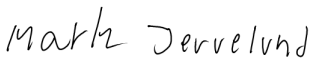



CC, Spring 2018
Bachelor project in Compiler Construction

Group	1
-------	---

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

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

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

This report contains a total of 60 pages.

Bachelor project in Compiler Construction
Mark Jervelund, Troels Blicher Petersen & Morten
Jæger
(Mjerv15, Trpet15, Mojae15)
Compiler (DM546)



UNIVERSITY OF
SOUTHERN DENMARK

IMADA

May 28, 2018

Contents

1	Introduction	1
2	The Shere Khan Compiler	1
2.1	Language limitations	1
2.2	Extensions	1
	Language Extensions	1
	Runtime Safety Improvements	1
	Advanced extensions	2
	Extra extensions	2
2.3	Implementation status	2
2.4	Building and running the Compiler	2
	Program flags	2
3	Design	3
3.1	Project	3
	Structure	3
3.2	Symbol table	4
	The symbol table	4
	Symbols	4
	Symbol types	4
3.3	Scanner and Parser	5
	Scanner	5
	Tokens	5
	Scanner Errors	5
	Parser	7
	Parser Errors	7
	Error Phases	7
3.4	Abstract Syntax Tree (AST)	10
3.5	Pretty Printer	10
3.6	Weeder	10
	Main purpose	10
	Dangers of weeding	11
3.7	Type checker	12
	Setup Phase	12
	Pickup Phase	12
	Check Phase	13
3.8	Code generation	13
	Structure of generated code	13
	Generating Assembly code	14
	Function calls	14
	Static Link	14
	Runtime Checks	15
3.9	Liveness Analysis and Register Allocation	16
	Liveness Analysis	16
	Register Allocation	17
3.10	Rewriter	19
3.11	Peephole	20
	Implemented Optimizations	20
3.12	Assembly printer	20
3.13	Auxiliary Structures	21
	Graph	21
	Bit Vector	21
	Table	22
	Stack	22
4	Example of a program through all phases	23

5	Implementation	33
5.1	The symbol table	33
5.2	Scanner	33
5.3	Parser	34
5.4	Abstract Syntax Tree (AST)	34
5.5	Pretty printer	35
5.6	Weeder	35
5.7	Type checking	36
	Setup phase	36
	Pickup phase	36
	Check phase	36
5.8	Code Generation	37
	General code generation	37
	Function Calls	38
	Static Linking	38
	Runtime Checks	39
5.9	Liveness Analysis	39
	Building the flowgraph	39
	Analysis	40
5.10	Register Allocation	40
5.11	Rewriter	41
	Replacing temporaries	41
	Function rewriting	41
	Adding push and pop of live registers	42
5.12	Peephole	42
	Removing move-to-self	42
5.13	Assembly printer	43
5.14	Auxiliary structures	43
	Graph	43
	Bit vector	43
	Table	44
	Stack	44
6	Testing	45
6.1	Our tests	45
6.2	Provided tests	57
7	Conclusion	59

1 Introduction

This report has been made to document the project "Bachelor project in Compile Construction". The main purpose of this project was to create a compiler for the Shere Khan language. The project was made by a group consisting of Mark Jervelund, Troels Blicher Petersen & Morten Jæger

The sections in this report follows the design, implementation and testing of the compiler. It is targeted at those who have already passed the course "DM546 - Compiler Construction".

The coding of the compiler was done mainly in the `c` programming language, with the scanner and parser as exceptions to this.

2 The Shere Khan Compiler

This section serves as an introduction to what the limitations of the Shere Khan we have implemented are, what extensions we have chosen to implement, and the general status of our implementation.

2.1 Language limitations

Besides the limitations of the grammar that defined the Shere Khan language, some other limitations have been introduced:

- The name of a function must be the same in both the head and in the tail.
- All function must have a return type, so all functions must contain at least one return statement.
- Return statements outside of a function is not allowed.
- A `write` call must take a integer or boolean argument.
- A `allocate` call must take a list or record argument.
- Comparing records or arrays is done by reference, not by content.
- In a `for` loop, the first statement must be an assignment. This also means that we do not allow a variable to be defined in the for loop.
- A boolean expression is processed completely, as long as we cannot determine the outcome during compilation (more on this in "Section 2.6: Weeder").
- Multiline comments must be closed, even at the end of the file.

2.2 Extensions

Several types of extensions were implemented as part of the project.

Language Extensions

Extensions to the language itself consist of:

- Unary minus.
- For loops.

Runtime Safety Improvements

Runtime checks have been implemented in the compiler. The return codes for these checks are:

- Array index value (Return code 2).
- Division by zero (Return code 3).
- Positive argument for array allocation (Return code 4).
- Uninitialized variable check (Return code 5).
- Out-of-memory check (Return code 6).

Advanced extensions

Several advanced extensions have been implemented as part of the project:

- Peephole optimization
- Advanced register allocation

Extra extensions

Some extra extensions have been implemented as part of the project:

- Constant folding
- Zero initialization of variables, arrays, and records.

2.3 Implementation status

The Shere Khan compiler works for the most part, however there are problems when dealing with large programs, such as "O_Knapsack.src". In this case, the generated program does not work properly. Also, programs with multi-dimensional arrays will also cause the generated program to not work properly.

For the most part however, the compiler works as intended.

2.4 Building and running the Compiler

Building the compiler is very straightforward and can be done using the provided `makefile`. Simply run:

```
1 $ make
```

in the project root directory.

To clean the project, run:

```
1 $ make clean
```

To clean the project and remove the compiled compiler, run:

```
1 $ make clean-all
```

It is not possible to run `make` with the `-j<num>` flag which compiles in parallel, since `bison` and `flex` are using separate compilers, and `GCC` and `Make` does not know how to handle dependencies from separate compilers.

Program flags

The compiler is very versatile in what kinds of information it can output while compiling and ways to compile. All of which can be toggled with command line flags, but are disabled by default to make it work with the python test script, provided in the examples directory. The flags will be thoroughly described below, but can also be found in the `man` by executing:

```
1 $ ./compiler -h
```

`-a` make an executable binary of the compiled Shere Khan program. This flag can only be used if the `-f` flag is being used as well, since it requires a file to be present.

`-o` Directory where the compiler should output the assembly files¹ and binaries if `-a` is used as well.

`-n` Will output assembly file for both before and after peephole optimization. Files will have the name `*_nopeep.s` and `*peep.s` for initial and peephole optimized versions, respectively.

¹There are assembly files for both before and after peephole optimization, if specified with `-n`

- e This will also execute the compiled binary.
- v Verbose, outputs statements when starting different modules of the compiler. Also outputs the result of the liveness analysis, the graph created in liveness analysis, the interference graph made in register allocation, process of assigning colors in register allocation and the spilled nodes, when the compiler takes care of those.
- p Pretty prints the program. This flag can have a value of 1 or 2 "-p 1" or "-p 2". Setting 1 will just pretty print. Setting 2 will also print types as well.
- m This is the memory flag. With this it is possible to define the amount of memory available to the compiled program. By default this value is 80 kB
- r Disable runtime checks. This can come in handy if the user of the compiler know, that there will be no runtime errors, by which this flag can increase performance.
- f Outputs the assembly code to a file instead of `stdout`.

Without any flags, the compiler will allow either a filename or raw `stdin` and simply output an assembly program.

3 Design

In the design phase focus will be on multiple perspectives of designing the compiler. This includes how to make it follow the spec of the Shere Khan Language itself, but also how the compiler from a broader perspective is designed. It is important to distinguish the two, since there are places where the Shere Khan language is not strictly defined, thus letting that up to the designers of the compiler, all of which is described in Section 1. How these defined rules are designed, will be described in this section. As mentioned however, there will also be focus on how the compiler follows general assumptions of how a good compiler should look and behave like. This includes readability, code-base modularity, extensibility and the general components a compiler, and how this compiler follows these rules.

3.1 Project

Structure

```

1 root/
2   ./src/
3     ./modules/
4       ./example_module/
5         ./include/
6   ./include/
7   ./out/
8   Makefile

```

From the very beginning modularity was thought into the design of the project. A simple and coherent way of adding new "modules" to the project has been laid out. By using a strict directory structure, it is possible to design a generic Makefile, so that adding new modules wont need any modifications to the Makefile. Is structure is that every module has its own include directory inside it, containing all headers for this module. General files can be put in the `src` folder and the header files for those in the `include` folder.

The modules are not necessarily independent of one another, but as a general rule of thumb are somewhat isolated. This means that little amount of work has to be done, to replace a module without having to rewrite the entire compiler. This is important, since this is a good way of dividing the individual components of the compiler into separate pieces.

The *Structure*-listing above shows how the project is laid out in general terms, with possibility to add an arbitrary number of modules, provided they follow the same module structure. Although not used, modules can also have subdirectories, if the respective include-folder also has the same subdirectories.

3.2 Symbol table

The Symbol table is where the symbols read from the input is stored. It functions like a hash table, where values that hash to the same value are chained together.

The symbol table

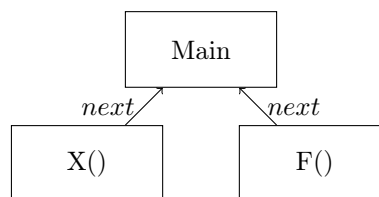
The symbol table itself, as mentioned before, functions like a hash table. The value that is hashed is the name of the variable that we want to insert into the table. This hash is then used to create a new "symbol" which is inserted into the table.

The symbol table also contains a pointer to its "next" symbol table. The "next" symbol table is used to separate different scopes in the program, so each function has its own symbol table, and a "next" pointer to the outer scope of the function. An example of this can be seen here:

```

1 | var a : int;
2 | :
3 | func F(): int
4 | :
5 | end F
6 | :
7 | func X(): int
8 | :
9 | end X
10 | :
11 | *do something*
```

The symbol tables for the program above would look like this:



The symbol tables for "F" and "X" now have a "next" pointer to the outer scope of the main function.

The use of this is to differentiate between the symbols used in the different functions. If, for example, the function "X" includes a symbol called "a", and we need to use that for something, we need to be able to tell the difference between this "a" and the "a" in the main function.

The way we differentiate between these two symbols, is when we need the symbol, we first search the current symbol table for the scope we are in for the symbol. If we don't find it there, we follow the "next" pointer, and search the symbol table it points to. This is done recursively until we find the symbol.

Symbols

A symbol in a symbol table corresponds to a variable in the program. When a variable is seen in the program, a new symbol with the name of the symbol is created and inserted into the appropriate symbol table. This makes it possible to search for the symbol in the symbol table when needed, to check if the program contains a reference to a variable that does not exist. If a symbol hashes to the same value as another symbol, the symbols are chained together. This is done by having each symbol containing a "next" pointer, which points to the next symbol in the chain.

Symbol types

Symbol types are used to differentiate between the different types a symbol can have in the program. For an example a symbol can be an integer, a record, a function and so on. This is comes in handy in

the type checking phase, to check if the use of the variables in the program are legal. More information about legal and illegal use of variables will be described in "Section 2.7: Type checker".

3.3 Scanner and Parser

The scanner and parser is where the Shere Khan language (and any language for that matter) is defined in the compiler. The Scanner is responsible for cleaning up and create the tokens for use in the Parser. The Parser is where the language is formally defined in the compiler and where the compiler gets and understanding of the language and what the given program has to do. In some cases this compiler actually does some weeding in this phase, however, this is the exception more than the norm, as overlapping of modules functionality is generally not preferable, when following the project guidelines. However, why this makes sense, will be described in (Section 2.3 Parser).

Scanner

The scanner is a fundamental part of the compiler. Here any text is translated into tokens, that can more easily be understood by the Parser. Since the Shere Khan language has already been defined² in Project Part 2, it serves as the foundation of this compiler. The tokens represent everything the language is supposed to know anything about. The tokens can be divided into three main groups.

The first group is the group of defined words and letter combinations that has been strictly defined in the language. This is where operators and language specific syntax is defined.

The second group is the group of identifiers. Since the language should not be a limiting factor when writing a program, it makes sense to allow an arbitrary combination of letters, as long as the identifier does not match any language defined "words". Furthermore, the group has defined, that identifiers can not begin with a number, but is allowed to contain numbers everywhere else. The reasoning behind this is, that it removes ambiguity for both the compiler and any programmer writing a program in the language. For an example an identifier beginning with a number could theoretically just be a number, which would make it hard to distinguish it from an actual number. This however, can be circumvented, by defining that an identifier must have at least one letter in it somewhere. The second reason behind it is therefore also, that it not only makes it easier for a programmer to read a Shere Khan program, it also forces programmers to follow a similar patterns when choosing an identifier, which makes hopefully makes the language more coherent and readable.

The third group is for numbers, integers to be more specific. The compiler only supports integers, which means this groups is very small. Here it is important to note, that it will only allow numbers, that does not begin with at least one zero. It does not make sense to add zeros in front of a number, and if a programmer might want to align variables and their values each other, it still is possible to do with spaces or tabs.

The language also supports things that should not get saved as a token for the Parser. This is defined to be everything in the comment sections. Generally speaking, most languages support some sort of way to add comments to a program. However, there can be many ways to do that. In the Shere Khan language there has to be two ways of commenting. The first is one-line comments and the second is multi-line comments. The multi-line comments also has to be able to nest inside each other, which is different from a language such as C, where nested comments are not allowed. Nested comments are ideal as a simple and fast tool for programmers, that want to "disable" parts of a program, without having to worry about other comments already existing in the code.

Tokens

In Table 1 is a compilation of all the tokens and what they derive from.

Scanner Errors

As part of the scanner, it makes sense to have some simple error checking at an early stage. The errors that can be reported this early, includes identifiers and numbers, where identifiers have to begin with a letter and numbers cannot begin with an arbitrary number of zeros. Furthermore, it is also possible to check if multi-line comments have been closed accordingly i.e. same amount of comment openers as

²to some extend. More on that later.

Table 1: Tokens

String	Token
==	EQ
!=	NEQ
<	LT
<=	LEQ
>	GT
>=	GEQ
if	IF
else	ELSE
while	WHILE
for	FOR
return	RETURN
&&	AND
true	TRUE
false	FALSE
null	_NULL
func	FUNC
end	END
int	INT
bool	BOOL
array of	ARRAY_OF
record of	RECORD_OF
type	TYPE
var	VAR
write	WRITE
allocate	ALLOCATE
of length	OF_LENGTH
then	THEN
do	DO
0 ([1-9][0-9]*)	tINTCONST
[a-zA-Z_][a-zA-Z0-9_]*	tIDENTIFIER

the amount of comment closers. The reason is, that if a program contains undefined text, one can be sure, that it won't compile and run, since the Parser won't know how to deal with it. Same goes with multi-line comments, where a program in most cases won't work, if a programmer accidentally forgot to close a comment, rendering the remaining of the program unusable for the parser. While in theory a programmer could leave a multi-line comment open in the bottom of the program, this could leave many open questions about the program for the compiler and even other programmers. A comment left open in the bottom, could mean that only parts of a program was copied - both a programmer and the scanner could get this "thought". Leaving a comment open also does not help with language coherency, thus again forcing the programmer to do some things, might make the language more attractive in the future.

Parser

The parser is where the syntax of the language is defined. It consists of two things; the parser itself and an abstract syntax tree, to give a structural representation of the input, while checking if it follows the syntax correctly. This means that all the tokens returned from the scanner now will be put into context, based on the order they are arranged.

The language is divided into a *head*, *body* and a *tail*, which is a pattern that will show up many times in a big program. The head is where information about a function is stored. This information includes an id of the head, so that it can be referenced from other places further down the AST. There is also a type associated with a function or a head. This can be any valid type in the language, but whatever type is chosen for a function, has to be the type that is returned from that particular function. This becomes important in the type checking phase later on in the compilation, but in the parser phase, it is simply only saved to make ready for the type checking phase. An important part of function definitions and usage of functions is the ability to require input parameters, so that a function can solve an arbitrary problem with a valid arbitrary input, so that it can be reused with different inputs. Therefore the head also saves a list of declared parameters for that particular function.

Now, a head is nothing without a body. Therefore the parser has to make sure, that any function has a corresponding body. The body is defined to contain *declarations* and *statements*. Declarations are everything that can be defined for use later on. This includes variables, records of variable and functions, which makes it possible to declare a function inside the scope of another function. A statement can be either expression, where a variable is assigned by a constant value or calculated somehow. However, the group of statements is not limited to that. Therefore if-statements, while loops and for loops are part of the statements and thus belong in the body as well. Since the body is so general in its usage, this has been defined, to be the start symbol of the language.

Lastly there is a tail part. The tail is only used when a function is declared, and is situated at the end of that particular function. The tail is an END token with a `tIDENTIFIER` behind. The `tIDENTIFIER` is the name of the function. The tail tells the compiler, that the function has ended. Technically the tail with the identifier is not required, as seen in many other languages. However, it has been defined in the language specification of the Shere Khan language, and therefore it is required in this compiler. While it might seem a little redundant to have a function identifier at the end of the function, it might also serve a practical purpose. Since functions can be declared inside each other, it can help the programmer, to easier see which function she is working on. It therefore might increase readability of the program.

Parser Errors

As with the scanner, some errors can also be reported in the parser phase. These errors include syntactical checks, but it still does not check for types yet. Type checking is performed in the Type checking phase later on. Instead it checks whether the input program is adhering to the language specification. If not, it has to report that and where the error is present in the given input program. Having it tell where the error is located, eases the programming labor needed by the programmer, since it tells where and what is wrong with the written code.

Error Phases

This is a somewhat abstract idea of how the error reporting works. As seen, both the Scanner and Parser will return errors in their respective phases and "areas of responsibility". Instead of having an entire error checking phase, all the errors checking is spread out to their respective phases. This is by

far the most efficient way to implement error checking. Not only does it decrease compile time, it also eases the updating and modularity of the compiler itself. The compile time is increased (compared to having an error phase), by being able to halt as soon as an error is noticed and report that error to the programmer. Having error checking in each phase also makes it follow the modularized approach to the compiler, since errors are part of the phase or module.

Table 2: Context-free grammar

<program>	:	<body>
<function>	:	<head><body><tail>
<head>	:	FUNC tIDENTIFIER (<par_decl_list>) : <type>
<tail>	:	END tIDENTIFIER
<type>	:	tIDENTIFIER INT BOOL ARRAY_OF <type> RECORD_OF { <var_decl_list> }
<par_decl_list>	:	<var_type>, <var_decl_list> <var_type>
<var_type>	:	tIDENTIFIER : <type>
<body>	:	<decl_list><statement_list>
<decl_list>	:	<declaration><decl_list> ε
<declaration>	:	TYPE tIDENTIFIER = <type>; <function> VAR <var_decl_list>;
<statement_list>	:	<statement> <statement><statement_list>
<statement>	:	RETURN <expression>; WRITE <expression>; ALLOCATE <variable>; ALLOCATE <variable> OF_LENGTH <expression>; <variable> = <expression>; IF <expression> THEN <statement> IF <expression> THEN <statement> ELSE <statement> WHILE <expression> DO <statement> FOR (<statement><expression>; <statement>) <statement> { <statement_list> }
<expression>	:	<expression> + <expression> <expression> - <expression> <expression> * <expression> <expression> / <expression> (<expression>) <expression> EQ <expression> <expression> NEQ <expression> <expression> GT <expression> <expression> LT <expression> <expression> GEQ <expression> <expression> LEQ <expression> <expression> AND <expression> <expression> <expression> - <expression> <term>
<term>	:	tINTCONST ! <term> <expression> TRUE FALSE _NULL <variable> tIDENTIFIER (<act_list>)
<act_list>	:	<exp_list> ε
<exp_list>	:	<expression> <expression>, <exp_list>

3.4 Abstract Syntax Tree (AST)

The Abstract Syntax Tree, or AST, is the main data structure of the program. It is a tree-like structure build up by the parser. When the parser matches on something, it creates a node in the AST, with the corresponding amount of children. An example of this could be a "+" expression. When finding such a expression, the parser would create a "+" node, with each side of the "+" as a child. This way, when we want go through the AST in f.x the type checking phase, we simply follow each nodes children to traverse the tree.

3.5 Pretty Printer

The Pretty Printer is a module of the compiler that prints the current AST. This is done by traversing the AST and printing corresponding text to standard output. Since the first printing of the program is done after the "weeding" phase, the output may not look precisely like the original program, but it should function the same when running the code.

An option of the pretty printer is to print with types. This will be done after the type checking phase, where we have found the types of the different variables and such. An example of what printing with types looks like can be seen here:

```
1 var a : int;
2 a = 42 : int;
3 write a : int;
```

As seen above, the type of the variable "a" is now printed when the variable is used.

3.6 Weeder

The weeder phase is a phase where the code is checked for expressions that can be evaluated now, instead of generating code for it later. However, the weeder's purpose is also to report an error, when illegal programs are found. The program the weeder checks for are programs not containing proper function returns, and programs containing a division by zero. The reason for implementing this in the weeder, and not in a later or separate phase, is simply because the weeder phase is the first time the AST is traversed and checked.

Main purpose

As mentioned, the weeder's main purpose is to find expressions, that can be removed from the program altogether. An example of what could be looked for in the weeder phase can be seen here:

```
1 var a : int;
2 a = 1+2+3;
3 write a;
```

When an expression like this is present in the AST, the weeder checks the expression for numbers and tries to evaluate the expression at compile time. The example above would be changed into the following:

```
1 var a : int;
2 a = 6;
3 write a;
```

This can only be done, because the values are numbers and not variables, since the value of a variable is not necessarily known at compile time.

The weeder is also responsible for trying to weed out boolean expressions. The idea is the same; try to evaluate the expression at compile time, instead of having to generate code for it. An example of this could be the following:

```
1 if (false) then
2     write 1;
3 else
4     write 2;
```

Since the boolean "false" is a constant and not a variable, its value would be known at compile time. The weeder would then weed out the part of the "if" statement, that would never be executed, resulting in the following:

```
1 write 2;
```

Another kind of boolean expression the weeder checks for are boolean expressions containing a "||" (or) or a "&&" (and). The weeder's job is again to check for constants, and apply that knowledge to possibly remove unused expressions. An example of this could be:

```
1 var a : bool;
2 a = true;
3 if (false && a) then
4     write 1;
5 else
6     write 2;
```

The weeder will recognize the constant value "false" and the expression as an "&&" expression, meaning that both sides of the symbol needs to evaluate to true. However, since the weeder already knows that one of the values is false, it can remove that part of the "if" statement altogether, resulting in the following:

```
1 write 2;
```

Dangers of weeding

Since the main purpose of the weeder is to remove expression that will never be executed, there can be some dangers involved. An example of this could be the following:

```
1 func F() : bool
2 :
3 end F
4
5 if (false && F()) then
6     write 1;
7 else
8     write 2;
```

As mentioned in the description of the weeder, the "&&" expression will be removed by the weeder, since it contains a "false" constant. However, the programmer could still want the function "F" to be executed, possibly because it causes changes to some global value. However, the weeder will remove the expression altogether, causing the function "F" to not be executed at all. A workaround for this could be the following:

```
1 func F() : bool
2 :
3 end F
4 if (F()) then
5     if (false) then
6         write 1;
7 else
8     write 2;
```

This would cause the function "F" to not be weeded out by the weeder, meaning that "F" would be executed.

3.7 Type checker

The type checker phase is where the compiler checks if a given input program is legal. This is done by figuring out the type of all variables and constants in the program, and checking if they are used properly. An example of what the compiler would be looking for in the type checker phase can be seen here:

```
1 var a : int;
2 var b : bool;
3 a = a+b;
4 write a;
```

In the program example above, an integer and a boolean is added together. This is one of the things the type checker would notice and deem illegal, since adding a integer and a boolean together is not allowed in the Shere Khan language.

The type checking phase is split up into three separate phases, the "*setup*" phase, the "*pickup*" phase, and the "*check*" phase.

Setup Phase

The "*setup*" phase is the first phase of the type checker. In this phase the compiler traverses through the AST of the program and sets up symbols and symbol tables for the program.

The symbols inserted into the symbol tables are f.x. the variables in the program. When finding a variable, the "symbol type" of the variable is set and inserted into the corresponding symbol table.

It should be noted that the "*setup*" phase only sets the type of "simple" types, meaning types that are not defined by the user. An example of this could be:

```
1 type t1 = int;
2 var a : t1;
3 var b : int;
```

In program above, the variable "b" has a simple, predefined type that is already known. The variable "a" however, has a user defined type. The type of "a" will not be resolved in the "*setup*" phase, since this phase only focuses on simple types. However, this is resolved in a later phase of the type checker.

To differentiate between variables with the same name in different scopes of the function, a new symbol table for each function is created, and inserted into that function's variables there. That symbol table will then have a "next" pointer to the symbol table in the outer scope, as described in (Section 2.2: Symbol table).

Since this phase is just setting up symbol tables and simple types of variables, there is the possibility of "merging" this pass-through of the AST with other phases, as to reduce compiling time a bit.

Pickup Phase

The "*pickup*" phase is the second phase of the type checker. In this phase the compiler traverses the AST again, but this time it will try to pick up missing types of variables. These types are things that did not get resolved in the "*setup*" phase, like variables with user defined types. An example of what the "*pickup*" phase would take care of can be seen here:

```
1 type t1 = int;
2 var a : t1;
3 var b : int;
```

In the program above, the variable "b" already has the type of an integer from the "*setup*" phase. The "*pickup*" phase would then "pick up" the type of "a" by looking at what "t1" was defined as.

The "*pickup*" phase also takes care of some illegal programs, by looking for user defined types, that are defined as each other. An example of this could be:

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


In the program above, there are two types, which are defined as each other, meaning that they don not actually have a proper type. This will be spotted in the "*pickup*" phase, and a error will be printed to the user.

Check Phase

The "*check*" phase is the third and last phase of the type checker. In this phase, once again, the compiler traverses the entire AST again, but this time it does the actual type checking, since it now knows the type of all variables and constants in the program. The purpose of the "*check*" phase is to find illegal programs and return an error message if one of these illegal programs are found. An example of this could be:

```
1 var a : int;
2 var b : bool;
3 a = 1;
4 b = false;
5 write a + b;
```

In the example program above, an integer and boolean is tried added. According by the language specifications of the Shere Khan language, this is not legal, so an error will be returned to the user, stating that the expression contains non-integer operators.

The most important part of the "*check*" phase (besides finding illegal programs) is to return clear and precise errors to the user, when an illegal program is discovered. Good error messages allow the user to easily fix their program and make it legal, which is what we want the user to do. Using the program above, the error message given from the "*check*" phase is the following:

```
1 Operators in arithmetic expression are not integers at line 5
```

The error message tells the user what kind of error that was discovered and where it was discovered. An extension of these kinds of error messages would be to also include the type and name of the variable or constant, that caused the error in the first place.

3.8 Code generation

The code generation phase is where the compiler starts outputting code based on the given input program.

Just like the type checker and weeder, the process of the code generation follows the AST, that has been generated in earlier stages of the compiler. This means that the compiler generates code for a node in the AST, and then generates code for that node's children, and so on. The generated code from this phase is not legal assembly code, since it contains temporaries for most of the instructions instead of actual registers. The reason behind this, is because it allows the compiler to optimize the register usage in the later phases, liveness analysis and register allocation.

Structure of generated code

The structure used for generating code has been made as easy a possible, to help with both reading and writing the generation of assembly code.

The structure for the generated code is basically a doubly linked list, where each node in the list is a structure that will be transformed into a single line of assembly code. This structure contains information about what kind of instruction the line should have, and what operators it should use.

When generating code, each function starts with an empty linked list. Then, after generating some code for the corresponding node in the AST's children, the linked lists of the children is inserted into the linked list of the function, creating a larger linked list. This list is then returned from the function. This process is done until there is only one big linked list left, which is the generated assembly program.

The linked list approach also makes it easy to insert smaller lists between nodes, since that would simply be changing what the lists point to as their previous and next nodes. This kind of injection is done f.x. in register allocation, when push and pops instructions are needed before "*call*" instructions.

Generating Assembly code

As mentioned before, to generate the assembly code, the compiler goes through AST. At each node it encounters, it generates some code for that node. Since it has been decided to use temporaries for most operations, we don't need to think about optimizing the use of registers in this phase, which makes it easier to generate the code. However, some place it has been decided to use registers instead of temporaries. One of these places is when we want to write a value. For this we use some specific registers to pass a string and the value that needs to be written, before "calling" the "printf" function. Another example of specific register usage is function calls. We decided to always use "RAX" as the register where the return value of a function would be located. This made it easier to get the return value from a function call, instead of having to choose a temporary number, saving that number, and then using that temporary to get the return value later.

To generate code for the different nodes in the AST, a "template" based approach was chosen, where a specific way of generating code for different parts of the AST has been defined. An example of such a template can be seen here. In this case a template for generating code for a "+" expression is shown: This gives the possibility to change the "template" if needed, f.x. if a faster way to do some opera-

"expression1" + "expression2"
Generate code for "expression1"
Remember temporary result from "expression1" is in
Generate code for "expression2"
Remember temporary result from "expression2" is in
Add temporary from "expression1" with temporary from "expression2"

tion is found. If such a thing is found, only the template for that specific part of the code has to be changed, giving us some "modularity" for generating code.

One important thing that needs to hold when generating code, is that results from operations must be in the second operator of the last instruction generated. The reason for this is when we need to get a result from some generated code, we always get the second operator from the "tail" of some generated code.

Using the example above, when "Remember temporary result from "expression1" is in", the compiler actually looks at the last element of the linked list for the function, where the generated code for "expression1" has just been inserted. Then the second operator of the last instruction of that generated code, is where the result from "expression1" is stored, whatever it might be. Then to "remember it", it looks at that last instructions second operator, and saves it for later usage. It is always assumed, that the result needed from some generated code is at this location, so should a "template" change, it still needs to uphold this. A way to ensure this always holds, is to insert a "dummy move" in the program. A "dummy move" is where some operator is moved from itself, to itself, such that the operator will be in the last instruction's second operator. This is ONLY done, so the operator can be retrieved, when generating the rest of the code, as the "dummy move" will be removed in the Peephole phase later on (See Section 3.11).

Function calls

To generate code for function calls, the compiler starts by checking the amount of parameters the function requires. A specific threshold is specified (which can be changed in the file "reg_alloc.h", namely the macro "PLACE_IN_REGS"), which determines if a function's parameters should be placed on the stack or in registers. Since there are 14 registers at our disposal, the threshold has been set to 4, as most functions do not have more than 4 parameters.

Another thing to note about functions in general, is that when they are present in some scope, the variables in that scope are put on the stack instead of storing them in registers. This is done, so the variables are reachable from a given function, should the function be called, by using the "static link".

Static Link

The "static link" is a "link" to the outer scope of some function. A way to see it is to look at the symbol table for a function, and then follow its "next" pointer. This pointer gives access to the outer scope of the function, and that scope's variables. The same applies for the "static link", which is used to give a function access to variables out of its scope. The way we have chosen to handle "static linking", is

by pushing a copy of the base pointer of a scope, just before calling a function. This address is then retrieved in the function, and used to get the variables of the outer scope if needed. An illustration of the stack has been made and can be seen here:

Base pointer
Variable1
Variable2

Table 3: Stack before pushing the static link, before a function call

Table 3 shows the stack before copying the base pointer to a new register, and pushing that onto the stack, giving us the static link needed for the function we want to call.

Base pointer
Variable1
Variable2
Static link

Table 4: Stack after pushing the static link, before a function call

Table 4 shows the stack after pushing the static link just before a function call.

Base pointer
Variable1
Variable2
Static link
Pushed rbp
Placement of the new rbp

Table 5: Stack after entering a new function, and saving the previous base pointer

Table 5 shows the stack after entering a new function. The first thing a new function does is, to save the old base pointer, by pushing it on the stack, and then setting the base pointer to point to the same thing as the stack pointer. Using the new base pointer, we use an offset of 16 to get the static link, and from there on we can access the variables on the stack.

Runtime Checks

Runtime checks are checks that are inserted when generating code. The purpose of these is to check different things, that cannot be checked at compile time. This includes checking if a given program tries to divide by 0, or if a given program tries to access an item in an array, at an index that does not exist. The value used in a given program may be a variable, meaning that it cannot be checked for during the compilation, which means that it has to check it when the program is run, thereof the name Runtime Checks.

The runtime checks should simply check if some condition is true, like if the the divisor is 0, and then jump to a label, corresponding to the type of error. If and when a runtime error has been spotted, the program should return the proper error code (these can be seen in (Section 1.2: Language Extensions)), and then close the program, since execution cannot continue with a faulty program. The labels will be added in the very bottom of the compiled code, and should only exist, for the checks that are present in the program. This means that if no divisions are being made, it should not include a division by 0 label in the code, since that would never be used anyways.

Since runtime checks add a lot of comparisons in the compiled program, they can potentially not only slow down the program a bit, but also increase the size of the compiled program. Therefore it makes sense to have a way of disabling these if needed. In some cases a programmer might be sure, that nothing can go wrong with the program. Therefore runtime checks can be disabled, using the `-r` flag when compiling.

3.9 Liveness Analysis and Register Allocation

As a part of the project, it has been chosen to implement advanced register allocation. This includes doing liveness analysis, so the compiler knows which registers and temporaries are live, and at what time they are live. After doing the liveness analysis, the register allocation can be done, and afterwards, rewrite the program.

Liveness Analysis

In the liveness analysis phase, the compiler wants to figure out which registers and temporaries are live at specific times in the program we have generated. The reason for doing so, is so that it can maximize the usage of the available register, when it does the register allocation.

The liveness analysis is split into two parts, building the flow graph for the program, and doing the analysis of the program.

Building the flow graph The flow graph is a graph that represents the "flow" of a given program. The nodes of the graph should represent the different instructions in the program, while the edges represent which instructions that could come after a given node. This means f.x., that nodes which contain a conditional "jump" instruction would have multiple edges, since there are multiple possibilities for what the next instruction would be, as the compiler does not know whether the "jump" is executed or not.

When building the flow graph the compiler also wants to set the "use"-sets and "def"-sets of each node. These sets define which temporaries or registers are used or/and defined in the given node. For example, if there is an instruction in the program that looks like the following:

```
1 addq t1, t2
```

-the compiler knows that the temporary "t1" is used, and that the temporary "t2" is both used and defined in this node. A table of what is added to the "use"-sets and "def"-sets in specific instructions can be seen here:

Instruction	1. Operator		2. Operator		Other operators	
	Used?	Defined?	Used?	Defined?	Used?	Defined?
MOVQ	✓	-	-	✓	-	-
IMUL, IDIV	✓	-	-	-	RAX	RAX
ADDQ, XORQ, SARQ, LEAQ	✓	-	✓	✓	-	-
SUBQ	✓	✓	✓	-	-	-
CMP	✓	-	✓	-	-	-
PUSH	✓	-	-	-	-	-
POP	-	✓	-	-	-	-
CDQ	-	-	-	-	RAX	RAX, RDX
RET	-	-	-	-	RAX	-
CALL	-	-	-	-	-	RAX
JMP, JG, JGE, JL, JLE, JNE, INT_, LABEL	-	-	-	-	-	-

Table 6: Table of what operators are defined and used in different instructions

In table 6, an overview what the different instructions use and define can be seen. This information is then used to add the corresponding temporaries or registers to a node's "use" and "define" sets. An exception to this table, is when a "printf" call is made, to print some output to the user. Since "printf" is not function we have defined, it may use other registers than those defined in the row of the "CALL". To handle this, all non-callee-save registers are set to be defined, when calling "printf". This ensures that during the liveness analysis and register allocation, the compiler is aware that "printf" may change the value of some registers.

Analysis After generating the flow graph and creating the initial "use" and "def" sets, the compiler has to do the actual analysis. This is done by first defining two new sets for each node, the "old" set and the "new" set. These sets will hold information about the temporaries and registers, that are live in a specific node. These sets are calculated for each node, again and again, until they don not change anymore, which is checked by comparing the "old" and the "new" set. This is because the "old" set is just a copy of the "new" set, copied before the "new" set was calculated in a given iteration. This way the compiler has the set both before and after it has been "updated", and it can therefore compare these two.

The way we calculate the "new" set for a given node " i ", is by using the following calculation:

$$new_i = use_i \cup \left(\bigcup_{j \in succ_i} Z_j - def_i \right) \quad (1)$$

$$Z_j = \begin{cases} new_j & \text{if } j > i \\ old_j & \text{otherwise} \end{cases}$$

The way the "new" set is calculated, is by looking at all the successors of some node in the flow graph. If the successor is a "higher number", meaning that the successor is an instruction later in the program (Think of it as the first line in the program has value 1, second line has value 2, and so on), the compiler uses that successor's "new" set for the calculation. Otherwise it uses the "old" set of that node.

This calculation is done backwards through the flow graph, meaning that it starts with the last instruction of the program, and works its way up. This is done since this follows the flow of the program, and is therefore more efficient.

As mentioned before, this kind of calculation of "new" sets is done for each node, until no change is found. This means that the compiler is done with the liveness analysis, and that it can now move on to the Register Allocation.

Register Allocation

The the register allocation phase, the compiler has to change the temporaries in the program to actual registers, so it becomes a legal assembly program. This is done by using the flow graph created in the liveness analysis phase, which tells the compiler, which temporaries and registers are used, and when they are used. Using this information, an interference graph is created to tell, which temporaries and registers interfere with each other. At last it should end up with a coloring of this graph, where each color corresponds to a register we can use in the assembly program.

The algorithms used for the register allocation is from the book by Andrew W. Appel [1]. The register allocation consists of multiple phase, which will be described here.

Main phase The first thing to do in the register allocation, is to build the interference graph. Building the interference graph is achieved, by going backwards through the program, and for each instruction, add an edge between all temporaries or registers, that resides in the "def"-set of that instruction, and all temporaries or registers that reside in the "new"-set of that instruction (Corresponds to all "live" temporaries and registers).

At this point a list called "**worklist_moves**" is being made in the program, which contains all moves in the program. This list will be used in a later phase.

After building the interference graph, different "worklists" are created, which will be used in the program. These worklists are the "**spill_worklist**", the "**freeze_worklist**" and the "**simplify_worklist**". The "**spill_worklist**" contains all temporaries that are "spilled". A temporary is spilled if its degree in the interference graph is larger than the amount of available registers, meaning that a register cannot to replace the temporary with. Handling spilled temporaries will be mentioned later.

The "**freeze_worklist**" contains all temporaries, that are not spilled, but are used in a "move" instruction. This temporary will now be considered "frozen", which means that it will no longer be considered during coalescing. What coalescing is will be mentioned later.

The "**simplify_worklist**" should contain all other temporaries, which are temporaries that are not used in "move" instructions, and does not have a high degree in the interference graph.

After the initial setup, the compiler checks the different worklists, and if they are not empty, it has to run the corresponding algorithm on the program. This means that if f.x. the "**simplify_worklist**"

is not empty, we "simplify", if the "freeze_worklist" is not empty, it will "freeze", and so on. This is done until all worklists are empty. When all worklists are empty, the colors has to be assigned to the temporaries in the program, and, if necessary, rewrite spills in the program. If there are spills that needs to be rewritten, the whole process of doing liveness analysis and register allocation has to be done all over again, since temporaries still has to be removed from the program.

Simplify To "simplify", the compiler removes a temporary from the "simplify_worklist", and put it onto a stack. The purpose of this stack, is to keep track of temporaries that can be colored later. It also makes sure to decrement the degree of other temporaries this temporary interfered with in the interference graph, since it has removed the temporary and it no longer interferes with other temporaries.

Coalescing Coalescing is the process of looking a "move" instructions, checking if there is an interference between the source and destination temporary, and, if no interference exists, coalescing the two temporaries together into one temporary.

The coalescing phase involves going through the worklist, "worklist_moves", which contains all moves in the program. When trying to coalesce two temporaries, the compiler first checks if the two temporaries are the same. If this is the case, it can easily coalesce the move, and by doing so, mark it as coalesced, if it checks it in the future.

However, if the "move" instruction contains two "precolored" register (non-temporary register, f.x "RAX" or "RBX"), or there is an interference between the two nodes we are trying to coalesce, this move is marked as "constrained".

If the node the compiler is trying to move is a "precolored" node, it does a check corresponding to the coalescing strategy by George, as described on page 239 in [1]. The check consists of checking all adjacent temporaries or registers of the particular node the compiler moves the "precolored" node to, and check if these have a low degree or conflict with the destination node. If this holds, the compiler allows the node to be coalesced. However, if neither of the the nodes are "precolored", it performs a check corresponding to the coalescing strategy by Briggs, as described on page 239 in [1]. Here it check if there are less temporaries with a significant degree (a degree that is higher than or equal to the amount of available registers), than the amount of available registers in the union of the two nodes adjacency sets. If so, we allow the nodes to be coalesced.

Freezing To "Freeze", the compiler removes a temporary from the "freeze_worklist". It then removes all "move" instructions, where this temporary is involved from consideration. This means that the "move" instruction will no longer be considered when doing coalescing. If doing so results in the node not being in any more "move" instructions, and the degree for that node is less than the amount of available registers, the temporary can be added to the "simplify_worklist".

Spilling In the "spill" phase, it goes through the "spill_worklist". From that worklist, a node is selected to remove from the worklist using a heuristic. The heuristic is calculated, by taking the amount of uses of a node, divided by the degree of the node. Using this heuristic, the node with the smallest value from the calculation is removed, hopefully giving a node that is used few times, and interferes with a lot of other nodes. This node should then be removed from the "spill_worklist" and placed in the "simplify_worklist" instead, indicating that it is now ready to be removed colored.

Coloring When all worklists are empty, the actual coloring is started. This is done by taking each temporary on the stack, that has been created in the "simplify" phase, and assigning a color to the temporary. If at some point a temporary is unable to be colored, this temporary is marked as "spilled". This means that the program needs to be rewritten with the now spilled temporary, and repeat the whole register allocation process.

If no temporaries were spilled during the coloring, the compiler can move on to rewriting the temporaries in the program with their now corresponding register.

Rewriting spills If a temporary is spilled during the coloring process, it needs to be placed on the stack, instead of being placed in a registers. When the temporary needs to be used, it will be fetched it from the stack, placed in some register, which will have a tiny lifespan, do some action with the temporary, and then store the temporary on the stack again. A table of what needs to be store and fetched for each instruction can be seen here:

Instruction	1. Operator		2. Operator	
	Fetch before use?	Store after use?	Fetch before use?	Store after use?
MOVQ	✓	-	-	✓
ADDQ, SUBQ, XORQ, SARQ, LEAQ	✓	-	✓	✓
CMP	✓	-	✓	-
PUSH	✓	-	-	-
POP	-	✓	-	-
CDQ, CALL, RET JMP, JG, JGE, JL, JLE, JNE, INT_, LABEL	-	-	-	-

Table 7: Table of what operators are fetched and stored in different instructions

In table 7, an overview of what is fetched and stored in the different instructions can be seen. This information is used when rewriting the program.

The actual rewriting of the program consists of adding "move" instructions to the program, where the compiler, based on the information in table 7, either moves a value from the stack to a register, or moves the value from a register to a stack location. This, of course, increases the amount of temporaries in the program, but the temporaries have a very short lifespan, so they wont interfere with too many other temporaries.

When the register allocation process is done, the entire program has to be rewritten, so that it can be executed properly.

3.10 Rewriter

In the rewriter phase the compiler goes through the program, and does rewrites where necessary. This includes replacing temporaries with the register they were assigned during the register allocation phase, but also adding "push" and "pop" instructions before and after function calls.

Replacing temporaries During the rewriter phase all temporaries has to be rewritten and replaced with either registers or stack locations. This is simply done by replacing each temporary with the register that corresponds to the color they were assigned in the register allocation phase. If a temporary f.x. has the color "0", it would be replaced with the register "RAX", and so on. However, if the temporary is spilled, a stack location for it can be created, since it does not have an assigned register.

Function rewriting The rewriter is also responsible for adding the post- and prefix of functions. The postfix of a function is where the basepointer is stored, before doing anything in the function. The prefix is where the basepointer is restored again. The rewriter should also allocate stack space for variables in the function, should be function have any variables stored on the stack.

Adding push and pop of live registers Another function of the rewriter is adding "push" and "pop" instruction of register before and after function calls. This is done by checking which registers are live before calling a function, and checking if the function uses any of those registers. If it is found, that a register is used by a function, but should not be overwritten, this register is "pushed" to the stack before calling the function. This way the register can be "popped" after the function call, restoring the value of the register, and preventing the value from being overwritten.

3.11 Peephole

In the "peephole" phase, the compiler goes through the program and makes small optimizations to the program. The purpose of these optimizations is to remove or change parts of the code, such that the program is faster/uses less memory/whatever, depending on what one may want to optimize towards.

Implemented Optimizations

As part of this project we have implemented single small optimization, but other optimizations are of course possible. The optimization we have chosen to implement was removing "move" instructions where the source and destination where the same.

Removing move-to-self The purpose of removing "move" instructions where the source and destination are the same, is because these instructions are basically pointless. For the most part, these exists solely for the purpose of getting a specific register when we generate the code, and as such are classified as "dummy moves". Since they serve no real purpose, we have decided to remove these instructions from the program.

Since the structure of the generated code is a linked list, it is fairly simple to remove these instructions. To do so, when we one of these nodes is spotted, the compiler makes the previous nodes next go to the next of the current node and vice versa, essentially "skipping" the node with the "move" instruction.

Other possible optimizations Other optimizations we thought about was f.x. checking for nodes where we divide by 2. This can be replaced with a bitwise "shift-right" operation, which should be faster. This would also remove the need for using both "RAX" and "RBX" when dividing.

Another possible, and relatively simple optimization, is to check for "move" instructions, where the value "0" is moved to a register. This can be replaced with an "XOR" operation, where both operators are the desired register. This can be done since an "XOR" operation on the same value will always give "0".

3.12 Assembly printer

As the last step of the compiler, the generated code has to be outputted to the user, so it can be assembled and ideally executed.

To print a given program, the printer first inserts strings that are used when calling the "printf" function. The printer also inserts the code needed for allocating the memory that is used for records and arrays. After the initial prints, the printer goes through the program and prints the appropriate code for each instruction. This includes formatting the different operator types to legal assembly code. Given an example program:

```
1 var a : int;
2 a = 42;
3 write a;
```

The assembly printer will output the following code:

```
1 .wrt_INT:                # Integer write label
2   .string "%d\n"         # String used for printing integers
3 .wrt_TRUE:               # TRUE write label
4   .string "TRUE\n"       # String used to print TRUE
5 .wrt_FALSE:             # FALSE write label
6   .string "FALSE\n"      # String used for printing FALSE
7   .comm MEM, 80000       # Available memory
8 .globl main
9 main:                   # Start of body
10  push %rbp              # Push old base pointer to stack
11  movq %rsp, %rbp        # Move stack pointer to base pointer
12  subq $8, %rsp          # Make space for variables and spills
13  movq $1, MEM           # Init MEM
14  movq $0, %rsi          # Init to zero
15  movq $42, %rsi         # Moving constant to register
```



```

16  push %rax           # Register is live in function, so saving it before CALL
17  movq $.wrt_INT, %rdi # First argument for printf
18  movq $0, %rax       # No vector arguments
19  call printf         # Calling printf
20  pop %rax            # Register was live in function, so restoring it after CALL
21  main_end:           # End of body
22  addq $8, %rsp        # Remove space for variables and spills
23  movq %rbp, %rsp     # Restore old stack pointer
24  pop %rbp            # Restore old base pointer
25  movq $0, %rax       # Return "no error" exit code
26  ret                 # Program return

```

If the given program is run, it prints the following output: **42**, which is correct. An example of a bigger program will be shown in "Section 4: Example of a program through all phases".

3.13 Auxiliary Structures

Several other data structures are used when compiling. They are described here, since they did not quite fit anywhere else. Since these structure are not that relevant, a brief explanation will be given about their general purpose.

Graph

The graph structure is based on the implementation of a graph from the book and website by Andrew W. Appel [1] [2].

Instead of the classic graph approach with a set of vertices and a set of edges, this graph only contains a list of vertices, or nodes. Each of these nodes then function as a kind of "linked list", where each node has a list of successors and predecessors. These successors and predecessors then represent the edges between the corresponding nodes in the graph. This way, when an edge needs to be added between two nodes, we simply add the first node to the list of predecessors of the second node, and add the second node to the list of successors of the first node.

Bit Vector

The bit vector structure is based on the implementation by Shun Yan Cheung [3].

A bit vector (or bit array) is a way of using integers to represent arrays of bits, instead of having a array of integers with the values 1 or 0. Since an integer is 32 bits in C, we can store 32 indices of an ordinary array in just one integer. This means that we get a much smaller array of integers, since each integer hold 32 values. This also means that when we need to loop through the bit array, there are a lot less indices that need to be checked.

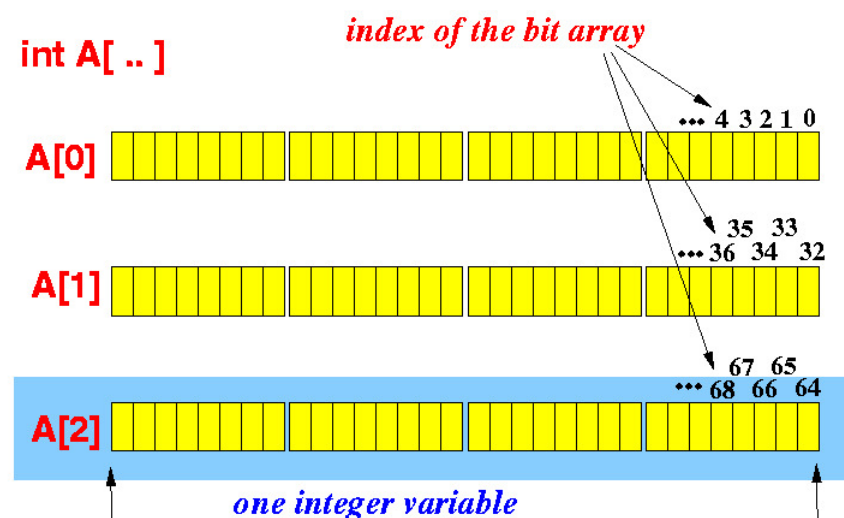


Figure 1: An example of a bit array[3]

In figure 1 an example of a bit array can be seen.

The way we set a bit in the bit array is by starting with the value "1". Bitwise, it consists of 31 zero's, and 1 one. This one is then shifted the desired amount of spaces, corresponding to the bit we want to set. We then find the index in the array where the desired bit is located, f.x. if we want to the the 42nd bit to be set, we use the index "1" of the array. We then "or" the integer at this index with the bit string we just created. This sets the bit at the desired position, without changing anything else. Bit vectors are used as the "sets" in the liveness analysis and register allocation phases.

Table

The table structure is based on the implementation of a table by Andrew W. Appel [2].

The table is basically a generic hash table, however, the value that is hashed in the table can be anything. What is meant by this, is that when we want to insert something into the hash table, we cast the object we want to insert to a unsigned integer and use this to calculate its placement in the hash table. This gives us the possibility to insert any object into the hash table.

Stack

The stack structure is a simple stack structure, where a pointer to some object can be pushed onto the stack. The pointer at the top of the stack can then either be read or popped from the stack completely. We chose to create the stack structure by using "void pointers", meaning that any pointer can be pushed onto the stack, instead of just one specific structure.

4 Example of a program through all phases

This section will show the output from the compiler through the different phases it goes through, when compiling a program. All of these outputs are created using the flags defined in "Section 2.4: Building and running the compiler".

The program chosen for this example is the following program:

```

1 # Code generation, realistic example of recursive code
2 func factorial(n: int): int
3   if (n == 0) || (n == 1) then
4     return 1;
5   else
6     return n * factorial(n-1);
7 end factorial
8
9 var a : array of int;
10 var i : int, n : int;
11
12 allocate a of length 10;
13 for (i = 0; i < 10; i = i +1;){
14   a[i] = factorial(i);
15 }
16
17 for(n = 0; n < 10; n = n +1;){
18   write a[n];
19 }

```

This program was created to show how the compiler handles different interesting things, such a recursive functions, lists, and one of our extensions, the "for" loop.

The first output the compiler can generate is from the "pretty-printer", which is shown here:

```

1 func factorial(n : int) : int
2   if ((n == 0) || (n == 1)) then
3     return 1;   else
4     return (n*factorial((n-1)));
5 end factorial
6 var a : array of int;
7 var i : int, n : int;
8 allocate a of length 10;
9 for (i = 0;(i < 10) ; i = (i+1);)
10 {
11   a[i] = factorial(i);
12 }
13 for (n = 0;(n < 10) ; n = (n+1);)
14 {
15   write a[n];
16 }

```

Since no error was detected before the output, we can go ahead to the next output, which is pretty-printing, but with types:

```

1 func factorial(n : int) : int
2   if ((n : int == 0 : int) : boolean || (n : int == 1 : int) : boolean) : boolean then
3     return 1 : int;   else
4     return (n : int*factorial((n : int-1 : int) : int) : int) : int;
5 end factorial : function(n : int) : int
6 var a : array of int;
7 var i : int, n : int;
8 allocate a of length 10 : int;
9 for (i = 0 : int;(i : int < 10 : int) : boolean ; i = (i : int+1 : int) : int;)
10 {
11   a[i : int] = factorial(i : int) : int;
12 }

```

```

13 for (n = 0 : int;(n : int < 10 : int) : boolean ; n = (n : int+1 : int) : int;)
14 {
15     write a[n : int] : int;
16 }

```

After this printing, we start generating assembly code for the program:

```

1  .wrt_INT:                # Integer write label
2  .string "%d\n"           # String used for printing integers
3  .wrt_TRUE:               # TRUE write label
4  .string "TRUE\n"        # String used to print TRUE
5  .wrt_FALSE:             # FALSE write label
6  .string "FALSE\n"       # String used for printing FALSE
7  .comm MEM, 80000        # Available memory
8  .globl main
9  factorial:              # Start of body
10     movq %r15, t14       # Move parameter to register in function
11     movq t14, t14        # Var is in this register
12     movq t14, t15        # Copy val to new temp, to not harm it
13     movq $0, t16         # Moving constant to register
14     cmp t16, t15         # Compare, EQ
15     je cmpTrue_1        # If true, jump to label
16     movq $0, t17         # Setting result to 0 (false)
17     jmp endCMP_1        # Jump to after compare label
18 cmpTrue_1:              # Compare true label
19     movq $1, t17         # Setting result to 1 (true)
20 endCMP_1:                # After compare label
21     movq t17, t17        # Used to get "target" when creating next instruction
22     movq $1, t18         # Setting default value of result to true
23     cmp $1, t17          # Compare left side of OR with true
24     je endBoolCMP_2     # If true, skip right expression
25     movq t14, t14        # Var is in this register
26     movq t14, t19        # Copy val to new temp, to not harm it
27     movq $1, t20         # Moving constant to register
28     cmp t20, t19         # Compare, EQ
29     je cmpTrue_3        # If true, jump to label
30     movq $0, t21         # Setting result to 0 (false)
31     jmp endCMP_3        # Jump to after compare label
32 cmpTrue_3:              # Compare true label
33     movq $1, t21         # Setting result to 1 (true)
34 endCMP_3:                # After compare label
35     movq t21, t21        # Used to get "target" when creating next instruction
36     movq t21, t18        # Result is set to the value of the right expression
37 endBoolCMP_2:           # OR expression label
38     movq t18, t18        # Used to get target for next instruction
39     cmp $1, t18          # Check if IF expression is true
40     jne else_1           # Expression is false, jump to ELSE part
41     movq $1, t22         # Moving constant to register
42     movq t22, %rax       # Return value placed in RAX
43     jmp end_factorial    # Jump to functions end label
44     jmp if_end_1        # Skip ELSE part
45 else_1:                 # Start of ELSE
46     movq t14, t14        # Var is in this register
47     movq t14, t23        # Copy val to new temp, to not harm it
48     movq t14, t14        # Var is in this register
49     movq t14, t24        # Copy val to new temp, to not harm it
50     movq $1, t25         # Moving constant to register
51     subq t25, t24        # Subtraction
52     movq t24, %r15       # Moving function parameter to register
53     movq 16(%rbp), t26   # Retrieving static link
54     push t26             # Storing static link for function
55     call factorial       # Calling function
56     addq $8, %rsp        # Remove static link

```

```

57     movq %rax, t27      # Saving return value from function in temp
58     movq t23, %rax      # Using RAX for multiplication
59     movq t27, %rdx      # Using RDX for multiplication
60     imul %rdx           # Multiplication using RAX and RDX
61     movq %rax, t28      # Storing result here (temp)
62     movq t28, %rax      # Return value placed in RAX
63     jmp end_factorial   # Jump to functions end label
64 if_end_1:              # End of IF
65 end_factorial:         # End of body
66     ret                 # Return from function
67
68 main:                  # Start of body
69     movq $1, MEM        # Init MEM
70     movq $0, 0(%rbp)     # Init to zero
71     movq $0, -8(%rbp)    # Init to zero
72     movq $0, -16(%rbp)   # Init to zero
73     movq $10, t29        # Moving constant to register
74     cmp $0, t29          # Check if allocated length is positive
75     jle positive_allocate # Jump to error label
76     movq MEM, t31        # Move pointer to MEM to another register
77     movq t29, t30        # Moving length of array to new reg
78     addq $1, t30         # Add 1 to array length
79     movq t30, MEM(,t31,8) # Move allocated length to mem location
80     movq 0(%rbp), 0(%rbp) # Var is on stack
81     movq t31, 0(%rbp)    # Move mem location to variable
82     addq t31, t30        # Add length of array to mem register
83     cmp $80000, t29      # Check if out of memory
84     jge out_of_mem      # Jump to error label
85     movq t30, MEM        # Update mem pointer
86     movq $0, t32         # Moving index 0 to register
87     addq 0(%rbp), t32    # Adding index to start of array
88     movq t29, MEM(,t32,8) # Assigning length of array to index 0
89     movq 0(%rbp), 0(%rbp) # Used to get target for next instruction
90     movq $0, t33         # Moving constant to register
91     movq -8(%rbp), -8(%rbp) # Var is on stack
92     movq t33, -8(%rbp)   # Assigning value to var
93 loop_start_1:          # Start of for
94     movq -8(%rbp), -8(%rbp) # Var is on stack
95     movq -8(%rbp), t34    # Copy val to new temp, to not harm it
96     movq $10, t35        # Moving constant to register
97     cmp t35, t34         # Compare, LT
98     jl cmpTrue_4        # If true, jump to label
99     movq $0, t36         # Setting result to 0 (false)
100    jmp endCMP_4         # Jump to after compare label
101 cmpTrue_4:             # Compare true label
102    movq $1, t36         # Setting result to 1 (true)
103 endCMP_4:              # After compare label
104    movq t36, t36        # Used to get "target" when creating next instruction
105    cmp $1, t36          # Check if condition in while is true
106    jne loop_end_1       # If condition is false, jump to end
107    movq -8(%rbp), -8(%rbp) # Var is on stack
108    movq -8(%rbp), t37    # Copy val to new temp, to not harm it
109    movq t37, %r15        # Moving function parameter to register
110    movq %rbp, t38        # Setting address wanted for static link
111    push t38              # Storing static link for function
112    call factorial        # Calling function
113    addq $8, %rsp         # Remove static link
114    movq %rax, t39        # Saving return value from function in temp
115    movq -8(%rbp), -8(%rbp) # Var is on stack
116    movq -8(%rbp), t40    # Copy val to new temp, to not harm it
117    movq 0(%rbp), 0(%rbp) # Var is on stack
118    cmp $0, 0(%rbp)      # Check if variable is initialized
119    je uninit_var        # Jump to error label

```

```

120     movq $0, t41           # Want to get value at index 0 of array
121     addq 0(%rbp), t41      # Get value at index 0
122     movq MEM(,t41,8), t42  # Copy length of array to reg
123     cmp t42, t40           # Compare index to size
124     jge array_index       # Jump to error label
125     cmp $0, t40           # Compare index to 0
126     jl array_index        # Jump to error label
127     leaq 1(t40), t43       # Copy val to new temp to not harm it, also add 1 to get correct
                             index
128     addq 0(%rbp), t43      # Adding index to start of array
129     movq MEM(,t43,8), MEM(,t43,8) # Used to get target for next instruction
130     movq t39, MEM(,t43,8)  # Assigning value to var
131     movq -8(%rbp), -8(%rbp) # Var is on stack
132     movq -8(%rbp), t44     # Copy val to new temp, to not harm it
133     movq $1, t45           # Moving constant to register
134     addq t44, t45          # Addition
135     movq -8(%rbp), -8(%rbp) # Var is on stack
136     movq t45, -8(%rbp)     # Assigning value to var
137     jmp loop_start_1       # Jump to start of while
138 loop_end_1:               # End of for
139     movq $0, t46           # Moving constant to register
140     movq -16(%rbp), -16(%rbp) # Var is on stack
141     movq t46, -16(%rbp)    # Assigning value to var
142 loop_start_2:             # Start of for
143     movq -16(%rbp), -16(%rbp) # Var is on stack
144     movq -16(%rbp), t47    # Copy val to new temp, to not harm it
145     movq $10, t48          # Moving constant to register
146     cmp t48, t47           # Compare, LT
147     jl cmpTrue_5           # If true, jump to label
148     movq $0, t49           # Setting result to 0 (false)
149     jmp endCMP_5           # Jump to after compare label
150 cmpTrue_5:               # Compare true label
151     movq $1, t49           # Setting result to 1 (true)
152 endCMP_5:                 # After compare label
153     movq t49, t49          # Used to get "target" when creating next instruction
154     cmp $1, t49            # Check if condition in while is true
155     jne loop_end_2         # If condition is false, jump to end
156     movq -16(%rbp), -16(%rbp) # Var is on stack
157     movq -16(%rbp), t50    # Copy val to new temp, to not harm it
158     movq 0(%rbp), 0(%rbp)  # Var is on stack
159     cmp $0, 0(%rbp)        # Check if variable is initialized
160     je uninit_var          # Jump to error label
161     movq $0, t51           # Want to get value at index 0 of array
162     addq 0(%rbp), t51      # Get value at index 0
163     movq MEM(,t51,8), t52  # Copy length of array to reg
164     cmp t52, t50           # Compare index to size
165     jge array_index       # Jump to error label
166     cmp $0, t50           # Compare index to 0
167     jl array_index        # Jump to error label
168     leaq 1(t50), t53       # Copy val to new temp to not harm it, also add 1 to get correct
                             index
169     addq 0(%rbp), t53      # Adding index to start of array
170     movq MEM(,t53,8), MEM(,t53,8) # Used to get target for next instruction
171     movq MEM(,t53,8), t54  # Copy val to new temp, to not harm it
172     movq $.wrt_INT, %rdi   # First argument for printf
173     movq t54, %rsi         # Second argument for printf
174     movq $0, %rax          # No vector arguments
175     call printf            # Calling printf
176     movq -16(%rbp), -16(%rbp) # Var is on stack
177     movq -16(%rbp), t55    # Copy val to new temp, to not harm it
178     movq $1, t56           # Moving constant to register
179     addq t55, t56          # Addition
180     movq -16(%rbp), -16(%rbp) # Var is on stack

```

```

181     movq t56, -16(%rbp)    # Assigning value to var
182     jmp loop_start_2      # Jump to start of while
183 loop_end_2:              # End of for
184 main_end:                # End of body
185     movq $0, %rax         # Return "no error" exit code
186     ret                  # Program return
187
188 array_index:             # Add array index runtime check
189     movq $1, %rax         # Interrupt code
190     movq $2, %rbx         # Set return code
191     int $0x80             # Call exit
192
193 positive_allocate:       # Add positive allocate runtime check
194     movq $1, %rax         # Interrupt code
195     movq $4, %rbx         # Set return code
196     int $0x80             # Call exit
197
198 uninit_var:              # Add uninitialized variable runtime check
199     movq $1, %rax         # Interrupt code
200     movq $5, %rbx         # Set return code
201     int $0x80             # Call exit
202
203 out_of_mem:              # Add out of memory runtime check
204     movq $1, %rax         # Interrupt code
205     movq $6, %rbx         # Set return code
206     int $0x80             # Call exit

```

This code is the first code we generate, which means we have not gone through the first peephole phase yet. After going through the first peephole phase, we can output the program again. We have chosen to shorten the amount of the output we put in the report for this step:

```

1
2 main:                    # Start of body
3     movq $1, MEM          # Init MEM
4     movq $0, 0(%rbp)      # Init to zero
5     movq $0, -8(%rbp)     # Init to zero
6     movq $0, -16(%rbp)    # Init to zero
7     movq $10, t29         # Moving constant to register
8     cmp $0, t29           # Check if allocated length is positive
9     jle positive_allocate # Jump to error label
10    movq MEM, t31          # Move pointer to MEM to another register
11    movq t29, t30          # Moving length of array to new reg
12    addq $1, t30           # Add 1 to array length
13    movq t30, MEM(,t31,8)  # Move allocated length to mem location
14    movq t31, 0(%rbp)      # Move mem location to variable
15    addq t31, t30          # Add length of array to mem register
16    cmp $80000, t29        # Check if out of memory
17    jge out_of_mem        # Jump to error label
18    movq t30, MEM          # Update mem pointer
19    movq $0, t32           # Moving index 0 to register
20    addq 0(%rbp), t32      # Adding index to start of array
21    movq t29, MEM(,t32,8)  # Assigning length of array to index 0
22    movq $0, t33          # Moving constant to register
23    movq t33, -8(%rbp)     # Assigning value to var
24 loop_start_1:           # Start of for
25     movq -8(%rbp), t34     # Copy val to new temp, to not harm it
26     movq $10, t35         # Moving constant to register
27     cmp t35, t34          # Compare, LT
28     jl cmpTrue_4          # If true, jump to label
29     movq $0, t36          # Setting result to 0 (false)
30     jmp endCMP_4          # Jump to after compare label
31 cmpTrue_4:              # Compare true label
32     movq $1, t36          # Setting result to 1 (true)

```

```

33 endCMP_4:                                # After compare label
34     cmp $1, t36                          # Check if condition in while is true
35     jne loop_end_1                       # If condition is false, jump to end
36     movq -8(%rbp), t37                   # Copy val to new temp, to not harm it
37     movq t37, %r15                       # Moving function parameter to register
38     movq %rbp, t38                       # Setting address wanted for static link
39     push t38                             # Storing static link for function
40     call factorial                       # Calling function
41     addq $8, %rsp                        # Remove static link
42     movq %rax, t39                       # Saving return value from function in temp
43     movq -8(%rbp), t40                   # Copy val to new temp, to not harm it
44     cmp $0, 0(%rbp)                     # Check if variable is initialized
45     je uninit_var                       # Jump to error label
46     movq $0, t41                         # Want to get value at index 0 of array
47     addq 0(%rbp), t41                   # Get value at index 0
48     movq MEM(,t41,8), t42               # Copy length of array to reg
49     cmp t42, t40                         # Compare index to size
50     jge array_index                     # Jump to error label
51     cmp $0, t40                         # Compare index to 0
52     jl array_index                      # Jump to error label
53     leaq 1(t40), t43                     # Copy val to new temp to not harm it, also add 1 to get correct
        index
54     addq 0(%rbp), t43                   # Adding index to start of array
55     movq t39, MEM(,t43,8)               # Assigning value to var
56     movq -8(%rbp), t44                   # Copy val to new temp, to not harm it
57     movq $1, t45                         # Moving constant to register
58     addq t44, t45                       # Addition
59     movq t45, -8(%rbp)                   # Assigning value to var
60     jmp loop_start_1                     # Jump to start of while
61 loop_end_1:                             # End of for
62     movq $0, t46                         # Moving constant to register
63     movq t46, -16(%rbp)                 # Assigning value to var
64 loop_start_2:                           # Start of for
65     movq -16(%rbp), t47                 # Copy val to new temp, to not harm it
66     movq $10, t48                       # Moving constant to register
67     cmp t48, t47                         # Compare, LT
68     jl cmpTrue_5                       # If true, jump to label
69     movq $0, t49                         # Setting result to 0 (false)
70     jmp endCMP_5                       # Jump to after compare label
71 cmpTrue_5:                             # Compare true label
72     movq $1, t49                         # Setting result to 1 (true)
73 endCMP_5:                               # After compare label
74     cmp $1, t49                         # Check if condition in while is true
75     jne loop_end_2                       # If condition is false, jump to end
76     movq -16(%rbp), t50                 # Copy val to new temp, to not harm it
77     cmp $0, 0(%rbp)                     # Check if variable is initialized
78     je uninit_var                       # Jump to error label
79     movq $0, t51                         # Want to get value at index 0 of array
80     addq 0(%rbp), t51                   # Get value at index 0
81     movq MEM(,t51,8), t52               # Copy length of array to reg
82     cmp t52, t50                         # Compare index to size
83     jge array_index                     # Jump to error label
84     cmp $0, t50                         # Compare index to 0
85     jl array_index                      # Jump to error label
86     leaq 1(t50), t53                     # Copy val to new temp to not harm it, also add 1 to get correct
        index
87     addq 0(%rbp), t53                   # Adding index to start of array
88     movq MEM(,t53,8), t54               # Copy val to new temp, to not harm it
89     movq $.wrt_INT, %rdi                 # First argument for printf
90     movq t54, %rsi                       # Second argument for printf
91     movq $0, %rax                       # No vector arguments
92     call printf                          # Calling printf
93     movq -16(%rbp), t55                 # Copy val to new temp, to not harm it

```



```

94     movq $1, t56           # Moving constant to register
95     addq t55, t56          # Addition
96     movq t56, -16(%rbp)    # Assigning value to var
97     jmp loop_start_2      # Jump to start of while
98 loop_end_2:               # End of for
99 main_end:                 # End of body
100    movq $0, %rax          # Return "no error" exit code
101    ret                    # Program return
102
103 array_index:              # Add array index runtime check
104     movq $1, %rax          # Interrupt code
105     movq $2, %rbx          # Set return code
106     int $0x80              # Call exit
107
108 positive_allocate:        # Add positive allocate runtime check
109     movq $1, %rax          # Interrupt code
110     movq $4, %rbx          # Set return code
111     int $0x80              # Call exit
112
113 uninit_var:               # Add uninitialized variable runtime check
114     movq $1, %rax          # Interrupt code
115     movq $5, %rbx          # Set return code
116     int $0x80              # Call exit
117
118 out_of_mem:               # Add out of memory runtime check
119     movq $1, %rax          # Interrupt code
120     movq $6, %rbx          # Set return code
121     int $0x80              # Call exit

```

The next output is after doing the liveness analysis, register allocation, and second peephole phase. The result of the liveness analysis and register allocation (such as the "use" and "def" sets, interference graph and list of spilled nodes) are now shown here, but can be seen by using the "-v" flag when compiling:

```

1  .wrt_INT:                 # Integer write label
2  .string "%d\n"            # String used for printing integers
3  .wrt_TRUE:                # TRUE write label
4  .string "TRUE\n"          # String used to print TRUE
5  .wrt_FALSE:               # FALSE write label
6  .string "FALSE\n"         # String used for printing FALSE
7  .comm MEM, 80000          # Available memory
8  .globl main
9  factorial:                # Start of body
10     push %rbp              # Push old base pointer to stack
11     movq %rsp, %rbp        # Move stack pointer to base pointer
12     movq $0, %rax          # Moving constant to register
13     cmp %rax, %r15          # Compare, EQ
14     je cmpTrue_1           # If true, jump to label
15     movq $0, %rbx          # Setting result to 0 (false)
16     jmp endCMP_1           # Jump to after compare label
17 cmpTrue_1:                # Compare true label
18     movq $1, %rbx          # Setting result to 1 (true)
19 endCMP_1:                  # After compare label
20     movq $1, %rax          # Setting default value of result to true
21     cmp $1, %rbx           # Compare left side of OR with true
22     je endBoolCMP_2        # If true, skip right expression
23     movq $1, %rax          # Moving constant to register
24     cmp %rax, %r15          # Compare, EQ
25     je cmpTrue_3           # If true, jump to label
26     movq $0, %rax          # Setting result to 0 (false)
27     jmp endCMP_3           # Jump to after compare label
28 cmpTrue_3:                # Compare true label
29     movq $1, %rax          # Setting result to 1 (true)

```

```

30 endCMP_3:                # After compare label
31 endBoolCMP_2:            # OR expression label
32     cmp $1, %rax          # Check if IF expression is true
33     jne else_1            # Expression is false, jump to ELSE part
34     movq $1, %rax         # Moving constant to register
35     jmp end_factorial     # Jump to functions end label
36     jmp if_end_1          # Skip ELSE part
37 else_1:                  # Start of ELSE
38     movq %r15, %rbx       # Copy val to new temp, to not harm it
39     push %r15             # Register is live in function, so saving it before CALL
40     push %rbx             # Register is live in function, so saving it before CALL
41     movq $1, %rax         # Moving constant to register
42     subq %rax, %r15       # Subtraction
43     movq 16(%rbp), %rax   # Retrieving static link
44     push %rax             # Storing static link for function
45     call factorial        # Calling function
46     addq $8, %rsp         # Remove static link
47     movq %rax, %rdx       # Saving return value from function in temp
48     pop %rbx             # Register was live in function, so restoring it after CALL
49     pop %r15             # Register was live in function, so restoring it after CALL
50     movq %rbx, %rax       # Using RAX for multiplication
51     imul %rdx             # Multiplication using RAX and RDX
52     jmp end_factorial     # Jump to functions end label
53 if_end_1:                # End of IF
54 end_factorial:           # End of body
55     movq %rbp, %rsp       # Restore old stack pointer
56     pop %rbp             # Restore old base pointer
57     ret                  # Return from function
58
59 main:                    # Start of body
60     push %rbp             # Push old base pointer to stack
61     movq %rsp, %rbp       # Move stack pointer to base pointer
62     subq $24, %rsp        # Make space for variables and spills
63     movq $1, MEM          # Init MEM
64     movq $0, 0(%rbp)      # Init to zero
65     movq $0, -8(%rbp)     # Init to zero
66     movq $0, -16(%rbp)    # Init to zero
67     movq $10, %rsi        # Moving constant to register
68     cmp $0, %rsi          # Check if allocated length is positive
69     jle positive_allocate # Jump to error label
70     movq MEM, %rax         # Move pointer to MEM to another register
71     movq %rsi, %rdx       # Moving length of array to new reg
72     addq $1, %rdx         # Add 1 to array length
73     movq %rdx, MEM(,%rax,8) # Move allocated length to mem location
74     movq %rax, 0(%rbp)    # Move mem location to variable
75     addq %rax, %rdx       # Add length of array to mem register
76     cmp $80000, %rsi      # Check if out of memory
77     jge out_of_mem       # Jump to error label
78     movq %rdx, MEM        # Update mem pointer
79     movq $0, %rax         # Moving index 0 to register
80     addq 0(%rbp), %rax    # Adding index to start of array
81     movq %rsi, MEM(,%rax,8) # Assigning length of array to index 0
82     movq $0, %rax         # Moving constant to register
83     movq %rax, -8(%rbp)   # Assigning value to var
84 loop_start_1:            # Start of for
85     movq -8(%rbp), %rdx   # Copy val to new temp, to not harm it
86     movq $10, %rax        # Moving constant to register
87     cmp %rax, %rdx        # Compare, LT
88     jl cmpTrue_4         # If true, jump to label
89     movq $0, %rax         # Setting result to 0 (false)
90     jmp endCMP_4         # Jump to after compare label
91 cmpTrue_4:               # Compare true label
92     movq $1, %rax         # Setting result to 1 (true)

```

```

93 endCMP_4:                                # After compare label
94     cmp $1, %rax                          # Check if condition in while is true
95     jne loop_end_1                        # If condition is false, jump to end
96     push %r15                             # Register is live in function, so saving it before CALL
97     push %rbx                             # Register is live in function, so saving it before CALL
98     movq -8(%rbp), %r15                   # Copy val to new temp, to not harm it
99     movq %rbp, %rax                       # Setting address wanted for static link
100    push %rax                             # Storing static link for function
101    call factorial                         # Calling function
102    addq $8, %rsp                          # Remove static link
103    pop %rbx                              # Register was live in function, so restoring it after CALL
104    pop %r15                              # Register was live in function, so restoring it after CALL
105    movq -8(%rbp), %rsi                   # Copy val to new temp, to not harm it
106    cmp $0, 0(%rbp)                      # Check if variable is initialized
107    je uninit_var                         # Jump to error label
108    movq $0, %rdx                         # Want to get value at index 0 of array
109    addq 0(%rbp), %rdx                    # Get value at index 0
110    movq MEM(,%rdx,8), %rdx               # Copy length of array to reg
111    cmp %rdx, %rsi                        # Compare index to size
112    jge array_index                      # Jump to error label
113    cmp $0, %rsi                          # Compare index to 0
114    jl array_index                       # Jump to error label
115    leaq 1(%rsi), %rcx                    # Copy val to new temp to not harm it, also add 1 to get correct
        index
116    addq 0(%rbp), %rcx                    # Adding index to start of array
117    movq %rax, MEM(,%rcx,8)               # Assigning value to var
118    movq -8(%rbp), %rdx                   # Copy val to new temp, to not harm it
119    movq $1, %rax                         # Moving constant to register
120    addq %rdx, %rax                       # Addition
121    movq %rax, -8(%rbp)                   # Assigning value to var
122    jmp loop_start_1                      # Jump to start of while
123 loop_end_1:                             # End of for
124     movq $0, %rax                        # Moving constant to register
125     movq %rax, -16(%rbp)                 # Assigning value to var
126 loop_start_2:                           # Start of for
127     movq -16(%rbp), %rcx                 # Copy val to new temp, to not harm it
128     movq $10, %rax                       # Moving constant to register
129     cmp %rax, %rcx                       # Compare, LT
130     jl cmpTrue_5                         # If true, jump to label
131     movq $0, %rax                        # Setting result to 0 (false)
132     jmp endCMP_5                         # Jump to after compare label
133 cmpTrue_5:                             # Compare true label
134     movq $1, %rax                        # Setting result to 1 (true)
135 endCMP_5:                                # After compare label
136     cmp $1, %rax                          # Check if condition in while is true
137     jne loop_end_2                        # If condition is false, jump to end
138     movq -16(%rbp), %rcx                 # Copy val to new temp, to not harm it
139     cmp $0, 0(%rbp)                      # Check if variable is initialized
140     je uninit_var                         # Jump to error label
141     movq $0, %rax                        # Want to get value at index 0 of array
142     addq 0(%rbp), %rax                    # Get value at index 0
143     movq MEM(,%rax,8), %rax               # Copy length of array to reg
144     cmp %rax, %rcx                       # Compare index to size
145     jge array_index                      # Jump to error label
146     cmp $0, %rcx                         # Compare index to 0
147     jl array_index                       # Jump to error label
148     leaq 1(%rcx), %rbx                    # Copy val to new temp to not harm it, also add 1 to get correct
        index
149     addq 0(%rbp), %rbx                    # Adding index to start of array
150     movq MEM(,%rbx,8), %rbx               # Copy val to new temp, to not harm it
151     push %rax                             # Register is live in function, so saving it before CALL
152     movq $.wrt_INT, %rdi                 # First argument for printf
153     movq %rbx, %rsi                      # Second argument for printf

```

```

154     movq $0, %rax           # No vector arguments
155     call printf             # Calling printf
156     pop %rax                # Register was live in function, so restoring it after CALL
157     movq -16(%rbp), %rcx    # Copy val to new temp, to not harm it
158     movq $1, %rax           # Moving constant to register
159     addq %rcx, %rax         # Addition
160     movq %rax, -16(%rbp)    # Assigning value to var
161     jmp loop_start_2        # Jump to start of while
162 loop_end_2:                 # End of for
163 main_end:                   # End of body
164     addq $24, %rsp           # Remove space for variables and spills
165     movq %rbp, %rsp         # Restore old stack pointer
166     pop %rbp                # Restore old base pointer
167     movq $0, %rax           # Return "no error" exit code
168     ret                     # Program return
169
170 array_index:                # Add array index runtime check
171     movq $1, %rax           # Interrupt code
172     movq $2, %rbx           # Set return code
173     int $0x80               # Call exit
174
175 positive_allocate:          # Add positive allocate runtime check
176     movq $1, %rax           # Interrupt code
177     movq $4, %rbx           # Set return code
178     int $0x80               # Call exit
179
180 uninit_var:                 # Add uninitialized variable runtime check
181     movq $1, %rax           # Interrupt code
182     movq $5, %rbx           # Set return code
183     int $0x80               # Call exit
184
185 out_of_mem:                 # Add out of memory runtime check
186     movq $1, %rax           # Interrupt code
187     movq $6, %rbx           # Set return code
188     int $0x80               # Call exit

```

This is the final program, which is now ready to be run by the user. Running the program gives the following output:

```

1 1
2 1
3 2
4 6
5 24
6 120
7 720
8 5040
9 40320
10 362880

```

Which corresponds to the factorial for the numbers from 0-9.

5 Implementation

5.1 The symbol table

As mentioned in the design section, the symbol table is implemented as a hash table with chaining. This means that when we insert a value into the table that hashes to the same as an already existing test, we "chain" them together, using a linked list.

```

1      SYMBOL *symbol = Malloc(sizeof(SYMBOL));
2      symbol->name = name;
3      symbol->value = value;
4      symbol->stype = st;
5      symbol->next = Malloc(sizeof(SYMBOL));
6
7      //Placed in front of the list
8      symbol->next = t->table[hashValue];
9      t->table[hashValue] = symbol;

```

The code above is a part of the code used to insert a new symbol into the symbol table. To insert a value in the symbol table, a new symbol must be created, which is what is done here. In case that another symbol have been inserted with the same hash value, we let the new symbols "chain" point to its future place in the symbol table, and then insert the symbol into the table. This ensures that if there was a existing symbol at that place in the symbol table, the new symbols "next" will now point to that symbol.

If we want to check for a symbol in the symbol table, we calculate the hash value of the symbol we are looking for, and then check if the symbol is in that place. This process can be seen here:

```

1      SYMBOL *symbol = t->table[hashValue];
2      if (symbol == NULL) {
3          return NULL;
4      } else {
5          while (symbol != NULL) {
6              if (strcmp(name, symbol->name) == 0) {
7                  return symbol;
8              }
9              symbol = symbol->next;
10         }
11     }

```

If a symbol is found, we follow the "chain" until we find the symbol we are looking for. This code snippet is actually from the function "check_local" which only checks one symbol table. When searching for a symbol, we want to follow the symbol tables "next" pointer if necessary, as described in the design section.

5.2 Scanner

The scanner is implemented using Flex, which is a fast lexical analyser generator. It makes sense to choose that one, since it works well with the Bison parser in the parsing phase. It is important, that this implementation follows the specification in Table 1.

```

1  /* abbreviation of symbols we match on */
2  SYMBOLS [+\\-*\\/\\(\\)\\[\\]\\{\\}!\\|\\.\\.=;:]
3
4  %%
5  [ \\t]+      /* ignore */;
6  \\n          lineno++;
7
8  {SYMBOLS}    return yytext[0];

```

Above is a small part of the "exp.l" file, which is the file Flex uses to scan the given program. This code handles the different symbols that we want to match on, when scanning the file. We use an list

of all the symbols we can match on, as this was easier than writing each and every symbol out individually. Furthermore, each time a newline symbol is seen "\n" it increments the lineno (line number) variable by one. The lineno variable is used for error reporting. If any error is caught, the compiler can tell where it stumbled upon that error. This is part of the error reporting design, where the compiler will tell which and where any error is happening.

```

1 <COMMENT_MULTI>{
2
3 \n          lineno++;
4 "(*"        nested_comment++;
5 "*)"        { nested_comment--;
6              if (nested_comment == 0){
7                  BEGIN(0);
8              }
9          }
10 .          /* ignore */
11 <<EOF>>    fprintf(stderr, "Comment not closed at the end of the file. Found at line: %i\n",
              lineno); exit(1);
12 }
```

Above is the part of the scanner that takes care of multi-line comments. The state for multi-line comments check for nested comments, and, as described in the design section, returns an error if the comment is not closed by the end of the file.

5.3 Parser

```

1 expression : expression '+' expression
2             {$$ = make_EXP(exp_PLUS, $1, $3);}
3             | expression '-' expression
4             {$$ = make_EXP(exp_MIN, $1, $3);}
5             | expression '*' expression
6             {$$ = make_EXP(exp_MULT, $1, $3);}
7             | expression '/' expression
8             {$$ = make_EXP(exp_DIV, $1, $3);}
9             | '(' expression ')'
10            {$$ = $2;}
```

Above is a small part of the "exp.y" file, which is the file Bison uses for parsing the input from the scanner. This part of the code is a part of the code that handles expressions, by making nodes in the AST.

```

1 function : head body tail
2           {$$ = make_Func($1, $2, $3);}
3           if (check_Func($1, $3) != 0){
4               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);
5               YYABORT;
6           }
7 ;
```

As described in the design section, we want to check if a function has the same name in the head and the tail. This part of the code checks this, by calling the function "check_func()". This function checks the "id" of the given head and tail. If they are not equal, we give the user an error message, stop the parsing, since the input program is not valid.

5.4 Abstract Syntax Tree (AST)

The Abstract Syntax Tree, or AST, is the main structure of the program, before generating code. The AST is build up from the different parts of the grammar, which can be seen in table 2. The tree structure is build from the parser, where each nodes has certain children. This tree structure can then be traversed later, when we f.x. to type checking.

```

1  statement *s;
2      s = NEW(statement);
3      s->lineno = lineno;
4      s->kind = statement_IF_ELSE;
5      s->val.ifthen.expression = e;
6      s->val.ifthen.statement1 = s1;
7      s->val.ifthen.statement2 = s2;

```

The code above shows how a "if-else" node would be build up, where the expression in the "if" statement is one child of the node, and the two statements are two other children.

When traversing the AST, we simply follow the children of the nodes, until we reach the bottom of the tree.

5.5 Pretty printer

As described in the design section, the purpose of the pretty printer is to output the program, before doing code generation. This can be used to check if the AST functions properly, to check what kind of changes that occur during the weeding phase, or to check if the type checker applies the correct types to the variables and constants in the program.

The pretty printer works by traversing the AST, and printing a "pretty" version of the input program. This is done by using indentation and parentheses, to make it possibly look better than the input version.

```

1      prettyHead(f->head);
2      indent_depth++;
3      prettyBody(f->body);
4      indent_depth--;
5      prettyTail(f->tail);

```

The code above is part of the function that prints a function. As mentioned before, we traverse the AST, meaning that given a "head" structure, we start by printing the "head", then the "body" and last the "tail". The structure of the different node can be seen in table 2. This code snippet also show the indentation we have implemented. When printing f.x. a function, we would like to have the "body" of the function indented a bit. The "indent_depth" variable serves as a "counter" to see how deep we need to indent some text.

If choosing to print with types, the type of both variables and expression will be printed.

5.6 Weeder

As described in the design section, the purpose of the weeder is to search for expressions and statements that can be evaluated during the compilation, and report any errors it finds.

This is done by going through the AST, and checking each node for something we can "weed" out.

```

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

```

The code above what how we weeding functions on a "+" expression. This case is reached when both sides of the expression are constant numbers, that we know at compile time. We can then calculate the expression, and replace the node in the AST with a new node that corresponds to the result of the calculation.

A more complicated pattern we try to weed out are boolean expressions. As mentioned in the design section, we try to evaluate boolean expressions during compilation, even though this may cause problems, which was also described in the design section.

```

1      if (left_exp->kind == exp_TERM) {
2          if (left_exp->val.term->kind == term_TRUE) {
3              return expression->val.ops.right;
4          }
5          if (left_exp->val.term->kind == term_FALSE) {

```

```

6           temp = make_Term_boolean(1);
7       }
8   }

```

The code above shows the process of weeding out a "AND" expression. We start by checking the left expression, and based on the boolean value of this, decide if we should just change to expression to be "false" or if we should evaluate the right side as well. Again, as mentioned in the design section, this can cause some problems in an actual program, but for now this kind of behavior is allowed.

5.7 Type checking

As described in the design section, the type checker consist of three phases. All the phases consist of going through the AST and looking for things to do.

Setup phase

The purpose of the setup phase is to setup all the symbols, symbol types, and symbol tables of the program. This is done looking for the names of variables and functions, creating symbols, and adding these symbols to a symbol table.

```

1   head->table = table;
2   symbol_type *st;
3   st = NEW(symbol_type);
4   st->type = symbol_FUNCTION;
5   put_symbol(outer_scope, head->id, 0, st);

```

The code above shows how we insert a symbol corresponding to a function name, and inserting it into the symbol table of the outer scope of the function. The symbol type set to "symbol_FUNCTION", which is used in the later phases of the type checking, where we want to compare the actual types of different symbols.

Pickup phase

The purpose of the pickup phase is to pickup the remaining types of symbols, mainly those that have been defined as a user created type. If such a type is found, we want to resolve the type to a predefined type that we can generate code for, such as an integer or record.

```

1   if (type->kind == type_ID) {
2       SYMBOL *s;
3       s = get_symbol(type->table, type->val.id);
4       if (s == NULL || s->stype->type != symbol_ID) {
5           print_error("Undefined identifier", 0, type->lineno);
6       }
7       temp = resolve_recursive_type(s->stype->val.id_type);
8   }

```

The code above is a part of the "resolve_recursive_type" function, whose purpose is to go through the user defined type of a variable, until a type we know is found. If no such type is found, we make sure to return an error to the user, telling them that their type definitions don't conclude to a concrete, recognizable type.

Check phase

The purpose of the check phase, is to do the actual type checking. This is done by comparing the "symbol type" of each symbol used in f.x. and expression, to make sure they have the proper type.

```

1   if (exp->val.ops.left->stype->type == symbol_INT && exp->val.ops.right->stype->type ==
2       symbol_INT) {
3       st = NEW(symbol_type);
4       st->type = symbol_INT;
5       exp->stype = st;

```



```

6         } else {
7             print_error("Operators in arithmetic expression are not integers", 0, exp
                        ->lineno);
8         }

```

The code above is part of the code that takes care of checking the types of a "+", "-", "*" or "/" expression. As mentioned before, to do the type checking, we compare the "symbol type" of the two sides of the expression. If both sides are not integers, we return an error to the user, informing them what kind of error occurred, and where it occurred.

5.8 Code Generation

General code generation

As described in the design section, we decided on a "template" based approach when we generate the code for a program. One of these templates can be seen here:

```

1 case (exp_PLUS):
2     add_2_ins(&head, &tail, ADDQ, left_target, right_target, "Addition");
3     break;

```

In this case, "left_target" and "right_target" are the results of generating code for the left and right side of a "+" expression. An "ADDQ" instruction is then added to the linked list for the function that handles expressions, where the two operators for the instruction are the two results, or "targets" as we call them.

A more interesting case is when we generate code for "AND" and "OR" expressions.

```

1 if (exp->kind == exp_OR) {
2     make_bool_label(label_bool);
3     temp = make_op_temp(); //Used to hold the result of the expression
4     add_2_ins(&head, &tail, MOVQ, make_op_const(1), temp, "Setting default value of
                        result to true");
5
6     add_2_ins(&head, &tail, CMP, make_op_const(1), left_target, "Compare left side of
                        OR with true");
7     add_1_ins(&head, &tail, JE, make_op_label(label_bool), "If true, skip right
                        expression");
8
9     right_exp = generate_exp(exp->val.ops.right);
10    asm_insert(&head, &tail, &right_exp);
11    right_target = get_return_reg(tail);
12
13    //If this is executed, left expression was false, so the result depends entirely
14    //on the right expression
15    add_2_ins(&head, &tail, MOVQ, right_target, temp, "Result is set to the value of
                        the right expression");
16
17    add_label(&head, &tail, label_bool, "OR expression label");
18
19    add_2_ins(&head, &tail, MOVQ, temp, temp, "Used to get target for next
                        instruction");
20
21    return head;
22 }

```

In this case, we have a "OR" expression. We have generated code for the left side of the "OR" expression, but don't want to generate code for the right side just yet. First we need to insert a check that makes it possible to skip the check of the right part of the "OR" expression. We have chosen to allow this, since only one part of an "OR" expression needs to be true for the whole "OR" expression to be true. An example of the "dummy moves" we mentioned in the design section can also be seen here, as the last instruction we insert into the linked list.

Function Calls

As mentioned in the design section, we allow function arguments to be passed either by registers or stack locations. To handle this, we check the amount of arguments a functions takes when generating code for it.

```

1      //If more than "PLACE_IN_REGS" parameter, place parameters on stack
2      if (s->stype->val.func_type.func->head->args > PLACE_IN_REGS) {
3          el = generate_elist(term->val.list->list, &arg_count);
4          asm_insert(&head, &tail, &el);
5      } else {
6          elist = term->val.list->list;
7          used_reg = AVAIL_REGS - 1;

```

Seen above is how we have chosen to implement this check. We first check if the amount of arguments of the function is larger than what we allow, and then either push arguments to the stack, or put the arguments in registers. If we use registers, we start by using the registers with the "highest number". These are the least used registers, since we allocate register starting from the "lowest number" and up. This means that in the general case, the registers with the "high numbers", will not be used as much. The code after this snippet simply moves the desired values for the function parameters into the desired registers.

Static Linking

As described in the design section, we also need to give the static link to a function when we call it. This is done by first looking at the "depth" of the function we are calling. This "depth" is calculated by following the symbol tables "next" pointer, until we find the table where the function is defined. When we have this "depth", we know how many times we need to follow the static link for the function.

```

1      add_2_ins(&head, &tail, MOVQ, op_STATIC_LINK, static_link, "Retrieving
      static link");
2      //Resolve static link
3      depth--;
4      while (depth > 0) {
5          temp = make_op_stack_loc(0, &static_link);
6          static_link = make_op_temp();
7          add_2_ins(&head, &tail, MOVQ, temp, static_link, "Copying static
      link");
8          depth--;
9      }
10     add_1_ins(&head, &tail, PUSH, static_link, "Storing static link for
      function");

```

The code above is from the case where the "depth" of the function is not "0". This means that we have to follow the static link a couple of times. The operator "op_STATIC_LINK" is constant operator that corresponds to the location "16(%rbp)", which is the location where we retrieve the static link, as described in the design section.

If we need to access any variable, we also need to check if the variable is outside of the scope of the current function. Just like before, we check the "depth" of the variables name, and if the "depth" is not "0", we need to follow the static link.

```

1      while (depth > 0) {
2          temp = make_op_stack_loc(-2, &static_link);
3          static_link = make_op_temp();
4          add_2_ins(&head, &tail, MOVQ, temp, static_link, "Copying static
      link");
5          depth--;
6      }
7      v = make_op_stack_loc((s->offset), &static_link);

```

The code above follows the same principle as before, where we need to follow the static link a certain amount of times, before we reach the variable we want.

Runtime Checks

To implement runtime checks, we started by finding the places where they were needed. This would be when we generate code for f.x. a division expression, if we want to check for a "division-by-zero" error.

```

1      if (runtime_checks) {
2          div_zero_flag = 1;
3          make_div_zero_label(label_true);
4          add_2_ins(&head, &tail, CMP, make_op_const(0), reg_RBX, "Checking if value
           is 0");
5          add_1_ins(&head, &tail, JE, make_op_label(label_true), "Jump to error
           label");
6      }

```

Since runtime checks can be disabled with a flag, we need to first check if they are enabled. We then set a flag that tells the compiler that a label should be inserted in the end of the program, where we return the proper exit code. Otherwise, we simply to the appropriate check, and if the check is true, we jump to the error label.

The way we print the errors from the runtime checks is by first setting the appropriate code in "RBX", and then interrupting the program.

```

1      add_label(head, tail, "div_zero", "Add division by zero runtime check");
2      add_2_ins(head, tail, MOVQ, make_op_const(1), reg_RAX, "Interrupt code");
3      add_2_ins(head, tail, MOVQ, make_op_const(3), reg_RBX, "Set return code");
4      add_1_ins(head, tail, INT_, make_op_label("$0x80"), "Call exit");

```

The code above shows how this interruption is done. We use the assembly instruction "int" with the operator "0x80", which, with the value "1" in "RAX", interrupts the program. Originally, we used a "syscall" instead of the "int" instruction, but this caused the program to be interrupted with a "segmentation fault" instead of just stopping with no output.

5.9 Liveness Analysis

As described in the design section, the liveness analysis is split into two main parts, building the flowgraph, and doing the actual analysis.

Building the flowgraph

To build the flowgraph, we simply build a graph, where each node corresponds to an instruction in the program, and each edge shows the possible "direction" the program can take from that instruction. In most cases, the direction to take is just the next instruction, but then the instruction is a conditional "jump" instruction, another edge needs to be added.

```

1      jump_target = find_label_node(g, temp->val.one_op.op->val.label_id);
2      add_edge(list->head, jump_target->head);
3      if (list->next != NULL) {
4          add_edge(list->head, list->next->head);
5      }

```

The code above is from the case where the instruction of a node is a conditional "jump" instruction. In that case we search through the program for the node that corresponds to that label, and add a edge between them.

When creating the flowgraph we also set the initial "use" and "def" sets of each node. This is done by checking what kind of instruction the nodes has, and then, using 6, setting the appropriate operators as used or defined by that node.

```

1      case (MOVQ):
2          set_op_bit(op->val.two_op.op1, use, def, 1, 0);

```

```

3      set_op_bit(op->val.two_op.op2, use, def, 0, 1);
4      break;

```

The code above shows how we set "use" and "def" sets for a "move" instruction. The last two parameters in the "set_op_bit" function determines whether the operator given to the function is used or/and defined, respectively. Using the information from table 6, we set the 1. operator to be used and not defined, and the 2. operator to be defined, and not used.

Analysis

The calculation of the "new" set is done as described in the design section.

```

1      successor = (struct a_asm *)get_data(succ->head);
2      if (succ->head->key > list->head->key) {
3
4          tempV = vector_union(tempV, successor->new);
5
6      } else {
7          tempV = vector_union(tempV, successor->old);
8      }

```

The code above shows the check we make, to see if the successor of a node has a higher "key" than the current node. The result of this check is the same as described in the design section, where we chose the successors "new" set if its key is higher, and the "old" set otherwise.

5.10 Register Allocation

The implementation of the register allocation followed the algorithms provide by the book [1], pretty closely. However, some changes were made to the algorithms, to implement them to our liking.

```

1      //Store the instruction a table, so we can retrieve them later
2      table_enter(move_table, (void *)move_counter, program);
3      set_bit(worklist_moves, move_counter);

```

The code above is from the "build_interference_graph", where we want to add a "move" instruction to a list, so we can possibly coalesce it later. Instead of an actual list, we have chosen to use a table instead. This was done since the "sets" we use are actually bit vectors, and therefore can't hold an actual value, such as the node the "move" instruction is from. A way to still use the bit vectors would be to set a bit that corresponds to the placement of the node of the "move" instruction in the program. However, due to our implementation of bit vectors, the size of the vectors can only accommodate the amount of temporaries in the program, which is not necessarily larger than the amount of move instructions in the program.

Therefore we decided to use a hash table, so we could lookup the move later in the program.

Rewriting the program was one thing were there wasn't a specific algorithm to follow. As described in the design section, the purpose of this phase is to rewrite the spilled temporaries in the program, by adding "fetches" and "stores" to the program. Using the information from table 7, we can set the operators that need to be fetched and stored in the program.

```

1      case (MOVQ):
2          //Fetch op1 if it's spilled, store op2 if it's spilled
3          left = rewrite_spill_reg(&theprogram->val.two_op.op1, 1, NULL);
4          asm_insert(&head, &tail, &left);
5
6          asm_insert_one(&head, &tail, &theprogram);
7
8          right = rewrite_spill_reg(&theprogram->val.two_op.op2, 0, NULL);
9          asm_insert(&head, &tail, &right);
10
11      break;

```

Above is the code for adding fetches and stores for a "move" instruction. As seen in table 7, we fetch the 1. operator if its spilled, and store the 2. operator if its spilled. The "1" and "0" in the "rewrite_spill_reg" function, tells the function if the operator is to be stored or fetched.

A more advanced example of adding fetches and stores, would be in a "add" instruction:

```

1      left = rewrite_spill_reg(&theprogram->val.two_op.op1, 1, NULL);
2      asm_insert(&head, &tail, &left);
3
4      right = rewrite_spill_reg(&theprogram->val.two_op.op2, 1, &new_temp);
5      asm_insert(&head, &tail, &right);
6
7      asm_insert_one(&head, &tail, &theprogram);
8
9      free(right);
10
11     right = rewrite_spill_reg(&theprogram->val.two_op.op2, 0, &new_temp);
12
13     asm_insert(&head, &tail, &right);

```

Here we need to both use and define the 2. operator. To do this, we start by giving the function "rewrite_spill_reg" an empty operator, which is then changed to a temporary. This temporary is used to store the value of the second operator, but only temporarily, since we don't want new temporaries to have too big of a lifespan. The value of this temporary is then used for the add instruction, and then the value is pushed to the the stack location of the 2. operator. Thus the temporary is no longer live, and can be freely overwritten.

5.11 Rewriter

As described in the design section, the rewriter is responsible for couple of things, and not just changing temporaries to registers.

Replacing temporaries

When going through the different instructions in the program, we of course, rewrite temporaries by changing them to registers.

```

1      temp = get_reg((*op));
2      if (temp != -1) {
3          if (!is_precolored(temp)) {
4              new_reg = get_corresponding_reg(colors[temp]);
5              replace_temp_op(op, new_reg);
6          }
7      }

```

The code above shows how we rewrite a temporary that is not a spill variable. This is done by checking if the temporary is precolored (meaning that the temporary is a legal register), and if not, we change the temporary to be the register it was colored as in the register allocation.

Function rewriting

To rewrite a function, we keep track of the start and the end of the function. When we find the end of the function, we add the post- and prefix for the function, as shown here:

```

1      add_1_ins(&head, &tail, PUSH, reg_RBP, "Push old base pointer to stack");
2      add_2_ins(&head, &tail, MOVQ, reg_RSP, reg_RBP, "Move stack pointer to base pointer");
3
4      //Add space for variables and spills in function
5      if (offset != 0) {
6          add_2_ins(&head, &tail, SUBQ, make_op_const(offset * 8), reg_RSP, "Make space for
          variables and spills");
7      }

```

The code above is from the "add_prefix" function, which add the storing of the basepointer and such, as is convention. We also add space on the stack in the start of a function. The value "offset" corresponds to the amount of variables in a function. To make space for each one, we allocate a byte of space for each variable. In the post fix of the function, this space is removed again, as to not cause any segmentation faults.

Adding push and pop of live registers

When checking if we need to add a push and pop of a register before function call, we need to know what registers are used in the function. This is done by simply going through the program until we find the function we are looking for, and then noting what registers are defined by the function, which lasts until we meet a "ret" instruction.

When we know what register are defined in a register we need to add the actual push and pop of the registers.

```

1      for (int j = 0; j < temps; j++) {
2          if (get_bit(call->new, j) && regs[colors[j]] && pushed_regs[colors[j]]) {
3              pushed_regs[colors[j]] = 0;
4              add_1_ins(&head, &tail, PUSH, get_corresponding_reg(colors[j]), "Register
5                  is live in function, so saving it before CALL");
6          }
    }
```

The code above shows how we check if a register needs to be pushed before a function call. The check itself consist of checking if a given register is currently live, meaning that it contains some value we want to preserve, if the register is defined in the function we are calling, which is represented by the array "regs", and if we have already pushed the register, represented by the "pushed_regs" register. If the register is live and is defined in the function, add a push instruction, and set the register as pushed in the "pushed_regs" array.

To restore the registers after a function call, we simply go backwards through the list of temporaries/registers, meaning that the first temporaries/registers that are popped, are those that were pushed last, giving us the proper values in the proper registers.

5.12 Peephole

As described in the design section, the peephole phase consists of going through the program, and finding the places where we can optimize something.

Removing move-to-self

For removing instructions where we move some value to itself we need to check if the two operators of the move instruction truly are the same. For normal registers and temporaries this is pretty simple, since we can just get the id of the temporary or the check if the registers are the same. However, for checking stack locations and such, we need to check the register the offset is set from. If we for example have the following move instruction:

```

1  movq 0(%rbp), 0(%rax)
```

In this case we need to check if the register used in the operators are the same.

If we find an move instruction where both operators are the same, we want to remove it from the program, since it is a useless instruction at this point.

```

1      head = head->next;
2      current = head->prev;
3      current->prev->next = head;
4      current->next->prev = current->prev;
5      free(current);
```

In the code above, we have found a move instruction where the operators are the same. Since we want to remove this instruction, we make the previous node in the linked list point to the next node and

vice versa, as described in the design section. We also make sure to not lose track of the node we want to remove, so we can properly "free" it.

5.13 Assembly printer

The implementation of the assembly printer is pretty simple. Its sole purpose is to go through the program, and print the correct output for each instruction. Since we have chosen to add comments to add instructions, these are printed as well.

```
1 case (MOVQ):
2     create_2_op("movq", head->val.two_op.op1, head->val.two_op.op2, head->comment);
3     break;
```

The code above shows how a "move" instruction is printed. The "create_2_op" function takes the two operators the instruction takes, formats them properly, and adds the comment at the end of the line. The output of the assembly printer can be properly seen in "Section 4: Example of a program through all phases".

5.14 Auxiliary structures

Graph

As mentioned in the design section, the graph implementation is base in the implementation by Andrew W. Appel [2]. The graph itself contains a list of nodes, but not a list of edges. The edges are instead associated with each node, where each node has a list of successors and predecessors.

```
1 if (from->graph != to->graph) {
2     return 0;
3 }
4 if (check_edge(from, to)) {
5     return 1;
6 }
7 to->pred = make_list(from, to->pred, NULL);
8 from->succ = make_list(to, from->succ, NULL);
```

The code above shows how we insert an edge into the graph. This is done by first checking if the two nodes are even in the same graph, since adding an edge between nodes not in the same graph wouldn't make much sense. We then check if an edge already exists, and if this is not the case, add the edge. As described in the design section, this is done by adding the nodes to each others successors and predecessors. When checking if an edge exists between two node, we can then go through these list, and check for the node we are looking for there.

Bit vector

As mentioned in the design section, the bit vector implementation is base of the implementation by Shun Yan Cheung [3]. The bit vector itself is just an array of integers, but instead of having each integer represent one bit, each integer represents 32 bits.

```
1 i = bit / 32;
2 pos = bit % 32;
3 flag = 1;
4
5 flag = flag << pos;
6
7 bv[i] = bv[i] | flag
```

The code above shows how we set a bit in the bit vector. As described in the design section, we start by having the integer "1", which we then shift the desired amount of positions. This value is then used in a bitwise "OR" operation with the integer at the appropriate index, setting the corresponding bit.

Table

As mentioned in the design section, the graph implementation is based in the implementation by Andrew W. Appel [2]. The table itself functions as a hash table, but with the possibility of hashing any value/struct.

```
1      int index;  
2      index = ((unsigned)key) % TABSIZE;  
3      t->table[index] = table_binder(key, value, t->table[index], t->top);  
4      t->top = key;
```

The code above shows how we insert a value into the hash table. By casting the "value" as a void pointer, we can insert anything into the hash table. A hash value is then calculated for the value we want to insert, and it is put into the hash table. By calculating this value again, we are able to get the value out of the table again.

Stack

The implementation of the stack was basically that of a linked list, but where only the top node is available.

```
1      stack_node *sn;  
2      sn = NEW(stack_node);  
3      sn->val = val;  
4      sn->next = stack->top;  
5      stack->top = sn;
```

The code above shows how a value would be put on the stack. We simply make a new node for the stack, and set that node as the new top. This node is then available for looking at if needed, and can be popped if necessary.

6 Testing

6.1 Our tests

This section is for all the tests made by the group. All tests in this section are executed with the following options:

```
1 $ ./compiler -aef -o ./out/ /tests/<the program to test>
```

This executes the program if it is a valid program. Therefore the output can both be the output from the compiler, when compiling, but it can also be the output from the compiled program, when that is executed.

general.src

```
1 func fac(n: int): int
2   if (n < 1) then
3     return 1;
4   else
5     return n * fac(n-1);
6 end fac
7
8 write fac(10);
```

This program contains many tests in one, yet the most important part of this test is recursion. It is supposed to calculate the faculty of some integer input using recursion. When running general.src it gives the following output:

```
1 > 3628800
```

From this very first test many things can be derived. First of all, the outputted result is correct since

$$10! = 3628800 \quad (1)$$

. Besides that, it shows that a function can be defined called with some arbitrary input, in this case 10. It can also be assumed, that if-else statements are working. However, since this is the first test, it could just be a lucky sample, thus more tests on this topic will be performed. What this test definitely shows is, that return and recursion works, since recursion can only work if return works.

or-test.src

```
1 func test(n : int) : int
2   if (n == 0 || n == 1 || ||a+b+c| == n) then
3     return -1;
4   else
5     return n;
6   end test
7
8 write test(1);
```

The or-test.src is made to show, that the compiler can handle both absolute values and or statements in the same expression. Now the as-is is actually a little off in terms of what it was supposed to show. It appears the a, b and c has not been defined, and while this shows some of the error reporting functionality, it does not show that e both absolute values and or statements in the same expression works.

```
1 > Symbol 'a' not defined at line 4
```

It simply does what is actually expected. Now it only tells that "a" is not defined at line 4 and not "b" and "c" as well. This is because it halts execution as soon as it catches an error. Now if we liberate ourself and actually fix this test:

```

1  var a : int;
2  var b : int;
3  var c : int;
4  func test(n : int) : int
5      if (n == 0 || n == 1 || ||a+b|+c| == n) then
6          return -1;
7      else
8          return n;
9      end test
10
11 write test(1);
12 write test(2);

```

It does show, that the compiler can handle both absolute values and or statements in the same expression. This can be seen in the output, which should be first a "-1" and then a "2".

```

1  > -1
2  > 2

```

- which is also the case.

typechecker/factorial.src

This test is exactly the same as general.src described above. Only this time the input value is 5 instead of 10. This means that the calculated answer using recursion should be:

$$5! = 120 \quad (2)$$

Output of program:

```

1  > 120

```

As shown, it also calculates the factorial of 5 correctly.

typechecker/list.src

The list test is testing whether multidimensional arrays are working.

```

1  # Code generation for arrays and records, nesting
2  type arrContent = record of {a : int, b : int, c : int};
3  type myArr1 = array of arrContent;
4  type myArr2 = array of myArr1;
5
6
7  var a : myArr2;
8  var b : int;
9  var t1 : int, t2 : int, t3 : int;
10 var p1 : arrContent;
11 var p2 : myArr1;
12
13
14 allocate a of length 5;
15
16 t1 = 0;
17 while ( t1 < |a| ) do
18 {
19     allocate a[t1] of length 6;
20     t2 = 0;
21     while ( t2 < |a[t1]| ) do
22     {
23         allocate a[t1][t2];
24         b = t1*2+t2;
25

```

```

26     a[t1][t2].a = b;
27     write b;
28
29
30     t2 = t2 + 1;
31 }
32 t1 = t1 + 1;
33 }
34
35 p1 = a[3][1];
36 write p1.a;

```

This does not work, and therefore we instead check for runtime errors.

```

1 $> echo $?
2 > 5

```

It returns 5, which means there is an uninitialized variable. This is probably because multidimensional arrays cannot be initialized properly.

typechecker/par_test.src

The par_test.src is about showing, that multiple variables can be present in the declaration of a variable "d".

```

1 var a : int;
2 var b : int;
3 var c : int;
4 var d : int;
5
6 a = 2;
7 b = 3;
8 c = 5;
9 d = (a*b)/c;
10
11 write d;

```

The following program should output

$$\frac{(2 \cdot 3)}{5} = d = 1.2 = 1 \quad (3)$$

The answer should be 1, since the language does not handle floating point numbers, or decimal numbers to say.

```

1 > 1

```

It does as expected and outputs a 1. This shows that declared variables can be assigned values and also that they afterwards can be used in a calculation, where the result of that calculation is stored in another variable "d".

typechecker/record.src

The record.src test is about testing the assigning and accessing of values in a record. To test this, the values of the record are added together.

```

1 # Type check of records, simple example of use of records
2 type recordType = record of {
3     x: int,
4     y: int
5 };
6 var p : recordType;
7

```

```

8 allocate p;
9
10 p.x = 1;
11 p.y = 2;
12 write p.x;
13 write p.y;
14 write p.x + p.y;

```

The calculation between p.x and p.y should give the following:

$$1 + 2 = 3 \quad (4)$$

The accessing of the variables of the record is the two first writes and the calculation is the third write.

```

1 > 1
2 > 2
3 > 3

```

This follows the expected output of the program. First it correctly accessed the variables of record "p" and then it correctly adds these values in the last output.

typechecker/test_arithmetic_typecheck.src

```

1 #test14.src
2 var a : int;
3 var b : int;
4 a = 1;
5 b = 3;
6
7 write a + b;

```

This test is used to test, whether the type checker allows a and b to be added. It is expected that this will work, since previous tests have already shown this. However, this is a formal test of the type checker's validity.

```

1 > 4

```

As expected, it outputs the correct value and it can be assumed, that the type checker allows integers to be added together.

typechecker/test_arithmetic_wrong_types.src

This test is an extension of the previous test. In this test two variables with different types are added together.

```

1 #test15.src
2 var a : int;
3 var b : bool;
4 a = 1;
5 b = true;
6
7 write a + b;

```

It has been defined, that integers and boolean types can not be used together in an arithmetic expression. Therefore it is expected, that the compiler will report this and that it tells where the error is present in the program

```

1 Operators in arithmetic expression are not integers at line 7

```

As expected, it tells that the expression on line 7 is invalid, as not all operators are integers. Therefore, the type checker not only allows correct expressions, as shown in previous test, but it also disallows incorrect expressions like the one in this test.

typechecker/test_array_index.src

This is another test for the type checker. Here an array is made and the value at index "true" accessed. However, accessing at index "true" makes very little sense, and therefore the type checker should report this and also where this is happening.

```

1 #test20.src
2 # Code generation of arrays, length
3 var a : array of int;
4 allocate a of length 7;
5 write a[true];

```

The output:

```

1 Expression in [] not an integer at line 5

```

As shown, it reports that it is not an integer at line 5. This fits well with the expected output.

typechecker/test_function_arguments_to_many.src

This is about testing whether the compiler will notice, that too many parameters for a function has been given, when calling it. The function requires two parameters, but three is given, when it is called. The rest of the program should be valid, which will be implicitly verified, if it returns the "Too many arguments" error, since this error is happening in the very bottom of the program.

```

1 #test18.src
2 # Type check and code generation, returning records from functions
3 type recordType = record of {
4     x: int,
5     y: int
6 };
7
8 func a(x : int, y : int) : recordType
9     var p2 : recordType;
10    allocate p2;
11    p2.x = x;
12    p2.y = y;
13    return p2;
14 end a
15
16 var p1 : recordType;
17 var b : bool;
18 b = true;
19
20 p1 = a(10, 2, 1);
21
22 write p1.x / p1.y;

```

The output:

```

1 > Too many function arguments at line 20

```

The compiler correctly identifies, that too many arguments are given when calling the function at line 20.

typechecker/test_function_arguments_too_few.src

This test is the essentially the opposite of the previous test. Only difference is that now there are too few parameters given to the called function.

```

1 #test17.src
2 # Type check and code generation, returning records from functions
3 type recordType = record of {

```

```

4           x: int,
5           y: int
6       };
7
8 func a(x : int, y : int) : recordType
9     var p2 : recordType;
10    allocate p2;
11    p2.x = x;
12    p2.y = y;
13    return p2;
14 end a
15
16 var p1 : recordType;
17 var b : bool;
18 b = true;
19
20 p1 = a(10);
21
22 write p1.x / p1.y;

```

The output:

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

Again it correctly sees that there are too few parameters passed to the called function at line 20.

typechecker/test_function_arguments.src

This is a type checker test in the same series of the function tests. Here it the type checker checks whether the input values have the same types as the required parameters of the function. In the test these types are not equal, and it is expected, that the type checker reports this mismatch of types.

```

1 #test16.src
2 # Type check and code generation, returning records from functions
3 type recordType = record of {
4     x: int,
5     y: int
6 };
7
8 func a(x : int, y : int) : recordType
9     var p2 : recordType;
10    allocate p2;
11    p2.x = x;
12    p2.y = y;
13    return p2;
14 end a
15
16 var p1 : recordType;
17 var b : bool;
18 b = true;
19
20 p1 = a(10,b);
21
22 write p1.x / p1.y;

```

Notice that function "a" requires two integers as parameters, but when "a" is called, it is done with an integer and a boolean, which is illegal.

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

As shown, it correctly reports that there is a type mismatch with the parameters for the function call at line 20.

typechecker/test_function_exists.src

In this test the compiler checks whether the called function actually exists.

```

1 #test19.src
2 # Type check and code generation, returning records from functions
3 type recordType = record of {
4     x: int,
5     y: int
6 };
7
8 func a(x : int, y : int) : recordType
9     var p2 : recordType;
10    allocate p2;
11    p2.x = x;
12    p2.y = y;
13    return p2;
14 end a
15
16 var p1 : recordType;
17 var b : bool;
18 b = true;
19
20 p1 = b(10, 2, 1);
21
22 write p1.x / p1.y;
```

In the example the declared function is called "a", but the called function is "b", which does not exist. It is therefore expected, that the compiler tells that this function does not exist and where this function is called in the program.

```
1 > Reference to function 'b', which does not exist at line 20
```

It does exactly that - reports that it cannot find "b", which is being called at line 20. Technically it could have been defined later on, so the error reporting is a little misleading. However, from the perspective of the caller, it does exist yet, since it only knows about declarations above itself in the program.

typechecker/test_function_wrong_return_type.src

Here the type checker will check whether the function return a type equal to the expected return type of the function.

```

1 #test12.src
2 # Type check and code generation, returning records from functions
3 type recordType = record of {
4     x: int,
5     y: int
6 };
7
8 func a(x : int, y : int) : recordType
9     var p2 : recordType;
10    allocate p2;
11    p2.x = x;
12    p2.y = y;
13    return 1;
14 end a
15
16 var p1 : recordType;
17
18 p1 = a(10,2);
19
20 write p1.x / p1.y;
```

It is expected, that the type checker tells that the function "a" is returning a wrong type at line 13.

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

As shown, it does exactly that.

typechecker/test_function_return_type.src

This test is similar to the one above. Only this time, the return type is valid.

```
1 #test13.src
2 # Type check and code generation, returning records from functions
3 type recordType = record of {
4     x: int,
5     y: int
6 };
7
8 func a(x : int, y : int) : recordType
9     var p2 : recordType;
10    allocate p2;
11    p2.x = x;
12    p2.y = y;
13    return 1;
14 end a
15
16 var p1 : recordType;
17
18 p1 = a(10,2);
19
20 write p1.x / p1.y;
```

This program should be valid and run with the calculation:

$$\frac{10}{2} = 5 \quad (5)$$

The output:

```
1 > Segmentation fault (core dumped)
```

This is unexpected, since the given program is valid. This is also what the compiler assumes, however, it appears that the compiled program is not valid.

typechecker/test_recursive_pickup_2.src

The recursive pickup is for checking whether the pickup phase of the type checker is working correctly. Here it checks whether a type actually has a known type.

```
1 #test7.src
2 type r1 = r2;
3 type r2 = r3;
4 type r3 = r1;
5
6 var v1 : r1;
7 var v2 : r2;
8 var v3 : r3;
9 write 42;
```

This test neither r1, r2 nor r3 has a type. The type checker should therefore report, that this is a recursive type definition.

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


As expected it reports, that there is a recursive type definition present in the program. This is good, since it prevents the programmer from accidentally defining types based on other types without any meaning.

typechecker/test_recursive_pickup.src

The second test of these is very similar, but here the third type "r3" has a meaningful value. The other two, "r1" and "r2" are defined from that.

```

1 #test6.src
2 type r1 = r2;
3 type r2 = r3;
4 type r3 = record of { x : int};
5
6 var v1 : r1;
7 var v2 : r2;
8 var v3 : r3;
9 write 42;
```

It is therefore expected, that this example is valid and that it outputs 42.

```

1 > 42
```

weeder/link.src

The link test is used to test the functionality of *static links*.

```

1 # Code generation for static link, referencing of a variable for which
2 # multiple static links must be followed in sequence
3 func func1() : int
4   func func2() : int
5     func func3() : int
6       {
7         i = 10;
8         return 5;
9       }
10    end func3
11    return func3();
12  end func2
13 return func2();
14 end func1
15
16 var i : int;
17
18 i = 0;
19 write i;
20 write func1();
21 write i;
```

Here it is expected, that the integer variable "i" is changed to 10 after the function "func1" has been called. This is due to the static link, which allows the function to edit variables outside of its own scope, as long as these variables are defined before the function call. It should also prove, that a nested function can still access that same variable.

```

1 > 0
2 > 5
3 > 10
```

As shown, it behaves as expected. First the value of "i" is 0, since the function has not been called yet. Then the function is called and it returns 5, but has also changed the value of "i" to 10. This is shown in the last line, where the new value of "i" is also 10.

weeder/test_arithmetic_zero_division.src

This test is for the weeder. Here it should notice, that a zero division is taking place.

```
1 #test1.src
2 var a : int;
3 a = 2/0;
```

It is expected that it reports where the zero division is taking place.

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

This is as expected.

weeder/test_test_if-else_boolean_expression.src

Since this is a weeder test, the weeded out program will be printed as well. This test shows, that the weeder correctly removes sections of the program that can never be executed, no matter the values of the variables. To get the pretty output for that, simply add "-p 1" as an option, when compiling the program.

```
1 #test3.src
2 var n : int;
3 n = 1;
4
5 if ((n>0) || true) then
6   write 1;
7 else
8   write 2;
```

It is expected that the weeder removes the else part, since this expression will always evaluate to True.

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

As shown, it is not able to remove the unused part correctly. However, in the end the program still works correctly

```
1 > 1
```

weeder/test_test_if-else_only_boolean.src

This test is similar to the one above, with the only difference being, that the if statement only contains boolean values.

```
1 #test2.src
2 if (true || false) then
3   write 1;
4 else
5   write 2;
```

As with the previous test, it is also expected here, that it removes the part, which will never be accessible at all.

```
1 write 1;
```

As shown, this time it correctly removes the part, that can never be run at all and reduces the program to a "write 1". The output of that program should therefore be 1, and it is:

```
1 > 1
```

weeder/test_no_return.src

The no return test is to check, that the compiler notices if a function is missing a return statement.

```
1 #test5.src
2 func test(n: int): int
3     var i: int;
4 end test
5
6 write test(1);
```

It is expected that it reports about missing function calls. However, it was never implemented to tell which functions, that were missing these function calls.

```
1 > Syntax error before end at line 4
2 > You appear to be missing any function calls
```

This shows, that it notices, that something is wrong. However, it appears to be giving a wrong error as well, since it also reports that any function calls are missing, which is not the case.

weeder/test_no_return_if.src

This test is very similar to the previous test. Here it checks whether it still notices that a return is missing, while at the same time there being an if statement present. This is an important test, since the compiler could wrongly assume, that a return statement is present inside the if statement.

```
1 #test8.src
2 func test(n: int): int
3     var n : int;
4     if (n == 0) || (n == 1) then
5         n = 3;
6 end test
7
8 write test(1);
```

The output:

```
1 > Function does not contain a return at line 6
```

This time it correctly identifies the error, which is a missing return statement. Now it does not give the name of the function missing that return statement, but instead gives the last line number of that function.

weeder/test_no_return_multiple_functions.src

Here it is tested, whether it still notices a missing return statement, when there are multiple functions present in the program.

```
1 #test9.src
2 func beta(n: int): int
3     var n : int;
4     if (n == 0) || (n == 1) then
5         n = 3;
6         return n;
7 end beta
8
9
10 func test(n: int): int
11     var n : int;
```

```

12   if (n == 0) || (n == 1) then
13       n = 3;
14   end test
15
16   return n;
17   write test(1);
18   write beta(1);

```

It is expected, that it returns an error similar to that of the previous test.

```

1 > Function does not contain a return at line 14

```

As expected it does that and reports that this return is missing at line 14.

weeder/test_return_inside_outside.src

The previous tests were about missing return statements. This test is about placing a return statement outside of a function, in the program body. It does not make sense to have a return statement located there, since there is nowhere to return it.

```

1 #test4.src
2 func test(n: int): int
3     return 2;
4 end test
5
6 return 2;
7
8 write test(1);

```

Here it is expected, that it reports that a return statement is located outside of a function.

```

1 > Return outside of function at line 6

```

As expected it does so and also reports where this return statement is located in the program.

weeder/test_return_not_enough.src

The last return statement test is for testing whether there is a reachable return statement present in a function.

```

1 #test10.src
2 func test(n: int): int
3     var n : int;
4     var a : int;
5     var b : int;
6     if (n >= 0) then {
7         if n == 2 then {
8             b = 2;
9         } else {
10             n = n + b;
11             return n;
12         }
13     }
14
15 end test
16
17 write test(1);

```

Even though a return statement is present deep inside the function, it should still report that it needs a return statement.

```

1 > Function does not contain a return at line 16

```

As expected it returns a similar error to the other return errors.

6.2 Provided tests

The provided tests are the ones given as part of the compiler project and can be found on the official website of the course[4].

By counting, it appears that the compiler passes at least 53³ of the 61 tests provided.

Some of the tests will be going through here, as these are testing some of the functionality that has been implemented as extensions.

runtime tests

The runtime tests will check, that first of all the runtime check works, but also that it returns the correct value. Lastly a short look at some assembly code will be done, to check that only used runtime checks are added to the program.

Array index value

The return value for an invalid array index value is 2.

Running `R_ErrOutOfBounds1.s` returns the value 2.

Division by 0

The return value for a division by zero is 3.

Running `R_ErrRuntimeDiv0.s` returns the value 3.

Positive argument for array allocation

The return for wrong array allocation value is 4.

Running `R_ErrRuntimeNegArraySize!.s` returns the value 4.

Uninitialized variable check

This is essentially a null-pointer and the return value for that should be 5.

Running `R_ErrRuntimeNullPointer.s` returns the value 5.

Out-of-memory check

The return value for an out-of-memory check is 6.

Running `R_ErrRuntimeOutOfMem.s` gets a segmentation fault and returns the value 139.

Review of runtime checks

Comparing the return values of the compiled programs and the language specification, it can be determined that 5 of the 6 runtime checks are working as they should. The out-of-memory check failing, might be a result of wrong memory footprint size calculation, which causes it to think that something is still using a valid memory location.

Looking at some of the assembly files for the runtime checks, it can be shown, that it only adds the labels for the runtime checks, that are actually being used in the program. For an example the bottom of the `R_ErrRuntimeDiv0.s` has only a "div_zero" label, whereas `R_ErrOutOfBounds1.s` has multiple runtime labels, since it checks for multiple kinds of runtime errors. These can be seen below. The first example is taken from `R_ErrRuntimeDiv0.s` and the second example is `R_ErrOutOfBounds1.s`.

```

71  :
72  pop %rax                # Register was live in function, so restoring it after CALL
73  main_end:              # End of body
74  addq $8, %rsp           # Remove space for variables and spills
75  movq %rbp, %rsp        # Restore old stack pointer

```

³Depends of the strict definition of passing.

```

76     pop %rbp                # Restore old base pointer
77     movq $0, %rax           # Return "no error" exit code
78     ret                     # Program return
79
80 div_zero:                  # Add division by zero runtime check
81     movq $1, %rax           # Interrupt code
82     movq $3, %rbx           # Set return code
83     int $0x80               # Call exit

```

```

190 main_end:                  # End of body
191     addq $16, %rsp          # Remove space for variables and spills
192     movq %rbp, %rsp         # Restore old stack pointer
193     pop %rbp                # Restore old base pointer
194     movq $0, %rax           # Return "no error" exit code
195     ret                     # Program return
196
197 array_index:                # Add array index runtime check
198     movq $1, %rax           # Interrupt code
199     movq $2, %rbx           # Set return code
200     int $0x80               # Call exit
201
202 positive_allocate:          # Add positive allocate runtime check
203     movq $1, %rax           # Interrupt code
204     movq $4, %rbx           # Set return code
205     int $0x80               # Call exit
206
207 uninit_var:                 # Add uninitialized variable runtime check
208     movq $1, %rax           # Interrupt code
209     movq $5, %rbx           # Set return code
210     int $0x80               # Call exit
211
212 out_of_mem:                 # Add out of memory runtime check
213     movq $1, %rax           # Interrupt code
214     movq $6, %rbx           # Set return code
215     int $0x80               # Call exit

```

Knapsack-problem

From the beginning the Knapsack problem was set a goal of being able to solve. Therefore this compiler also has to go through this test to prove itself. The program will not be shown here, but can be found on the official course page[4] with the name O_Knapsack.src.

The compiler is not able to compile a correctly working version of the program and returns the error 2, which means there is a problem with an array index.

```

1  $> echo $?
2  > 2

```

7 Conclusion

The Shere Khan compiler has been implemented to a point where it satisfies most of the specifications set by the assignment. There are still errors in the implementation when running several programs, but for the most part, it creates a correct program. There are also several other extensions and performance improvements that could have been implemented, but didn't make it. The final version of the compiler has been tested, and works mostly as expected, which is supported by the fact that most tests work as intended. This includes not only our own tests, but also the tests provided for the project. Ultimately the compiler is not able to create a working version of the Knapsack program, but is able to compile a working factorial program.

References

- [1] Andrew W. Appel *Modern Compiler Implementation in C*. Cambridge University Press, 1998.
- [2] Andrew W. Appel
<http://www.cs.princeton.edu/~appel/modern/c/project.html>
- [3] Shun Yan Cheung,
<http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html>
- [4] Kim Skak Larsen,
<https://imada.sdu.dk/~kslarsen/CC/ExamplePrograms/>