depth++;
        prettySTMT(s->val.ifthen.statement2);
        indent_depth--;
        break;

    case statement_WHILE:
        printf("while (");
        prettyEXP(s->val.loop.expression);
        printf(") do\n");
        prettySTMT(s->val.loop.statement);
        break;

    case statement_LIST:
        indent();
        printf("{\n");
        indent_depth++;
        prettySL(s->val.list);
        indent_depth--;
        indent();
        printf("}\n");
        break;
    }
}

void prettyVar(variable *v) {
    switch (v->kind) {

    case var_ID:
        printf("%s", v->id);
        break;

    case var_EXP:
        prettyVar(v->val.exp.var);
        printf("[");
        prettyEXP(v->val.exp.exp);
        printf("]");
        break;

    case var_RECORD:
        prettyVar(v->val.record.var);
        printf(".%s", v->val.record.id);
        break;
    }
}

void prettyEXP(expression *e) {
    exp_depth++;

    if (e->kind == exp_TERM){
        prettyTerm(e->val.term);
        return;
    }

    if(exp_depth > 1 && inside_par == 0){
        printf("(");
    }
    switch (e->kind) {

    case exp_MULT:
        prettyEXP(e->val.ops.left);
        printf("*");
        prettyEXP(e->val.ops.right);
        break;

    case exp_DIV:
        prettyEXP(e->val.ops.left);
        printf("/");
        prettyEXP(e->val.ops.right);
        break;

    case exp_PLUS:
        prettyEXP(e->val.ops.left);
        printf("+");
        prettyEXP(e->val.ops.right);
        break;

    case exp_MIN:
        prettyEXP(e->val.ops.left);
        printf("-");
        prettyEXP(e->val.ops.right);
        break;

    case exp_EQ:
        prettyEXP(e->val.ops.left);
        printf(" == ");
        prettyEXP(e->val.ops.right);
        break;

    case exp_NEQ:
        prettyEXP(e->val.ops.left);
        printf(" != ");
        prettyEXP(e->val.ops.right);
        break;

    case exp_GT:
        prettyEXP(e->val.ops.left);
        printf(" > ");
        prettyEXP(e->val.ops.right);
        break;

    case exp_LT:
        prettyEXP(e->val.ops.left);
        printf(" < ");
        prettyEXP(e->val.ops.right);
        break;

    case exp_GEQ:
        prettyEXP(e->val.ops.left);
        printf(" >= ");
        prettyEXP(e->val.ops.right);
        break;

    case exp_LEQ:
        prettyEXP(e->val.ops.left);
        printf(" <= ");
        prettyEXP(e->val.ops.right);
        break;

    case exp_AND:
        prettyEXP(e->val.ops.left);
        printf(" && ");
        prettyEXP(e->val.ops.right);
        break;

    case exp_OR:
        prettyEXP(e->val.ops.left);
        printf(" || ");
        prettyEXP(e->val.ops.right);
        break;

    }
    if(exp_depth > 1 && inside_par == 0){
        printf(")");

    }
    exp_depth--;

    if (types){
       // printf("\nCalling printStype in expression");
        printf( " : ");
        prettyStype(e->stype, e->lineno);
    }

}

void prettyTerm(term *t) {
    switch (t->kind) {

    case term_VAR:
        prettyVar(t->val.variable);
        break;

    case term_LIST:
        printf("%s(", t->val.list.id);
        prettyAL(t->val.list.list);
        printf(")");
        break;

    case term_PAR:
        if (exp_depth > 1){
            printf("(");
        }
        inside_par = 1;
        prettyEXP(t->val.expression);
        inside_par = 0;

        if (exp_depth > 1){
            printf("(");
        }
        break;

    case term_NOT:
        printf("!");
        prettyTerm(t->val.term_not);
        break;

    case term_ABS:
        printf("|");
        prettyEXP(t->val.expression);
        printf("|");
        break;

    case term_TRUE:
        printf("true");
        break;

    case term_FALSE:
        printf("false");
        break;

    case term_NULL:
        printf("null");
        break;

    case term_NUM:
        printf("%i", t->val.num);
        break;
    }

    if (types){
        //printf("\nCalling printStype in term");
                // Haps
        printf(" : ");
        prettyStype(t->stype, t->lineno);
    }

}

void prettyAL(act_list *al) {
    switch (al->kind) {

    case al_LIST:
        prettyEL(al->list);
        break;

    case al_EMPTY:
        break;
    }
}

void prettyEL(exp_list *el) {
    switch (el->kind) {

    case el_EXP:
        prettyEXP(el->expression);
        break;

    case el_LIST:
        prettyEXP(el->expression);
        printf(", ");
        prettyEL(el->list);
        break;
    }
}

void indent() {

    int spaces = 0;
    while (spaces < (indent_depth * 4)) {
        printf(" ");
        spaces++;
    }
}

void prettySymbol(symbol_table *table, char *id, int line){

    SYMBOL *s;
    s = get_symbol(table, id);
    if (s == NULL || s->stype == NULL){
        print_error("Symbol is not recognized", 0, line);
    }
    printf(" : ");
    prettyStype(s->stype, line);
}

void prettyStype(symbol_type *stype, int line){
   // printf("\nPrintSType of type: %d: ", stype->type);
    if (stype->printed){
        return;
    }
    stype->printed = 1;
    switch(stype->type){

        case (symbol_ID):
            prettyType(stype->val.id_type);
            break;

        case (symbol_INT):
            printf("int");
            break;

        case (symbol_BOOL):
            printf("boolean");
            break;

        case (symbol_RECORD):
            printf("record of {");
            prettyVDL(stype->val.record_type);
            printf("}");
            break;

        case (symbol_ARRAY):
            printf("array[");
            prettyStype(stype->val.array_type->stype, line);
            printf("]");
            break;

        case (symbol_FUNCTION):
            printf("function(");
            prettyPDL(stype->val.func_type.pdl);
            printf(") : ");
            prettyStype(stype->val.func_type.ret_type->stype, line);
            break;

        case (symbol_NULL):
            printf("NULL");
            break;

        // Should never happen
        case (symbol_UNKNOWN):
            printf("unknown");
            print_error("Unknown symbol type", 0, line);
            break;

    }
    stype->printed = 0;

}
```

### typechecker.h

```
#ifndef __typechecker_h
#define __typechecker_h
#include "tree.h"


int typecheck(body *program);




#endif
```

### typechecker.c

```
/**
 * @brief
 *
 * @file typechecker.c
 * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
 * @date 2018-03-09
 */

#include <stdio.h>
#include <stdlib.h>
#include "debug.h"
#include "check.h"
#include "typechecker.h"
#include "error.h"
#include "symbol.h"
#include "setup.h"
#include "pickup.h"

int typecheck(body *program) {

    symbol_table *table;
    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);

```

```
 



#endif
```

### weeder.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "tree.h"
#include "weeder.h"
#include "error.h"
#include "stack.h"

body *theprogram;

struct stack *function_stack;

/**
 * What we want to weed (TO BE EXPANDED):
 *      [IF] statements:
 *          true && term   - should be term
 *          false && term  - should be false
 *          true || term   - should be true
 *          false || term  - should be true
 *          break/continue - error (for now, break could possibly be implemented)
 *
 *      [CONSTANTS]
 *          F.x. 5*2       - should be 10, can be calculated during compile time
 *
 *      [FUNCTIONS]
 *          no return      - error
 *
 *
 */


body *weeder_init(body *program){

    function_stack = init_stack();
    program = weed_body(program);
    return program;


}

body *weed_body(body *body){
    ////printf("Weeding body\n");
    body->d_list = weed_dlist(body->d_list);
    body->s_list = weed_slist(body->s_list);
    return body;
}

function *weed_function(function *func){

    stack_push(function_stack, func);

    func->body = weed_body(func->body);
    func->head = weed_head(func->head);

```

```c
    stack_pop(function_stack);

    return func;

}

head *weed_head(head *head){
    ////printf("Weeding head\n");
    head->list = weed_pdl(head->list);
    head->type = weed_type(head->type);
    return head;

}

type *weed_type(type *type){

    ////printf("Weeding type\n");
    switch (type->kind){
        case (type_ARRAY):
            type->val.type = weed_type(type->val.type);
            break;

        case (type_RECORD):
            type->val.list = weed_vdl(type->val.list);
            break;

        default:
            break;
    }
    return type;

}

par_decl_list *weed_pdl(par_decl_list *pdl){
    ////printf("Weeding pdl\n");
    if (pdl->kind == pdl_EMPTY){
        return pdl;
    }

    pdl->list = weed_vdl(pdl->list);
    return pdl;
}

var_decl_list *weed_vdl(var_decl_list *vdl){
    switch (vdl->kind){

        case (vdl_TYPE):
            vdl->vartype = weed_vtype(vdl->vartype);
            break;

        case (vdl_LIST):
            vdl->list = weed_vdl(vdl->list);
            break;

    }
    return vdl;

}

var_type *weed_vtype(var_type *vtype){
    ////printf("Weeding vtype\n");

    vtype->type = weed_type(vtype->type);
    return vtype;

}

decl_list *weed_dlist(decl_list *dlist){
    ////printf("Weeding dlist\n");
    if (dlist->kind == dl_EMPTY){
        //Nothing to do
        return dlist;
    }
    dlist->list = weed_dlist(dlist->list);
    dlist->decl = weed_decl(dlist->decl);
    return dlist;

}

declaration *weed_decl(declaration *decl){
    ////printf("Weeding decl\n");
    switch (decl->kind){

        case (decl_TYPE):
            decl->val.type.type = weed_type(decl->val.type.type);
            break;

        case (decl_FUNC):
            decl->val.function = weed_function(decl->val.function);
            break;

        case (decl_VAR):
            decl->val.list = weed_vdl(decl->val.list);
            break;

    }
    return decl;

}

statement_list *weed_slist(statement_list *slist){
    //printf("Weeding slist\n");

    if (slist == NULL){
        return NULL;
    }

    slist->statement = weed_stmt(slist->statement);
    if (slist->kind == sl_LIST){
        slist->list = weed_slist(slist->list);
    }

    if (slist->statement == NULL){
        return slist->list;
    }



    return slist;

}

statement *weed_stmt(statement *stmt){
    //printf("Weeding statement, kind: %d\n", stmt->kind);

    struct function *f;

    switch(stmt->kind){

        case (statement_RETURN):
            //printf("Statement return\n");
            //printf("Before weeding expression:\n");
            //prettyEXP(stmt->val.ret);
            //printf("\n\n");
            stmt->val.ret = weed_expression(stmt->val.ret);
            //printf("After weeding expression:\n");
            //prettyEXP(stmt->val.ret);
            //printf("\n\n");

            f = stack_read(function_stack);
            //printf("Function this return belongs to: %s", f->head->id);
            if (f == NULL){
                print_error("Return outside of function", 0, stmt->lineno);
            }
            stmt->function = f;
            stmt->contains_ret = 1;
            break;

        case (statement_WRITE):
            //printf("Statement write\n");
            stmt->val.wrt = weed_expression(stmt->val.wrt);
            break;

        case (statement_ALLOCATE):
            //printf("Statement allocate\n");
            stmt->val.allocate.variable = weed_variable(stmt->val.allocate.variable);
            break;

        case (statement_ALLOCATE_LENGTH):
            //printf("Statement allocate length\n");
            stmt->val.allocate.variable = weed_variable(stmt->val.allocate.variable);
            stmt->val.allocate.length = weed_expression(stmt->val.allocate.length);
            break;

        case (statement_ASSIGNMENT):
            //printf("Statement assignment\n");
            stmt->val.assignment.variable = weed_variable(stmt->val.assignment.variable);
            stmt->val.assignment.expression = weed_expression(stmt->val.assignment.expression);
            break;

        case (statement_IF):
            //printf("Statement if\n");
            stmt->val.ifthen.expression = weed_expression(stmt->val.ifthen.expression);
            stmt->val.ifthen.statement1 = weed_stmt(stmt->val.ifthen.statement1);

            // Check if expression is always true/false
            if (stmt->val.ifthen.expression->kind == exp_TERM){
                if (stmt->val.ifthen.expression->val.term->kind == term_FALSE){
                    return NULL;
                }
                stmt = stmt->val.ifthen.statement1;
            }
            break;

        case (statement_IF_ELSE):
            //printf("Statement if else\n");
            stmt->val.ifthen.expression = weed_expression(stmt->val.ifthen.expression);
            stmt->val.ifthen.statement1 = weed_stmt(stmt->val.ifthen.statement1);
            //printf("Weeding statement 2\n");
            stmt->val.ifthen.statement2 = weed_stmt(stmt->val.ifthen.statement2);
            //printf("Done weeding statement 2\n");

            if (stmt->val.ifthen.statement1->contains_ret && stmt->val.ifthen.statement2->
                 contains_ret){
                stmt->contains_ret = 1;
            }

            // Check if expression is always true/false
            if (stmt->val.ifthen.expression->kind == exp_TERM){
                if (stmt->val.ifthen.expression->val.term->kind == term_FALSE){
                    stmt = stmt->val.ifthen.statement2;
                }
                stmt = stmt->val.ifthen.statement1;
            }

            break;

        case (statement_WHILE):
            //printf("Statement while\n");
            stmt->val.loop.expression = weed_expression(stmt->val.loop.expression);
            stmt->val.loop.statement = weed_stmt(stmt->val.loop.statement);
            if (stmt->val.loop.statement == NULL){
                return NULL;
            }
            break;

        case (statement_LIST):
            //printf("Statement list\n");
            stmt->val.list = weed_slist(stmt->val.list);
            if (stmt->val.list == NULL){
                return NULL;
            }
            stmt->contains_ret = stmt->val.list->statement->contains_ret;
            break;
    }

    //printf("\n");
    //prettySTMT(stmt);
    //printf("\n\n");

    return stmt;

}

variable *weed_variable(variable *variable){
    //printf("Weeding variable, kind: %d\n", variable->kind);
    switch (variable->kind){

        case (var_ID):
            //printf("ID of variable: %s\n", variable->id);
            break;


        case (var_EXP):
            variable->val.exp.exp = weed_expression(variable->val.exp.exp);
            variable->val.exp.var = weed_variable(variable->val.exp.var);
            break;

        case (var_RECORD):
            variable->val.record.var = weed_variable(variable->val.exp.var);
            break;

        default:
            break;

    }
    return variable;


}

expression *weed_expression(expression *expression){
    //printf("Weeding expression, kind: %d\n", expression->kind);

    struct expression *left_exp;
    struct expression *right_exp;

    struct term *left_term;
    struct term *right_term;
    struct term *temp;

    temp = NULL;


    if (expression->kind == exp_TERM){
        //printf("Weeding single term of kind: %d, expression kind: %d\n", expression->val.term->kind, expression->kind);
        expression->val.term = weed_term(expression->val.term);
        //printf("New term kind: %d\n", expression->val.term->kind);
        return expression;
    }

    expression->val.ops.left = weed_expression(expression->val.ops.left);
    expression->val.ops.right = weed_expression(expression->val.ops.right);

    left_exp = expression->val.ops.left;
    right_exp = expression->val.ops.right;

    if ((left_exp->kind == exp_TERM) && (right_exp->kind == exp_TERM)){
        left_term = left_exp->val.term;
        right_term = right_exp->val.term;

        if ((left_term->kind == term_NUM) && (right_term->kind == term_NUM)){
            //We have an expression with two constants
            switch(expression->kind){

                case (exp_MULT):
                    temp = make_Term_num(left_term->val.num * right_term->val.num);
                    break;

                case (exp_DIV):
                    if(right_term->val.num == 0){
                        print_error("Division by 0 error", 1, right_term->lineno);
                    }
                    temp = make_Term_num(left_term->val.num / right_term->val.num);
                    break;

                case (exp_PLUS):
                    temp = make_Term_num(left_term->val.num + right_term->val.num);
                    break;

                case (exp_MIN):
                    temp = make_Term_num(left_term->val.num - right_term->val.num);
                    break;

                case (exp_EQ):
                    if (left_term->val.num == right_term->val.num){
                        temp = make_Term_boolean(1);
                    } else {
                        temp = make_Term_boolean(0);
                    }
                    break;

                case (exp_NEQ):
                    if (left_term->val.num != right_term->val.num){
                        temp = make_Term_boolean(1);
                    } else {
                        temp = make_Term_boolean(0);
                    }
                    break;

                case (exp_GT):
                    if (left_term->val.num > right_term->val.num){
                        temp = make_Term_boolean(1);
                    } else {
                        temp = make_Term_boolean(0);
                    }
                    break;

                case (exp_LT):
                    if (left_term->val.num < right_term->val.num){
                        temp = make_Term_boolean(1);
                    } else {
                        temp = make_Term_boolean(0);
                    }
                    break;

                case (exp_GEQ):
                    if (left_term->val.num >= right_term->val.num){
                        temp = make_Term_boolean(1);
                    } else {
                        temp = make_Term_boolean(0);
                    }
                    break;

                case (exp_LEQ):
                    if (left_term->val.num <= right_term->val.num){
                        temp = make_Term_boolean(1);
                    } else {
                        temp = make_Term_boolean(0);
                    }
                    break;

            }
        }

        //Check for boolean expression
        switch(expression->kind){

            case (exp_AND):
                if ((left_term->kind == term_FALSE) || (right_term->kind == term_FALSE)){
                    temp = make_Term_boolean(0);
                }

                if ((left_term->kind == term_TRUE) && (right_term->kind == term_TRUE)){
                    temp = make_Term_boolean(1);
                }

                break;

            case (exp_OR):
                if ((left_term->kind == term_TRUE) || (right_term->kind == term_TRUE)){
                    temp = make_Term_boolean(1);
                }

                if ((left_term->kind == term_FALSE) && (right_term->kind == term_FALSE)){
                    temp = make_Term_boolean(0);
                }
                break;

            default:
                break;
        }

        //TODO Optimize this please, to many comparisons I think, or maybe put advanced patterns into a function for itself?
        if (temp == NULL){

            if (expression->kind == exp_AND){

                //Advanced patterns

                if (left_exp->kind == exp_TERM){
                    if (left_exp->val.term->kind == term_TRUE){
                        return expression->val.ops.right;
                    }
                    if (left_exp->val.term->kind == term_FALSE){
                        temp = make_Term_boolean(1);
                    }
                }

                if (right_exp->kind == exp_TERM){
                    if (right_exp->val.term->kind == term_TRUE){
                        return expression->val.ops.left;
                    }
                    if (right_exp->val.term->kind == term_FALSE){
                        temp = make_Term_boolean(1);
                    }
                }

            }

            if (expression->kind == exp_OR){

                //Advanced patterns

                if (left_exp->kind == exp_TERM){
                    if (left_exp->val.term->kind == term_FALSE){
                        return expression->val.ops.right;
                    }
                    if (left_exp->val.term->kind == term_TRUE){
                        temp = make_Term_boolean(1);
                    }
                }

                if (right_exp->kind == exp_TERM){
                    if (right_exp->val.term->kind == term_FALSE){
                        return expression->val.ops.left;
                    }
                    if (right_exp->val.term->kind == term_TRUE){
                        temp = make_Term_boolean(1);
                    }
                }
            }
        }
    }

    if (temp != NULL){
        //We reduced something
        //printf("Reduced something\n");
        expression->kind = exp_TERM;
        expression->val.term = temp;
        //printf("Done with expression, new kind: %d\n", expression->val.term->kind);
    }

    return expression;
}

term *weed_term(term *term){
    //printf("Weeding term, kind: %d\n", term->kind);
    struct expression *e;

    switch(term->kind){
        case (term_VAR):
            term->val.variable = weed_variable(term->val.variable);
            break;

        case (term_LIST):
            term->val.list.list = weed_alist(term->val.list.list);
            break;

        case (term_PAR):
            term->val.expression = weed_expression(term->val.expression);
            break;

        case (term_NOT):
            term->val.term_not = weed_term(term->val.term_not);
            if (term->val.term_not->kind == term_TRUE){
                term->kind = term_FALSE;
                break;
            }

            if (term->val.term_not->kind == term_FALSE){
                term->kind = term_TRUE;
                break;
            }

            if (term->val.term_not->kind == term_NOT){
                term = term->val.term_not->val.term_not;
                break;
            }

            if (term->val.term_not->kind == term_PAR){
                e = term->val.term_not->val.expression;
                switch(e->kind){

                    case (exp_EQ):
                        e->kind = exp_NEQ;
                        term = term->val.term_not;
                        break;

                    case (exp_NEQ):
                        e->kind = exp_EQ;
                        term = term->val.term_not;
                        break;

                    case (exp_LT):
                        e->kind = exp_GEQ;
                        term = term->val.term_not;
                        break;

                    case (exp_GT):
                        e->kind = exp_LEQ;
                        term = term->val.term_not;
                        break;

                    case (exp_LEQ):
                        e->kind = exp_GT;
                        term = term->val.term_not;
                        break;

                    case (exp_GEQ):
                        e->kind = exp_LT;
                        term = term->val.term_not;
                        break;

                    default:
                        break;

                }
            }
            break;

        case (term_ABS):
            term->val.expression = weed_expression(term->val.expression);
            if ((term->val.expression->kind == exp_TERM) && (term->val.expression->val.term->kind == term_NUM)){
                term->kind = term_NUM;
                term->val.num = abs(term->val.expression->val.term->val.num); // Should probably use a temp value instead of such a long value
            }
            break;

        default:
            break;

    }

    return term;

}

act_list *weed_alist(act_list *alist){
    //printf("Weeding alist\n");
    if (alist->kind == al_EMPTY){
        return alist;
    }
    alist->list = weed_elist(alist->list);
    return alist;

}

exp_list *weed_elist(exp_list *elist){
    //printf("Weeding elist\n");

    elist->expression = weed_expression(elist->expression);

    if (elist->kind == el_LIST){
        elist->list = weed_elist(elist->list);
    }

    return elist;

}
```

### setup.h

```
#ifndef __setup_h
#define __setup_h
#include "tree.h"
#include "symbol.h"

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

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

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

void setup_type(type *type, symbol_table *table);

int setup_pdl(par_decl_list *pdl, symbol_table *table);

int setup_vdl(var_decl_list *vdl, symbol_table *table);

void setup_vtype(var_type *vtype, symbol_table*table);

void setup_dlist(decl_list *dlist, symbol_table*table);

void setup_decl(declaration *decl, symbol_table *table);

void setup_slist(statement_list *slist, symbol_table *table);

void setup_stmt(statement *stmt, symbol_table *table);

void setup_var(variable *var, symbol_table *table);

void setup_exp(expression *exp, symbol_table *table);

void setup_term(term *term, symbol_table *table);

void setup_alist(act_list *alist, symbol_table *table);

void setup_elist(exp_list *elist, symbol_table *table);


#endif
```

### setup.c

```
/**
 * @brief
 *
 * @file setup.c
 * @author Morten Jæger
 * @date 2018-03-09
 */

```

```
#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);
    st->val.func_type.pdl = head->list;

}

void setup_type(type *type, symbol_table*table){
    type->table = table;
    symbol_type *st;

    st = NEW(symbol_type);
    //printf("Setting up type\n");

    switch(type->kind){

        case (type_ID):
            st->type = symbol_ID;
            type->stype = st;
            break;

        case (type_INT):
            st->type = symbol_INT;
            type->stype = st;
            break;

        case (type_BOOl):
            st->type = symbol_BOOL;
            type->stype = st;
            break;

        case (type_ARRAY):
            st->type = symbol_ARRAY;
            type->stype = st;
            setup_type(type->val.type, table);
            break;

        case (type_RECORD):
            st->type = symbol_RECORD;
            type->stype = st;
            setup_vdl(type->val.list, scope_symbol_table(table));
            break;
    }


}

int setup_pdl(par_decl_list *pdl, symbol_table*table){
    //printf("Setting up pdl\n");
    pdl->table = table;
    int args;
    args = 0;
    if (pdl->kind != pdl_EMPTY){
        args = args + setup_vdl(pdl->list, table);
    }
    return args;
}

int setup_vdl(var_decl_list *vdl, symbol_table *table){

    //printf("Setting up vdl\n");
    vdl->table = table;
    int args;
    args = 1;
    setup_vtype(vdl->vartype, table);
    if (vdl->kind == vdl_LIST){
        args = args + setup_vdl(vdl->list, table);
    }
    return args;
}

void setup_vtype(var_type *vtype, symbol_table*table){

    //printf("Setting up var_type\n");
    vtype->table = table;
    symbol_type *st;
    st = NEW(symbol_type);
    st->type = symbol_UNKNOWN; // Sikkert ikke rigtigt

    SYMBOL *s;
    s = put_symbol(table, vtype->id, 0, st);
    vtype->symbol = s;

    setup_type(vtype->type, table);


}

void setup_dlist(decl_list *dlist, symbol_table *table){

    //printf("Setting up dlist\n");
    if (dlist->kind != dl_EMPTY){
        setup_decl(dlist->decl, table);
        setup_dlist(dlist->list, table);
    }

}

void setup_decl(declaration *decl, symbol_table *table){

    //printf("Setting up declaration\n");
    decl->table = table;

    symbol_type *st;
    switch (decl->kind){

        case (decl_TYPE):
            st = NEW(symbol_type);
            st->type = symbol_ID;
            put_symbol(table, decl->val.type.id, 0, st);
            setup_type(decl->val.type.type, table);
            st->val.id_type = decl->val.type.type;
            break;

        case (decl_FUNC):
            setup_function(decl->val.function, table);
            break;

        case (decl_VAR):
            setup_vdl(decl->val.list, table);
            break;

    }

}

void setup_slist(statement_list *slist, symbol_table *table){

    //printf("Setting up slist\n");
    slist->table = table;
    setup_stmt(slist->statement, table);
    if (slist->list != NULL){
        setup_slist(slist->list, table);
    }

}

void setup_stmt(statement *stmt, symbol_table*table){

    //printf("Setting up statement\n");
    stmt->table = table;
    switch(stmt->kind){

        case (statement_RETURN):

            //printf("\tStatement return\n");
            setup_exp(stmt->val.ret, table);
            break;

        case (statement_WRITE):
            //printf("\tStatement write\n");
            setup_exp(stmt->val.wrt, table);
            break;

        case (statement_ALLOCATE):
            //printf("\tStatement allocate\n");
            setup_var(stmt->val.allocate.variable, table);
            break;

        case (statement_ALLOCATE_LENGTH):
            //printf("\tStatement allocate length\n");
            setup_var(stmt->val.allocate.variable, table);
            setup_exp(stmt->val.allocate.length, table);
            break;

        case (statement_ASSIGNMENT):
            //printf("\tStatement assignment\n");
            setup_var(stmt->val.assignment.variable, table);
            setup_exp(stmt->val.assignment.expression, table);
            break;

        case (statement_IF):
            //printf("\tStatement if\n");
            setup_exp(stmt->val.ifthen.expression, table);
            setup_stmt(stmt->val.ifthen.statement1, table);
            break;

        case (statement_IF_ELSE):
            //printf("\tStatement if else\n");
            setup_exp(stmt->val.ifthen.expression, table);
            setup_stmt(stmt->val.ifthen.statement1, table);
            setup_stmt(stmt->val.ifthen.statement2, table);
            break;

        case (statement_WHILE):
            //printf("\tStatement while\n");
            setup_exp(stmt->val.loop.expression, table);
            setup_stmt(stmt->val.loop.statement, table);
            break;

        case (statement_LIST):
            //printf("\tStatement list\n");
            setup_slist(stmt->val.list, table);
            break;
    }

}

void setup_var(variable *var, symbol_table*table){

    //printf("Setting up variable\n");
    var->table = table;

    switch (var->kind){

        case (var_ID):
            //printf("ID: %s\n", var->id);
            break;

        case (var_EXP):
            setup_var(var->val.exp.var, table);
            setup_exp(var->val.exp.exp, table);
            break;

        case (var_RECORD):
            setup_var(var->val.record.var, table);
            break;

    }

}

void setup_exp(expression *exp, symbol_table *table){

    //printf("Setting up expression\n");
    exp->table = table;
    //printf("Expression kind: %d\n", exp->kind);
    if (exp->kind == exp_TERM){
        setup_term(exp->val.term, table);
    } else {
        setup_exp(exp->val.ops.left, table);
        setup_exp(exp->val.ops.right, table);
    }

}

void setup_term(term *term, symbol_table *table){

    //printf("Setting up term\n");
    term->table = table;
    switch(term->kind){

        case (term_VAR):
            setup_var(term->val.variable, table);
            break;

        case (term_LIST):
            setup_alist(term->val.list.list, table);
            break;

        case (term_PAR):
            setup_exp(term->val.expression, table);
            break;

        case (term_NOT):
            setup_term(term->val.term_not, table);
            break;

        case (term_ABS):
            setup_exp(term->val.expression, table);
            break;

        default:
            break;

    }

}

void setup_alist(act_list *alist, symbol_table *table){

    //printf("Setting up alist\n");
    alist->table = table;
    if (alist->kind == al_LIST){
        setup_elist(alist->list, table);
    }

}

void setup_elist(exp_list *elist, symbol_table*table){
    //printf("Setting up elist\n");
    elist->table = table;

    switch(elist->kind){

        case (el_EXP):
            setup_exp(elist->expression, table);
            break;

        case (el_LIST):
            setup_exp(elist->expression, table);
            setup_elist(elist->list, table);
            break;
    }
}
```

### pickup.h

```
#ifndef __pickup_h
#define __pickup_h
#include "tree.h"

void pickup_body(body *body);

void pickup_function(function *function);

void pickup_head(head *head);

void pickup_pdl(par_decl_list *pdl);

void pickup_vdl(var_decl_list *vdl);

void pickup_vtype(var_type *vtype);

void pickup_dlist(decl_list *dlist);

void pickup_declaration(declaration *decl);

void pickup_type(type *type);

type *resolve_recursive_type(type *type);

#endif
```

### pickup.c

```
#include <stdio.h>
#include "debug.h"
#include "pickup.h"
#include "tree.h"
#include "symbol.h"
#include "error.h"



void pickup_body(body *body){
#if debugflag > 2
    printf("Picking up body\n");
#endif
    pickup_dlist(body->d_list);

}

void pickup_function(function *function){
#if debugflag > 2
    printf("Picking up function\n");
#endif
    pickup_head(function->head);
    pickup_body(function->body);
    function->stype = function->head->stype;
}

void pickup_head(head *head){
#if debugflag > 2
    printf("Picking up head\n");
#endif
    pickup_pdl(head->list);
    pickup_type(head->type);
#if debugflag > 3
    printf("Picked up Type in Head, Head type: %d, Symbol type: %d\n", head->type->kind, head->
        stype->type);
#endif
    head->stype->val.func_type.ret_type = head->type;
#if debugflag > 2
    printf("Assigned ret type\n");
#endif
}

void pickup_pdl(par_decl_list *pdl){
#if debugflag > 2
    printf("Picking up pdl\n");
#endif
    if (pdl->kind == pdl_LIST){
        pickup_vdl(pdl->list);
    }

}
```

```

void pickup_vdl(var_decl_list *vdl){
#if debugflag > 2
    printf("Picking up vdl\n");
#endif
    pickup_vtype(vdl->vartype);
    if (vdl->kind == vdl_LIST){
        pickup_vdl(vdl->list);
    }

}

void pickup_vtype(var_type *vtype){
#if debugflag > 3
    printf("Picking up vtype, id: %s\n", vtype->id);
#endif
    pickup_type(vtype->type);
#if debugflag > 2
    printf("Picked up Type in VType\n");
#endif
    vtype->symbol->stype = vtype->type->stype;

}

void pickup_dlist(decl_list *dlist){
#if debugflag > 2
    printf("Picking up dlist\n");
#endif
    if (dlist->kind == dl_LIST){
        pickup_declaration(dlist->decl);
        pickup_dlist(dlist->list);
    }
}

void pickup_declaration(declaration *decl){
#if debugflag > 2
    printf("Picking up declaration\n");
#endif

    switch (decl->kind){

        case (decl_FUNC):
            pickup_function(decl->val.function);
            break;

        case (decl_TYPE):
            pickup_type(decl->val.type.type);
            break;

        case (decl_VAR):
            pickup_vdl(decl->val.list);
            break;
    }

}

void pickup_type(type *type){
#if debugflag > 2
    printf("Picking up type, kind: %d\n", type->kind);
#endif

    SYMBOL *s;
    switch (type->kind){

        case (type_ARRAY):
            pickup_type(type->val.type);
            type->stype->val.array_type = type->val.type;
            break;

        case (type_BOOl):
            break;

        case (type_INT):
            break;

        case (type_RECORD):
            pickup_vdl(type->val.list);
            type->stype->val.record_type = type->val.list;
            break;

        case (type_ID):
#if debugflag > 3
            printf("ID we are looking for: %s, in table: %p\n", type->val.id, type->table);
#endif
            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\n", s->stype->type);
                }
                print_error("Identifier error", 0, type->lineno);
            }
            struct type *temp;
            temp = resolve_recursive_type(s->stype->val.id_type);
#if debugflag > 3
            printf("After recursive check\n");
#endif
            type->stype = temp->stype;
#if debugflag > 3
            printf("After assignment\n");
#endif
    }
#if debugflag > 3
    printf("After switch\n");
#endif

}

type *resolve_recursive_type(type *type){
#if debugflag > 2
    printf("Resolving recursive conflict\n");
#endif
    struct type *temp;
    temp = type;
    if (type->recursive_type == 1){
        print_error("Recursive type definition", 0, type->lineno);
    }
    type->recursive_type = 1;
#if debugflag > 3
    printf("Type kind: %d\n", type->kind);
#endif
    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);
        }
        temp = resolve_recursive_type(s->stype->val.id_type);
    }
#if debugflag > 3
    printf("Checked recursively\n");
#endif
    type->recursive_type = 0;
    return temp;
}
```

### check.h

```
#ifndef __check_h
#define __check_h
#include "tree.h"
#include "symbol.h"


void check_body(body *body);

void check_function(function *function);

void check_dlist(decl_list *dlist);

void check_decl(declaration *decl);

void check_slist(statement_list *slist);

void check_stmt(statement *stmt);

void check_var(variable *var);

void check_exp(expression *exp);

void check_term(term *term);

void check_alist(act_list *alist);

void check_elist(exp_list *elist);

int check_function_args(par_decl_list *pdl, act_list *alist);

int compare_stype(symbol_type *stype1, symbol_type *stype2);

int compare_record(symbol_type *stype1, symbol_type *stype2);



#endif
```

### check.c

```
#include <stdio.h>
#include "debug.h"
#include "tree.h"
#include "check.h"
#include "memory.h"
```

```
#include "error.h"
#include "symbol.h"
#include "pickup.h"


void check_body(body *body){

    check_dlist(body->d_list);
    check_slist(body->s_list);

}

void check_function(function *function){

    check_body(function->body);

}

void check_dlist(decl_list *dlist){

    if (dlist->kind == dl_LIST){
        check_dlist(dlist->list);
        check_decl(dlist->decl);
    }

}

void check_decl(declaration *decl){

    if (decl->kind == decl_FUNC){
        check_function(decl->val.function);
    }

}

void check_slist(statement_list *slist){

    check_stmt(slist->statement);
    if (slist->kind == sl_LIST){
        check_slist(slist->list);
    }

}

void check_stmt(statement *stmt){
#if debugflag > 2
    printf("Checking statement, kind: %d\n", stmt->kind);
#endif


    switch(stmt->kind){

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

        case (statement_WRITE):
            check_exp(stmt->val.wrt);
            if ((stmt->val.wrt->stype->type != symbol_INT) && (stmt->val.wrt->stype->type !=
                symbol_BOOL) && (stmt->val.wrt->stype->type != symbol_ID)){
               print_error("Wrong write type", 0, stmt->lineno);
            }
            break;

        case (statement_ALLOCATE):
            check_var(stmt->val.allocate.variable);
#if debugflag > 3
            printf("Allocating type: %d\n", stmt->val.allocate.variable->stype->type);
#endif
            if ( (stmt->val.allocate.variable->stype->type != symbol_ARRAY) && (stmt->val.
                allocate.variable->stype->type != symbol_RECORD)){
                print_error("Wrong allocate type", 0, stmt->lineno);
            }
            break;

        case (statement_ALLOCATE_LENGTH):
            check_var(stmt->val.allocate.variable);
            check_exp(stmt->val.allocate.length);
            if ( (stmt->val.allocate.variable->stype->type != symbol_ARRAY) && (stmt->val.
                allocate.variable->stype->type != symbol_RECORD)){
                print_error("Wrong allocate type", 0, stmt->lineno);
            }
            if (stmt->val.allocate.length->stype->type != symbol_INT){
                print_error("Allocate length must be integer", 0, stmt->lineno);
            }
            break;

        case (statement_ASSIGNMENT):
            check_var(stmt->val.assignment.variable);
            check_exp(stmt->val.assignment.expression);
            if (stmt->val.assignment.expression->stype->type == symbol_NULL){
                if ( (stmt->val.assignment.variable->stype->type != symbol_ARRAY) && (stmt->val.
                    assignment.variable->stype->type != symbol_RECORD)){
                    print_error("Can only assign array and record to NULL", 0 , stmt->lineno);
                }
                break;
            }
            if (!compare_stype(stmt->val.assignment.variable->stype, stmt->val.assignment.
                expression->stype)){
                print_error("Incompatible type assignment", 0, stmt->lineno);
            }
            break;

        case (statement_IF):
            check_exp(stmt->val.ifthen.expression);
            check_stmt(stmt->val.ifthen.statement1);
            if (stmt->val.ifthen.expression->stype->type != symbol_BOOL){
                print_error("If condition is not a boolean", 0, stmt->lineno);
            }
            break;

        case (statement_IF_ELSE):
            check_exp(stmt->val.ifthen.expression);
            check_stmt(stmt->val.ifthen.statement1);
            check_stmt(stmt->val.ifthen.statement2);
            if (stmt->val.ifthen.expression->stype->type != symbol_BOOL){
                print_error("If condition is not a boolean", 0, stmt->lineno);
            }
            break;

        case (statement_WHILE):
```

```c
            check_exp(stmt->val.loop.expression);
            check_stmt(stmt->val.loop.statement);
            break;

        case (statement_LIST):
            check_slist(stmt->val.list);
            break;

    }

}

void check_var(variable *var){
#if debugflag > 2
    printf("Checking variable, kind: %d\n", var->kind);
#endif

    SYMBOL *s;
    switch (var->kind){

        case (var_ID):
            s = get_symbol(var->table, var ->id);
            if (s == NULL){
                print_error("Symbol not defined", 0, var->lineno);
            }
            var->stype = s->stype;
            break;

        case (var_EXP):
            check_var(var->val.exp.var);
            check_exp(var->val.exp.exp);
            if (var->val.exp.exp->stype->type != symbol_INT){
                print_error("Expression in [] not an integer", 0, var->lineno);
            }
            if (var->val.exp.var->stype->type != symbol_ARRAY){
                print_error("Variable is not an array", 0, var->lineno);
            }
            var->stype = var->val.exp.var->stype->val.array_type->stype;
            break;

        case (var_RECORD):
            check_var(var->val.record.var);
            if (var->val.record.var->stype->type != symbol_RECORD){
                print_error("Variable is not an record", 0, var->lineno);
            }
            s = get_symbol(var->val.record.var->stype->val.record_type->table, var->val.record.id);
            if (s == NULL){
                print_error("Record entry does not exist", 0, var->lineno);
            }
            var->stype = s->stype;
            break;
    }

}

void check_exp(expression *exp){
#if debugflag > 2
    printf("Checking expression, kind %d\n", exp->kind);
#endif
    symbol_type *st;

    switch(exp->kind){

        case (exp_PLUS):      // Subject to change, could be made to work with strings, f.x. "Hi"+2 could be: "Hi2"
        case (exp_MIN):
        case (exp_MULT):      // Subject to change, could be made to work with strings, f.x. "Hi"*2 could be: "HiHi"
        case (exp_DIV):
#if debugflag > 3
            printf("Checking left expression, Arithmetic\n");
#endif
            check_exp(exp->val.ops.left);
#if debugflag > 3
            printf("Checking right expression, Arithmetic\n");
#endif
            check_exp(exp->val.ops.right);
            if (exp->val.ops.left->stype->type == symbol_INT && exp->val.ops.right->stype->type == symbol_INT){

                st = NEW(symbol_type);
                st->type = symbol_INT;
                exp->stype= st;

            } else {
                print_error("Operators in arithmetic expression are not integers", 0, exp->lineno);
            }
            break;

        case (exp_EQ):
        case (exp_NEQ):
#if debugflag > 2
            printf("Checking left expression, EQ/NEQ\n");
#endif
            check_exp(exp->val.ops.left);
#if debugflag > 2
            printf("Checking right expression, EQ/NEQ\n");
#endif
            check_exp(exp->val.ops.right);

            // Should check stype->type->val.func_type.ret_type if comparing a function
            if ((exp->val.ops.left->stype->type == symbol_RECORD) && (exp->val.ops.right->stype->type == symbol_NULL)){
#if debugflag > 3
                printf("Left type is record, and right type is NULL\n");
#endif
                st = NEW(symbol_type);
                st->type = symbol_BOOL;
                exp->stype = st;
                break;
            }

            if ((exp->val.ops.left->stype->type == symbol_NULL) && (exp->val.ops.right->stype->type == symbol_RECORD)){
#if debugflag > 3
                printf("Left type is NULL, and right type is record\n");
#endif
                st = NEW(symbol_type);
                st->type = symbol_BOOL;
                exp->stype = st;
                break;
            }
#if debugflag > 3
```

```
            printf("Left sides type: %d, Right sides type: %d\n", exp->val.ops.left->stype->type, exp->val.ops.right->stype->type);
#endif
            if (exp->val.ops.left->stype->type == exp->val.ops.right->stype->type){
#if debugflag > 3
                printf("Checked if type is the same\n");
#endif
                st = NEW(symbol_type);
                st->type = symbol_BOOL;
                exp->stype = st;

            } else {
                print_error("Operators in EQ or NEQ not the same", 0, exp->lineno);
            }
            break;

        case (exp_GEQ):
        case (exp_LEQ):
        case (exp_LT):
        case (exp_GT):
#if debugflag > 2
            printf("Checking left expression, GEQ/LEQ/LT/GT\n");
#endif
            check_exp(exp->val.ops.left);
#if debugflag > 2
            printf("Checking right expression, GEQ/LEQ/LT/GT\n");
#endif
            check_exp(exp->val.ops.right);
            if (exp->val.ops.left->stype->type == symbol_INT && exp->val.ops.right->stype->type == symbol_INT){

                st = NEW(symbol_type);
                st->type = symbol_BOOL;
                exp->stype = st;

            } else {
                print_error("Operators used in GEQ, LEQ, LT or GT not integers", 0, exp->lineno);
            }
            break;

        case (exp_AND):
        case (exp_OR):
#if debugflag > 2
            printf("Checking left expression, AND/OR\n");
#endif
            check_exp(exp->val.ops.left);
#if debugflag > 2
            printf("Checking right expression, AND/OR\n");
#endif
            check_exp(exp->val.ops.right);

            if (exp->val.ops.left->stype->type == symbol_BOOL && exp->val.ops.right->stype->type == symbol_BOOL){

                st = NEW(symbol_type);
                st->type = symbol_BOOL;
                exp->stype = st;

            } else {
                print_error("Operators used in AND or OR not boolean", 0, exp->lineno);
            }
            break;

        case (exp_TERM):
            check_term(exp->val.term);
            exp->stype = exp->val.term->stype;
            break;

    }

}

void check_term(term *term){
#if debugflag > 2
    printf("Checking term, kind: %d\n", term->kind);
#endif

    SYMBOL *s;
    symbol_type *st;

    switch(term->kind){

        case (term_VAR):
            check_var(term->val.variable);
            term->stype = term->val.variable->stype;
#if debugflag > 3
            printf("Term type: %d\n", term->stype->type);
#endif
            break;

        case (term_LIST):
            check_alist(term->val.list.list);
            s = get_symbol(term->table, term->val.list.id);
            if (s == NULL || s->stype->type != symbol_FUNCTION){
                print_error("Reference to function that does not exists", 0, term->lineno);
            }
            check_function_args(s->stype->val.func_type.pdl, term->val.list.list);
            term->stype = s->stype->val.func_type.ret_type->stype;
            break;

        case (term_PAR):
            check_exp(term->val.expression);
            term->stype->type = term->val.expression->stype->type;
            break;

        case (term_NOT):
            check_term(term->val.term_not);
            if (term->val.term_not->stype->type != symbol_BOOL){
                print_error("Cannot negate non-boolean", 0, term->lineno);

            }
            term->stype->type = term->val.term_not->stype->type;
            break;

        case (term_ABS):
            check_exp(term->val.expression);
#if debugflag > 3
            printf("Type of expression: %d\n", term->val.expression->stype->type);
#endif
            if ((term->val.expression->stype->type != symbol_INT) && (term->val.expression->stype->type != symbol_ARRAY)){
                print_error("Absolute value must be used on integer or array", 0 , term->lineno);
            }
            st = NEW(symbol_type);
            st->type = symbol_INT;
            term->stype = st;
            break;

        case (term_NUM):
            st = NEW(symbol_type);
            st->type = symbol_INT;
            term->stype = st;
            break;

        case (term_TRUE):
        case (term_FALSE):
            st = NEW(symbol_type);
            st->type = symbol_BOOL;
            term->stype = st;
            break;

        case (term_NULL):
            st = NEW(symbol_type);
            st->type = symbol_NULL;
            term->stype = st;
            break;

    }

}

void check_alist(act_list *alist){

    if (alist->kind == al_LIST){
        check_elist(alist->list);
    }

}


void check_elist(exp_list *elist){
    check_exp(elist->expression);
    if (elist->kind == el_LIST){
        check_elist(elist->list);
    }


}

int check_function_args(par_decl_list *pdl, act_list *alist){

    struct exp_list *elist;
    struct var_decl_list *vdl;
    struct type *temp;
    struct symbol_type *st1;
    struct symbol_type *st2;

    if (pdl->kind == pdl_EMPTY){
#if debugflag > 3
        printf("PDL is empty\n");
#endif
        if (alist->kind == al_EMPTY){
            return 1;
        } else {
            print_error("Too many function arguments", 0, alist->lineno);
        }
    } else {
        if (alist->kind == al_EMPTY){
            print_error("Too few function arguments", 0, alist->lineno);
        }
    }

    elist = alist->list;
    vdl = pdl->list;

    while ((vdl != NULL) && (elist != NULL)){
        //In case of recursive definition
        temp = resolve_recursive_type(vdl->vartype->type);

        st1 = temp->stype;
        st2 = elist->expression->stype;
#if debugflag > 2
        printf("ST1s type: %d, ST2s type: %d\n", st1->type, st2->type);
#endif

        if (!compare_stype(st1, st2)){
            print_error("Function argument type mismatch", 0, alist->lineno);
        }
        vdl = vdl->list;
        elist = elist->list;
    }

    if ((vdl == NULL) && (elist == NULL)){
        return 1;
    }
    if (vdl == NULL){
        print_error("Too many function arguments", 0, alist->lineno);
    }
    if (elist == NULL){
        print_error("Too few function arguments", 0, alist->lineno);
    }

    return 0;

}

int compare_stype(symbol_type *stype1, symbol_type *stype2){

    if (stype1 == NULL || stype2 == NULL){
#if debugflag > 3
        printf("Both NULL, return 0\n");
#endif
        return 0;
    }

    if (stype1->type == stype2->type){
#if debugflag > 3
        printf("Both equal, return 1\n");
#endif
        return 1;
    }

    if (stype1->type == symbol_ARRAY && stype2->type == symbol_ARRAY){
        return compare_stype(stype1->val.array_type->stype, stype2->val.array_type->stype);
    }

    if (stype1->type == symbol_RECORD && stype2->type == symbol_RECORD){
        return compare_record(stype1, stype2);
    }
#if debugflag > 3
    printf("Not equal, return 0\n");
#endif
```

```

    return 0;

}

int compare_record(symbol_type *stype1, symbol_type *stype2){

    var_decl_list *vdl1;
    var_decl_list *vdl2;

    symbol_table *table1;
    symbol_table *table2;

    SYMBOL *s1;
    SYMBOL *s2;

    vdl1 = stype1->val.record_type;
    vdl2 = stype2->val.record_type;

    table1 = stype1->val.record_type->table;
    table2 = stype2->val.record_type->table;

    while ((vdl1 != NULL) && (vdl2 != NULL)){

        s1 = get_symbol(table1, vdl1->vartype->id);
        s2 = get_symbol(table2, vdl2->vartype->id);

        if (s1 == NULL || s2 == NULL){
            return 0;
        }

        if (!compare_stype(s1->stype, s2->stype)){
            return 0;
        }
        vdl1 = vdl1->list;
        vdl2 = vdl2->list;
    }

    if ((vdl1 == NULL) && (vdl2 == NULL)){
        return 1;
    }

    return 0;

}
```

### debug.h

```
/**
 * @brief
 *
 * @file debug.h
 * @author Morten Jæger, Mark Jervelund & Troels Blicher Petersen
 * @date 2018-03-16
 */
#ifndef COMPILER_DEBUG_H


//0 none,
// 1 basic debug levels, eg. are the modules called correctly.
// 2 advanced debug levels, eg. is the program transversing the functions as expected.
// 3 all the debug things. print all debug information within functions.

#define debugflag 0

#define COMPILER_DEBUG_H

#endif //COMPILER_DEBUG_H
```

### kind.h

```
#ifndef __compiler_kind_h
#define __compiler_kind_h

typedef enum { exp_PLUS,
             exp_MIN,
             exp_MULT,
             exp_DIV,
             exp_EQ,
             exp_NEQ,
             exp_GT,
             exp_LT,
             exp_GEQ,
             exp_LEQ,
             exp_AND,
             exp_OR,
             exp_TERM } EXP_kind;

typedef enum { term_VAR,
             term_LIST,
             term_PAR,
             term_NOT,
             term_ABS,
             term_NUM,
             term_TRUE,
             term_FALSE,
             term_NULL } TERM_kind;

typedef enum { type_ID,
             type_INT,
             type_BOOl,
             type_ARRAY,
             type_RECORD } TYPE_kind;

typedef enum { pdl_LIST,
             pdl_EMPTY } PDL_kind;

typedef enum { vdl_LIST,
             vdl_TYPE } VDL_kind;

typedef enum { dl_LIST,
             dl_EMPTY } DL_kind;

typedef enum { decl_TYPE,
             decl_FUNC,
             decl_VAR } DECL_kind;

typedef enum { sl_STATEMENT,
             sl_LIST } SL_kind;

typedef enum { statement_RETURN,
             statement_WRITE,
             statement_ALLOCATE,
             statement_ALLOCATE_LENGTH,
             statement_ASSIGNMENT,
             statement_IF,
             statement_IF_ELSE,
             statement_WHILE,
             statement_LIST } STATEMENT_kind;

typedef enum { var_ID,
             var_EXP,
             var_RECORD } Var_kind;

typedef enum { al_LIST,
             al_EMPTY } AL_kind;

typedef enum { el_EXP,
             el_LIST } EL_kind;

typedef enum { jmp, //unconditional jump
             je, //jump equal
             jne, //Jump not equal
             jg, //jump greater
             jge, //jump greater or equal
             jl, //Jump less
             jle, //Jump less or equal

             push,
             pop,

             orl, //or less ?

             add,
             sub,
             mul,
             mov,
             movl,
             setne,
             cmpl,
             andb,
             cmp
} ASM_kind;

#endif //COMPILER_KIND_H
```

### memory.h

```
#ifndef __memory_h
#define __memory_h

void *Malloc(unsigned n);

#define NEW(type) (type *)Malloc(sizeof(type))

#endif
```

### memory.c

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
```

```c

void *Malloc(unsigned n) {
    void *p;
    if (!(p = malloc(n))) {
        fprintf(stderr, "Malloc(%d) failed.\n", n);
        fflush(stderr);
        abort();
    }
    return p;
}
```

### error.h

```c
#ifndef __error_h
#define __error_h

void print_error(char *error, int code, int line);


#endif
```

### error.c

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

void print_error(char *error, int code, int line){
    fprintf(stderr, "%s at line %d\n", error, line);
    exit(code);
}
```

### stack.h

```c
#ifndef __stack_h
#define __stack_h

typedef struct stack_node{
    void *val;
    struct stack_node *next;
} stack_node;

typedef struct stack{
    struct stack_node *top;
} stack;

stack *init_stack();

void stack_push(stack *stack, void *val);

void *stack_pop(stack *stack);

void *stack_read(stack *stack);

#endif
```

### stack.c

```c
#include <stdio.h>
#include "stack.h"
#include "memory.h"

stack *init_stack(){

    stack *s;
    s = NEW(stack);
    s->top = NULL;
    return s;

}

void stack_push(stack *stack, void *val){

    stack_node *sn;
    sn = NEW(stack_node);
    sn->val = val;
    sn->next = stack->top;
    stack->top = sn;

}

void *stack_pop(stack *stack){

    stack_node *top;
    void *val;

    top = stack->top;
    val = top->val;
    stack->top = top->next;
    free(top);
    return val;

}

void *stack_read(stack *stack){

    if (stack->top == NULL){
        return NULL;
    }
    return stack->top->val;
}
```