Bachelor Project in Compiler Construction

Kitty

May 2019

Report from group GROUPNUMBER: 9

Anton Nørgaard (antno16), Bjørn Glue Hansen (bhans09) & Thor Skjold Haagensen (thhaa16)

1 Introduction

The purpose of this project was to implement all of the phases in a compiler for the Kitty programming language. Kitty is a basic statically strongly typed language. One of its special features is support for nested functions. Aside from ensuring that the basic language is supported, additional extensions have been made for the language.

The target platform of the compiler is Linux 64 bit. The compiler does not create an executable, but an assembly file in GAS 64 bit format. It is expected that gcc is used to compile this file into an executable program.

1.1 Limitations

Most of the limitations of the compiler can be found in the type checker. Our compiler does not support decimal numbers or string data types. It also does not support any form of structural equivalence between records associated with different ids. There is also no support for applying operators to differing types e.g we cannot do operations of the form 42 + false. Other notable limitations is that when our compiler runs out of registers to allocate temporaries to, it does not spill them but aborts. Nested absolute operators requires that the bar tokens are not next to each other e.g. $|(|x-1|)-3|$ is legal, but $||x-1|-3|$ results in a parsing error.

1.2 Extensions

Overload of + : The plus operator can be used with arrays. This results in the two arrays being concatenated into one.

Modulo operator: Added % as modulo operator.

Increment and decrement: The increment and decrement operators have been added. In contrast to most other languages, they can only be used as a statement and not in expressions.

Copy statement: A statement for copying one array to another. This requires less runtime checks and uses the movsb instruction.

For loop: Added for loop statement.

Read statement: Reads integer from stdin and saves in the variable written after read. Skips all text that is not an integer, so can be used with complex files.

Square root function: A built-in square root function. Because integers is the only supported numerical types, the function returns the floor of the square root.

1.3 Implementation Status

All of the extensions have been successfully implemented, except for array concatenation using + operator. Due to a wrong id being passed to the allocate function, garbage collection does not correctly copy the array.

Our liveness does not support spilling and instead aborts when no registers are available. It is also possible for function calls to mess up the register allocation.

Int literals larger than 32 bit is not detected and results in the overflowed value. Similarly constant folding also doesn't detect 32 bit overflows that can happen from a calculation.

Garbage collection only scans the stack frame and doesn't modify live temps that are saved on the stack.

Otherwise everything functions as intended.

2 Parsing and Abstract Syntax Trees

2.1 The Grammar

Tokens inside $< T >$ means that T is a non-terminal. Bold tokens $\mathbf{T}$ means that the value of T is used in the make function. The statement rules was split into multiple rules so that the ';' token could be used as separator in the for statement.

$$
\begin{array}{rr}
< function >: & <\boldsymbol{head}><\boldsymbol{body}><\boldsymbol{tail}> \\
< head >: & func\,\boldsymbol{ID}\,(<\boldsymbol{par\_list}>):<\boldsymbol{type}> \\
< tail >: & end\,\boldsymbol{ID} \\
< type >: & \boldsymbol{ID} \\
| & int \\
| & bool \\
| & array\,of\ <\boldsymbol{type}> \\
| & record\,of\ <\boldsymbol{var\_list}> \\
< par\_list >: & <\boldsymbol{var\_list}> \\
| & \epsilon \\
< var\_list >: & \boldsymbol{ID}\ :<\boldsymbol{var\_type}> \\
| & <\boldsymbol{var\_type}> \\
< var\_type >: & \boldsymbol{ID}\ :<\boldsymbol{type}> \\
< body >: & <\boldsymbol{decl\_list}><\boldsymbol{statement\_list}> \\
< decl\_list >: & <\boldsymbol{decl\_list}><\boldsymbol{decleration}> \\
| & \epsilon
\end{array}
$$

$$
\begin{array}{rr}
<decleration>: & type\ \boldsymbol{ID}\ =\ \boldsymbol{type}\ ; \\
| & <\boldsymbol{function}> \\
| & var\ \boldsymbol{var\_list}\ ; \\
<statement\_list>: & <\boldsymbol{statement\_list}><\boldsymbol{statement\_sub}> \\
| & <\boldsymbol{statement\_sub}> \\
<statement\_sub>: & <\boldsymbol{statement\_line}>; \\
| & <\boldsymbol{statement\_compl}> \\
<statement\_line>: & return\ <\boldsymbol{exp}> \\
| & write\ <\boldsymbol{exp}> \\
| & read\ <\boldsymbol{variable}> \\
| & allocate\ <\boldsymbol{variable}> \\
| & allocate\ <\boldsymbol{variable}>\ of\ length\ <\boldsymbol{exp}> \\
| & <\boldsymbol{variable}> = <\boldsymbol{exp}> \\
| & copy\ <\boldsymbol{variable}>, <\boldsymbol{variable}> \\
| & copy\ <\boldsymbol{variable}>, <\boldsymbol{exp}>, <\boldsymbol{variable}>, <\boldsymbol{exp}>, <\boldsymbol{exp}> \\
| & <\boldsymbol{variable}>\ ++ \\
| & <\boldsymbol{variable}>\ -- \\
<statement\_compl>: & \{<\boldsymbol{statement\_list}>\} \\
| & if\ <\boldsymbol{exp}>\ then\ <\boldsymbol{statement}> \\
| & if\ <\boldsymbol{exp}>\ then\ <\boldsymbol{statement}>\ else\ <\boldsymbol{statement}> \\
| & while\ <\boldsymbol{exp}>\ do\ <\boldsymbol{statement}> \\
| & for\ <\boldsymbol{statement\_opt}>; <\boldsymbol{exp}>; \\
 & <\boldsymbol{statement\_opt}>\ do\ <\boldsymbol{statement\_sub}> \\
<statement\_opt>: & <\boldsymbol{statement\_line}> \\
| & \{<\boldsymbol{statement\_list}>\} \\
| & \epsilon \\
<variable>: & \boldsymbol{ID} \\
| & <\boldsymbol{variable}>\ [<\boldsymbol{exp}>] \\
| & <\boldsymbol{variable}>\ .\boldsymbol{ID} \\
<exp>: & <\boldsymbol{exp}>\ \boldsymbol{+}\ <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}>\ \boldsymbol{-}\ <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}>\ \boldsymbol{*}\ <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}>\ \boldsymbol{/}\ <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}> \boldsymbol{>} <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}> \boldsymbol{<} <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}> \boldsymbol{>=} <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}> \boldsymbol{<=} <\boldsymbol{exp}>
\end{array}
$$

$$
\begin{array}{rr}
| & <\boldsymbol{exp}> == <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}> \boldsymbol{!=} <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}> \ \&\& \ <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}> \ || \ <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}> \ \% \ <\boldsymbol{exp}> \\
| & - \ <\boldsymbol{term}> \\
| & <\boldsymbol{term}> \\
<term>: & <\boldsymbol{variable}> \\
| & \boldsymbol{ID} \, (<\boldsymbol{act\_list}>) \\
| & (<\boldsymbol{exp}>) \\
| & ! \ <\boldsymbol{term}> \\
| & | <\boldsymbol{exp}> | \\
| & \boldsymbol{NUM} \\
| & \boldsymbol{true} \\
| & \boldsymbol{false} \\
| & \boldsymbol{null} \\
| & \boldsymbol{sqrt} \, (<\boldsymbol{exp}>) \\
<act\_list>: & <\boldsymbol{exp\_list}> \\
| & \epsilon \\
<exp\_list>: & <\boldsymbol{exp\_list}>, <\boldsymbol{exp}> \\
| & <\boldsymbol{exp}>
\end{array}
$$

Figure 1: Grammar of Kitty.

2.2 Use of the `flex` Tool

The core purpose of the lexical analysis phase is to generate a token stream which is passed to bison. We also filter out anything which should be ignored, such as comments and illegal characters. To this end we use the tool Flex.
Flex scans a text file and matches elements with regular expressions. If an element matches such expression, a token represented by an enum value will be returned to bison.

2.2.1 Implementation of the `flex` Tool

Flex allows the program to be in different states according to some pattern being matched. The states are distinguished by the symbols $< STATE >$ enclosing a

state name as shown. In our implementation we use two states, $< INITIAL >$ and $< COMMENT >$. The program starts in the initial state and only enters the comment state when the token "(*" is matched, and returns to the initial state when the same amount of "*)" tokens are matched. These two tokens define a multi line comment and allows for nested multi line comments. Inside the comment state we ignore all other symbols except for newline, which increments the lineCount variable when matched. This variable is used for error messages and debugging information.
To finish comments, we'll quickly touch on single lined comments. A single lined comment in Kitty begins with the symbol #. We match this symbol with the following regular expression: on line 22 $< INITIAL > [\#][\wedge\backslash n][\backslash n]\{lineCount+\}$. Translated to human language this reads, on matching # ignore everything until the newline symbol where we make sure to increase lineCount.

The initial state consists of a combination of keywords, symbols, numbers and identifiers. The keywords and symbols are static. The keywords are placed before the identifier as both will match on the same string and higher placement gives higher priority. Identifiers and integers can vary and are matched with more complex regular expressions. Our regular expression for identifiers is $< INITIAL > [\_a-zAZ][\_a-zA-Z0-9]*$. This expression restricts identifiers to start with an underscore or a letter and can then be followed by a combination of underscores and alpha-numerals. Integers has the restriction of not starting with 0 unless it is exactly "0".

In addition to regular expressions for the keywords of the basic components of Kitty, we added the keywords 'for' to accommodate for loops, 'copy', 'read', 'write' and 'sqrt'. In addition the symbol '%' has also been added, which is the modulo operator similar to C.

2.3 Use of the `bison` Tool

In the bison tool, we handle shift-reduce conflicts in several ways. We specify that all binary operators, such as && , multiplication and addition, are left binding and given priorities identical to the mathematical precedence of the operators. The logical operator && binds more tightly than || as is expected of most programs. To handle the dangling-else problem, the "else" keyword has precedence over the "then" keyword, an expression such as : if a then if b then s else s2 would be evaluated as: if a then (if b then s else s2). This was more of an arbitrary choice of how to handle conflicts during parsing as there is no universal policy on how to resolve such conflicts.

```
%precedence "then"
%precedence "else"
%left "||"
%left "&&"
%nonassoc "<" ">" ">=" "<=" "==" "!="
%left "+" "-"
%left "*" "/" "%"
```

Figure 2: Precedence of operators.

When it comes to generating the syntax trees, we define a semantic action for each rule. We use the semantic actions to construct each node in the syntax tree.

For instance, the rule

```
var_type:          ID ":" type
                 { $$ = make_var_type($1, $3); }
```

defines that in the event we encounter an ID token followed by a colon token and then a type rule, we make a variable type node in the syntax tree, storing the semantic values of ID and type.

A short-hand for decrement and increment was also added. Any expression of the form variable++ or variable– was interpreted as a statement of the form variable + 1 or variable - 1 depending on the shorthand. Unary minus was also implemented in the same fashion where -variable was interpreted as 0 minus the value of that variable.

The other expansion was that a valid copy statement could be of two kinds: a simple copy that returns a copy of a variable and a copy-from a sub-section of an array to a sub-section of another array. For copy-from, we say that it is a valid copy-from statement, provided that both source and destination is contiguous, e.g from 1 to 10 and that the copied values do not go beyond the bounds of the destination array, meaning that a copy-from does not allow an array to grow to a bigger size as a result of the copy. This was mostly chosen for the sake of ease of implementation as allowing an array to grow in size from copying would require too much time to implement

2.4 Abstract Syntax Trees

To implement the abstract syntax tree and in turn help guide the discovery of syntax errors, we have a global integer called lineCount. We increment the variable as we move further down the parsing of the file. Several of the nodes in the syntax tree consist of a value, a line number for error messages and a kind which helps specifying which rule was applied. The value is an union of various types to have a single struct that support a group of rules.

The root of the tree is the "body" rule in our context free grammar, which consists of a declaration list and a statement list:

```
struct _body
{
    array_list *decl_list;
    array_list *stat_list;
};
```

Where each different possible declaration and statement type maintains an enumerate to denote what kind they are and in turn a union that holds the value for that particular kind, in order to define the components of the syntax tree.

2.5 Syntactic Sugar

Increment, decrement, unary minus and for loop are all handled through syntactic sugar. As all of these are short hands for existing code, they don't need their own kinds. By using syntactic sugar, only the parsing code has to be changed to add these extensions.

Increment and decrement are only added as statements as this simplifies the sugarcoating. We also don't have to consider pre- vs post-increment/decrement or side-effects.

Unary minus gets translated into 0 subtracted by the term. Later in the peephole optimization, this then gets translated into using the neg instruction.

The for loop gets translated into a statement list of the initial statement and while loop. The statement of the while loop is a statement list of the statement in the for loop followed by the statement after each loop. Because statement lists don't create separate scopes in Kitty, it doesn't affect the compiled code compared to a regular while loop. The pretty print might seem a bit odd though.

2.6 Weeding

Currently the compiler checks for return paths, matching id for "func id ... end id" and constant folding. The check for return paths is fairly conservative, ignoring expressions and only looking at specific statements. Both statements in the rule "if exp then statement else statement" are checked and if they both return on all paths, the if statement is regarded as also returning on all paths. Meanwhile "if" and "while" statements are never considered, even if they have constant expressions.

Checking for matching id is already done during parsing in the bison file and make_function(). From the rule "head body tail", make_function() receives the id in "func id" from head and the id in "end id". If they don't match, a global variable is set to 0, so the compiler knows it shouldn't continue.

Any expression consisting of literals is reduced during weeding, e.g. ”true || false” becomes ”true” and ”a + 6 * 5” becomes ”a + 30”. This helps reduce the size of the AST and the code generated in future phases. The weeding is done after type checking so we know that the types are compatible.

2.7 Test

The parsing was tested by checking if certain input produced errors when expected and was otherwise accepted. The structure of the abstract syntax tree was tested by making a pretty printer. The pretty printer should produce the input with added parentheses to show the precedence of expressions, functions, if, if else and while statements.

The table below shows the results of the tests. Op is an abbreviation for binary operators.

#	**Test**	**Expected Result**	**Pass**
	Parsing.sh: Boolean Precedence Tests		
1	Boolean ops are left most associative.	Inner parentheses around first op.	✓
2	&& op has higher precedence than || op.	Inner parentheses around && op.	✓
	Parsing.sh: Comparison Association Tests		
3	Comparison ops are non-associative.	Syntax error message.	✓
	Parsing.sh: Arithmetic Precedence Tests		
4	Ops are left most associative.	Inner parentheses around first op.	✓
5	/ and * have higher precedence.	Inner parentheses around second op.	✓
	Parsing.sh: Identifier Parsing Tests		
6	Accepted Identifiers.	Ouput is the same as input.	✓
7	Identifier starts with a number.	Syntax error message.	✓
	Parsing.sh: Combination Parsing Tests		
8	Precedence for all ops.	Arithmetic ops has highest precedence, then comparison ops and finally boolean ops.	✓
	Parsing.sh: Statement Parsing Tests		
9	Dangling else.	Ouput is the same as input.	✓
	./compiler <*.src files		
10	Use ./compiler on *.src files	Ouput is the same as input with added parentheses.	✓
11	E_IntLiteral.src	Error message.	✗
12	O_ConstantFolding.src	No errors	✗

Figure 3: Parsing tests

Test 11 and 12 both end up failing when involving integers larger than 32 bit. In test 11, the literal larger than 32 bit simply overflows and is never detected. In test 12 all literals are smaller than 32 bit, but the constant calculated from constant folding is larger than 32 bit. This is also not detected and results in an overflow.

3 The Symbol Table

3.1 Scope Rules

Variables, types and functions can be declared in the main scope and in a function. This allows for nested functions. Each function has its own scope, which can be accessed by either the function or a function declared within its scope. Parameters belong to the scope of the function they are declared with and can also be accessed by the nested functions. Declarations in the main scope are accessible from anywhere.

3.2 Symbol Data

The environment consists of two components, symbols and symbol tables. Each symbol represents something declared in a scope, such as variables or functions, while each symbol table is a hash table of the symbols within a scope.

A symbol is defined by an enumerate called kind to identify what union member to access, for type checking and for getting a list of symbols of the kind. Notable fields are the is_marked, which is an integer value that is set whenever it is declared to be of a type. Then, whenever a type is declared, we can look it up and see if the type it is declared to be is marked. This means there is a cyclic declaration and the compiler rejects the program.

The symbol table defines a scope of the identifier declarations. The integer depth denotes the number of scopes that came before the current. depth gets initialized with the value 0 denoting the main scope. The parent pointer points to the previous scope e.g. if we are in depth 3 parent will point to a scope of depth 2. The following figure shows the organization of the symbol table data

```c
typedef enum
{
        kind_s_type,
        kind_s_var,
        kind_s_ID,
        kind_s_function
} symbol_kind_t;

typedef union
{
        type_t *type;
        function_t *func;
} symbol_value_t;
```

```c
struct _sym_symbol {
    char *name;
    int is_marked;
    int is_parameter;
    symbol_kind_t kind;
    symbol_value_t value;
    int longest_jump;
    arg_t *arg;
};

struct _sym_table {
    hash_table_t *hash_table;
    int depth;
    symbol_table *parent;
};
```

Figure 4: Symbol table entries

3.3 The Algorithm

The symbol tables are constructed in the first 2 phases of type checking in a scope. First all identifiers in the declarations of the scope are added to the symbol table. Any type associated with the identifier is saved in a queue with the symbol of the identifier. Once all declarations have been added, all of the types in the queue are run through a check.

Symbols with ”id” types are modified by looking up the identifier in the symbol table. The symbol is then marked, so cyclic references can be detected. When either an id associated with a record or a symbol with a simple type is found, it terminates and traverses backwards. When traversing backwards, the types will be set to that of the previous symbol. If an id associated with a record was found, the type is set to a pointer of the symbol.

This way, all type nodes of kind ”id” should have been replaced by either the simple type it was referencing to or a pointer to a symbol with a record type. By using a pointer to the symbol and eliminating all ”id” types, conflicts with two declarations with the same identifier in different scopes are avoided.

#	Test	Expected Result	Pass
1	Put symbol in table.	Dump of table has the symbol.	✓
2	Get previous symbol.	The symbol is printed. && op.	✓
3	Put symbol with same identifier in child table.	Dump has symbol is both tables.	✓
4	Initialize multiple scopes from same table.	Both scopes point to the same table.	✓
5	Put symbol in child that has the same name as symbol in parent scope	Get method returns the correct scope	✓
6	Get symbol that only exist in parent scope.	Get the symbol from parent.	✓
7	Identifier starts with a number.	Syntax error message.	✓
8	Try to get symbol that is in a different child of the parent scope	Can't find the child and return error message	✓

Figure 5: Symbol table tests

3.4 Test

It is possible to print the symbol tables of the program using the command line option -t.

The symbol table functions as expected, with all tests passing.

4 Type Checking

4.1 Types

The language currently supports the following types:

Type	L Size	I Size	Description
int	4	8	Signed integer. Literals support 32 bit, but the calculations in the assembler support 64 bit.
bool	NA	8	Value of either true of false.
array of T	NA	16 + 8n	A pointer to a collection of n elements of type T.
record	NA	16 + 8n	A pointer to a complex type of n variables.
identifier	NA	NA	An alias for a type. Can be used for readability and for declaring recursive records.
symbol	NA	NA	A record associated with an identifier. Used internally in type checking.
undeclared	NA	NA	Internal type used to represent the null value.

Figure 6: Overview of types. L is the size of literals while I is the size on the heap.

The int type is only has support for 32 bit as a literal, due to some instructions such as cmp not being able to support 64 bit immediates.

Arrays and records are always stored as an 8 byte pointer in variables and array elements and it is not possible to pass them by value. On the heap they both have a 16 byte header followed by their content.

By having all types be 8 bytes, it simplifies code generation, code emit and handling of registers. On the other hand it is not as efficient in terms of storage.

4.2 Type Rules

Aside from array concatenation, there is no support for operator overloading.

In general, the compiler is very conservative with equivalence between types. ”int” can only be used with ”int” and ”bool” with ”bool”. Each identifier associated with a record is their own unique type, and thus not equivalent with any other type, even if the records would be equivalent. The idea is that if two records are equivalent, but associated with an identifier twice, then it most be because the user intended for them to be different.

Multiple variables that directly declare a record have some structural equivalence. In the declaration ”var a:record of {y:int, next:node}, b:record of {x:int, next:node}” the variable ”a” and ”b” are considered compatible. This is based on the types in the record and their order ”{<int>, <sym node:5> } == { <int>, <sym node:5> }”.

We disallow cyclic type declarations e.g.

```
type a = b;
type b = a;
```

An array of its own type is also considered a cyclic reference. An array of such a type can’t e used for anything.

```
type a = array of a;
```

By this logic a record that only references itself should also be disallowed. The reason for disallowing the previous example, is that the extra work provides no extra value for the user. Meanwhile records that can reference its own type is central for many data structures such as trees and graphs.

```
type a = record of {x:a};
```

4.3 The Algorithm

We perform breadth first on declarations in a scope to construct the symbol table for that scope. By using breadth first, it is possible to make use of any variables, types and functions in the parent scopes, even if they are declared further down in the source code. Constructing the symbol table is split in two phases. First all identifiers are put in the table and then afterwards checks are performed on references. This way declarations can be done in any order. Refer to chapter 3.3 for details.

While going through the declarations, functions are saved in a queue. After the symbol table is constructed and validated, these functions are checked, constructing and validating their symbol tables.

Finally in the fourth phase statements are validated. This is done by performing a DFS on the statement list of the function. Functions are called recursively until reaching a leaf, which is a term or variable. Some tree nodes have been updated to hold a reference to the type associated with the node. If leaf is a term, the type of the term is set based on its kind. Otherwise if the leaf is a variable, the type of the variable is set by looking up the symbol associated with the id of the variable. When going back through the recursive calls, the types of the children are checked and the type of the node is set. In some cases the type changes, e.g. at comparisons or index of an array, but in many other cases it remains unchanged.

If a previous phase fails, then any of the next phases are ignored and 0 returned back to the caller of the type checking.

4.4 Test

We have used a shell script *(compile_all.sh)* to compile several different programs that was made available by our supervisor. The outputs of these files are saved in a log file in the tests/logs folder. Some are expected to fail, but the majority are expected to compile successfully. Additionally the pretty_printer can be set to also show the type of variables, method calls and some expressions. This can be viewed by using the command line options –pt or -pw. -pt prints after type checking and includes types. -pw also includes types, but prints after weeding has been performed and so includes constant folding.

All tests run as expected. Test 5 compiles although it contains an unknown char. Unknown chars are just skipped during the lexer part, and so doesn't harm the structure of the code. As such, containing an unknown char won't result in the compiler aborting, but it is reported to the user as a warning.

5 Resource Computations

5.1 Resources

The resources of the program are the variables, parameters and intermediate results of expressions. Below figure describes where these resources are allocated.

Variables and parameters are computed in a phase before generating the intermediate code. Refer to figure 10 in code generation for an overview of the stack frame. Intermediate results are allocated in temporaries during code generation and are allocated in the liveness phase after code generation.

#	Test	Pass
	Expected Errors	
1	Assign incompatible value to variable	✓
2	Assign to id declared as type	✓
3	Pass incompatible value to function	✓
4	Pass incorrect amount of parameters	✓
5	Have unknown char in code	✓
6	Cyclic declaration of types	✓
7	Return in main scope	✓
8	Records of same structure but assigned to different id	✓
9	Return in only if	✓
	Expected Compile Successful	
10	Return in if and else	✓
11	Field in a record referencing the record	✓
12	Have 2 types with same id in different scopes	✓
13	Reference something that is declared later	✓

Figure 7: Type rule tests.

Parameters: Allocated on the stack before the call of function.

Variables: Allocated on the stack at the end of the stack frame.

Intermediate Results: Allocated in the r12-r15 and rbx registers.

Offset Table: Array in data section of the assembler code. It is a table of pointer variables in functions and records. Also stores the size of records. It is used by garbage collection when iterating the stack and heap.

Figure 8: Overview of resources and their location.

5.2 Parameter and Variable Allocation

Computing variables and parameter is done by using three functions, im_declaration_scan, im_assign_offsets and im_assign_parameters.

im_declaration_scan receives a symbol table, computes the variables using im_assign_offsets and iterates all of the functions in the table. For each of these functions, im_assign_parameters is called, receiving the function as an argument. After the parameters are assigned, im_declaration_scan is called with the function's symbol table as argument, resulting in a DFS traversal. By having im_assign_offsets at the start, only variables are computed for the main scope table and variables are computed after parameters.

im_assign_offsets gets the symbols of kind kind_sym_var from the symbol table. Each symbol that is a pointer is added to a queue and each record is also added to a queue. Because both variables and parameters share kind_sym_var, the loop continues if the symbol is a parameter. Otherwise the symbol is assigned an offset starting from -16. Because the symbols was received directly from the symbol table, they might not be computed in the order they were declared. This can hurt the readability of the assembler code, but there is otherwise no negative affect of this. Variables that are declared, but never used will not be computed and just ignored.

The pointer queue and record queue are used to build the offset table.

In the im_assign_parameters function, the offset of the parameters are assigned to the symbol of the parameter. The order of the parameters are important due to potential side-effects in the expressions getting computed when the function is called. As such the last parameter has the lowest offset of 24 and the first parameter has the highest offset. This way the expression of the first parameter is computed and then pushed, etc.

Variables and parameters that are only used in the scope they are declared use the rbp register. Otherwise the r9 register is used. r9 is dedicated for the current static link to a scope. How it is changed is further discussed in chapter 6.2.

5.3 Liveness and Temporary Allocation

Before any analysis is performed on the intermediate code, it is first translated into a different data structure. The data structure consists of a flow control graph and a linked list of statement blocks. A statement block is all of the flow control nodes associated with a statement in the Kitty program. Because all temporaries will live and die within the same statement block, they can be treated as basic blocks. This greatly simplifies the liveness analysis of the temporaries.

The flow control nodes, was intended to be used for liveness analysis of variables and parameters, but only liveness for temporaries was implemented.

Next the actual liveness analysis is performed. To calculate liveness we use roughly the same strategy as given in (Andrew W. Appel, Maia Ginsburg - Modern Compiler Implementation in C - Cambridge University Press (2004)). One row of use, def and in is represented using uint64_t as a bit array. Because statement blocks are basic blocks, we know we will only have to perform one iteration. This also means an out column is not needed, as it will always be equal to the in column of the next row.

Set operations are performed on the bit-arrays through bitwise operators, e.g. the set operation $A - B$ is translated to $A \wedge \neg B$. This makes the set operations both simple and fast to perform.

Because a row is represented by a single int64_t per column, a statement is limited to 64 temporaries. This does mean that some Kitty programs can not be compiled, but you would need a very large expression for it to be a problem. This could be fixed by creating a list of int64_t, allowing rows to expand dynamically, but slowing down the calculations.

We create the in table, which describes when temporaries are live, by starting at the bottom of the statement block. Using the equation from the algorithm, in_j is calculated as $in_j = use_j \cup (out_j - def_j)$, because it is a basic block, out_j can be replaced by

in_{j+1}. Translating to bitwise, we get the equation $in_j = use_j \vee (in_{j+1} \wedge \neg def_j)$.

After the in table has been created, another optimization is performed. In case a function call is part of the expression, any live temps before the call has to be saved. After they are saved, the registers are free to be used, but according to the liveness analysis they are still live. The optimization removes any temporary that was live at the start of the function call until after the the function has been called, but before the return value is assigned to a temporary.

With the final version of the in table it is now time to replace the temporaries with registers. For this we use a bit-array for registers being used, an array of which register a temporary is using and a bit-array of which temporaries were live in the previous iteration. Starting from the first row, if a temp is live and wasn't previously, we need to allocate a register. We start with the lowest register and move up until we find a register that is not being used. If no register is available, the compiler aborts with an error message. In case a temporary is not live, but it was live in the previous row, the register associated with it is freed.

We do not have support for spill in case we run out of registers. Because both arguments in an assembler instruction can not be an address, not all temporaries can use the memory instead of a register. We could push a register to free it, but not all of them are safe to free. Both cases would require more statistics to make the right decisions. Because of the time that remained, we decided to leave this out.

Figure 9 shows an example of the liveness on a very simple for loop. The cond instruction will jump to the label in argument 3 if argument 1 is equal to argument 2. Each square is a statement block and each instruction is a flow control node.

```
for i = 0; i < n; i++ do
    x = (x + 1) * 2;
```

5.4 Test

Not much testing has been done specifically for variable and parameter allocation. The commandline option -dl was added for printing the in table together with the intermediate code that each row is associated with. The file O_LivenessCall1.src does not work as intended due to an error in the register allocation. Because of the optimization with fcuntion calls, temporaries that are saved get re-allocated when becoming live again. This a problem because a temp $t1$ points to the same struct in memory as all other $t1's$ in that basic block. This is normally a benefit as you will not have to search for the temporary, when allocating registers, but just have to modify one pointer. In the case of the temp getting live again, it is possible that the register gets altered resulting in wrong behaviour. This happens when only the right side of an expression is a temp. The temp storing the intermediate result then gets allocated a register higher then the one no longer being used. If this temp is the saved before a function call, it gets assigned a different register when becoming live again.

This could be solved by checking if a temp is already assigned a register and then mark the register as used.

#	ins	use	def	in	used registers
1	mov 0, i	00	00	00	00000
2	while1_loop	00	00	00	00000
3	mov i, t1	00	10	00	10000
4	low n, t1	10	00	10	10000
5	cond 0, t1, while2_end	00	00	00	00000
6	mov x, t1	00	10	00	00000
7	inc t1	10	00	10	10000
8	mov t1, t2	10	01	10	10000
9	mult 2, t2	01	00	01	10000
10	mov t2, x	01	00	01	10000
11	inc i	00	00	00	00000
12	jmp while1_loop	00	00	00	00000
13	while2_end	00	00	00	00000

Figure 9: Liveness of for loop. Instructions should be considered as pseudo code of the intermediate language.

6 Code Generation

6.1 Strategy

The intermediate code the compiler generates is very close to the actual assembly itself. Each line in the text section is represented by a struct called entry. The kind of entry can be either instruction, label, comment, empty or tag. The tag kind is used during the liveness analysis for finding when a statement starts, function call starts and function call ends. Aside from conditional jumps and comparison operators being a single instruction, the only real abstraction we made from actual assembly was the usage of temporaries in place of registers. The usage of temporaries helps simplify the code generation.

The motivation for doing this was that we knew what architecture we were generating assembly for and could as a result do a more direct translation.

The compiler uses a suite of built-in functions. This reduces the amount of repeated code for some of the larger templates. An example could be on initialization of an array, the corresponding label for the built-in function that allocates space in memory will added to the intermediate code. The built-in functions is discussed further in chapter 8.

The stack frame of the main scope and functions consists of parameters, return address, old rbp, static link, global offset table start and local variables. Parameters and variables are covered in chapter 5.2. Return address is automatically added when using the call instruction. The old rbp is used for both restoring the stack when returning and to refer to the previous stack frame. The static link refers to the stack frame of the parent scope. It is received from the caller through the rax register. The global offset table start is used by garbage collection and is calculated at compile time. Below figure illustrates a stack frame and the offsets relative to rbp.

30	param 1
24	param 2
16	return address
8	old rbp
%rbp	static link
-8	offset table start
-16	local var 1
-24	local var 2

Figure 10: Stack frame of a function.

6.2 Static Link

While generating the code, the variable*jumps* is used to keep track of the current static link. At the beginning of a function, it is set to 0, meaning the current scope. When getting a symbol from the symbol table, the number of jumps before getting the variable is returned. If the variable is not only used locally and the current jumps of the static link doesn't match the jumps of the symbol, then the r9 register has to be updated. By making these checks at compile time, it is possible to avoid unnecessary updates of r9.

The register will always get updated after a label. Labels are always used with control flow, which can mess up the order the variables are used. It is possible to disable this check using the -ss command line option. Any program that has no nested functions, will still work with this option, but some programs with nested functions will not. The last code template in the next chapter is an example of a case where -ss would make the program not function correctly.

6.3 Code Templates

Our language supports short circuit/lazy evaluation of the || and && operator. The following expression:

```
a == 1 && b == 2
```

is translated into the following intermediate:

```
mov a, t1
eq  1, t1

cond 0, t1, and1_false
```

```
mov b, t2
eq  2, t2

cond 0, t2, and1_false

mov 1, t3
jmp and2_end

and1_false:
mov 0, t3

and2_end:
```

An array is allocated by using values computed at compile time and a built-in function. The following statement:

```
var a:array of int;
allocate a of length 10;
```

results in the following intermediate code:

```
mov 0, a

mov a, %rdi
mov 8, %rsi
mov 1, %rdx

call allocate_array
mov %rax, a
```

The first mov, sets ***a*** to null and allows garbage collection to remove any array ***a*** could be pointing to. The 8, is the size of each element. Although all types use 8 bytes currently, it is available as a parameter in case it would change in the future.

Finally an example of if and static links. The variable y is declared 1 jump from this scope, while x is declared 2 jumps away. Both are computed to offset -16.

```
y = 0;
x = 0;

if x == 0 then
    y = 20;
else
    x = 10;

return x;
```

For simplicity, the intermediate code

```
mov 1, %rdi
call get_static
mov %rax , %r9
```

will instead be shown as this.

```
call get_static 1
```

The above Kitty code results in the following intermediate code:

```
call get_static 1
mov 0, −16(%r9)

call get_static 2
mov 0, −16(%r9)

mov −16(%r9), t1
eq 0, t1

cond 0, t1, else1_start

call get_static 1
mov 20, −16(%r9)

jmp if_else1_end

else1_start:

call get_static 2
mov 10, −16(%r9)

if_else1_end:

call get_static 2
mov −16(%r9), %rax
```

6.4 Test

As touched on earlier, we made a pretty printer which can be invoked with the -pi flag. On compilation, this tool was used to validate the structure of the generated intermediate code. In conjunction with the pretty printer, we also made use of the supplied test files for static links whilst creating some ourselves. The result of the tests can be seen in the table below. Test 1 through 7 all pass however test 7 fails when we run the compiler with simplified static link assignment *(the -ss flag)*. The reason for this failure is that the variable x is declared in the main scope whilst y is declared inside the foo() function i.e in a scope one level deeper. When running with the -ss flag the static link labels get updated when following a variable. In the test we modify y after x hence changing the static link to point to y's scope, which results in us being unable to trace the link back to x's scope, making the test fail.

#	Test	Pass
1	O_StaticLinkTest	✓
2	O_StaticLinkTest2	✓
3	O_StaticLinkTest3	✓
4	O_StaticLink	✓
5	O_StaticLinkA	✓
6	O_StaticLinkB	✓
7	O_StaticLinkC	✓

Figure 11: Code Generation tests.

7 Phases before Emit

We have two separate phases before emitting the actual code, a register allocation phase and then a peephole optimizer phase. Liveness and register allocation is discussed in chapter 5.3.

7.1 Peephole

A simple Peephole optimizer has been added to the compiler. It can handle small changes e.g deleting redundant operations such as x*1, changing inefficient arithmetic operations to more efficient ones. A window of size 2 is used to identify these patterns. This enough for these cases, but a window of size 3 could have been used to help identify redundant assignments. We tried this at the current size, but it had false matches with comparison expressions.

Below is an example for subtraction involving 0.

```
mov x, t1                        mov x, t1
sub 0, t1

mov 0, t1                        sub x, t1
sub x, t1                        neg t1
```

Figure 12: Example of peephole optimization. Left is before and right is after.

The peephole optimization is done by iterating through the list of entries representing the text section. Non-code entries like such as empty lines are not skipped. The code that will be modified is usually grouped together, so it won't result in missed patterns.

Modified and unmodified entries are added to a new list. Entries are removed by then simply not adding them to the new list. After making sure the first and second entry is set, they are checked for any matches with our patterns. If a match was found, the proper action is taken, otherwise the first entry is added to the new list and set equal to the second entry, making it the next first entry. If a full iteration found no matches, function terminates. Otherwise the function is called recursively in case any new patterns emerged from our changes.

Although a terminal function wasn't used an example of one could be:
$ins + 2 * (mult + sub + add)$

8 Built-in Functions

8.1 Motivation

The motivation for making some things built-in functions, was that it made some things easier to work with and also reduced repetition of code.

8.2 Memory Management

8.2.1 Allocation And Usage

When allocating space on the heap for an array or record, it is first checked if the amount of data to be allocated fits in the current heap. In the event of failure, garbage collection is called. If garbage collection is unable to free the required space, the heap space is doubled until enough memory is available, up to a maximum of 1 gigabyte.

8.2.2 Garbage Collection

The chosen scheme for garbage collection is stop-and-copy. Reference counters can be unreliable with the occurrences of cyclic references and mark-and-sweep requires management of memory to avoid fragmentation. The compiler allows the run-time size of the program to grow up to 1 Giga-byte in size before it gives up allocating more memory and returns an out of memory error. Whenever the program finds that there is insufficient memory for allocating a record, it will call our garbage collection, afterwards checking if sufficient space has been made available. If garbage collection is unable to free enough space the heap memory will be re-sized(increased) by a factor of two. The intention is that in the long-run, the call to garbage collection will become more and more infrequent as each garbage collection run increases the available memory up to a certain point. A special case where a record with size greater than the heap space is requesting memory, the garbage collection step is skipped and the heap is increased.

8.3 Limitations

The compiler does not have the functionality needed to decrease the heap memory region.

9 Emit

9.1 Example Code

We will finally present a few small examples of the assembly code that is generated by the compiler. Note that some of the built-in assembly functions are not included in the resulting assembly file. This is because our compiler tries to not include functions that are not used in the run-time of the assembly program. To check this, the built-in functions have a struct with the instruction for calling the function. Along with this, they also have an associated flag and in the generation of the intermediate code, if any intermediate code is encountered that refers to a built-in function, its flag is set. After emitting the intermediate code, these flags are checked and the required functions are appended.

First is an example of a simple recursive factorial function:

```
func factorial(n: int): int
    if (n == 0) || (n == 1) then
        return 1;
    else
        return n * factorial(n-1);
end factorial

write factorial(5);
```

This becomes the following program, note that this is only part of the complete program. We left out built-in functions in this example, the complete program can be found under appendix A.

```
.section .data
    offset_table:
    .quad 0
    .quad 0

.global main
.section .text
main:
    # Preamble
    push %rbp
    push $0
```

```
    movq %rsp, %rbp
    push $0
    movq %rbp, %r9

    # Init memory


    # Main scope code


    # Function call start
    push %r9
    push $5
    movq %r9, %rax
    call f1_factorial
    add $8, %rsp
    pop %r9
    movq %rax, %r12
    # Function call end

    movq %r12, %rdi
    call int2string
    movq %rax, %rdi
    call write

    mov %rbp, %rsp
    pop %rbp
    ret
.type f1_factorial, @function
f1_factorial:
    # Preamble
    push %rbp
    push %rax
    movq %rsp, %rbp
    push $8
    movq %rbp, %r9

    # Function body


    movq 24(%rbp), %r12
    cmp $0, %r12
    mov $0, %r12
    sete %r12b
    cmp $1, %r12
    je or1_true
```

```
    movq 24(%rbp), %r12
    cmp  $1, %r12
    mov $0, %r12
    sete %r12b
    cmp  $1, %r12
    je or1_true
    movq $0, %r12
    jmp or2_end
or1_true:
    movq $1, %r12
or2_end:
    cmp  $0, %r12
    je else1_start

    # If statement

    movq $1, %rax
    jmp end1_factorial
    jmp if_else1_end

else1_start:


    # Function call start
    push  %r9

    movq 24(%rbp), %r12
    dec  %r12
    push  %r12
    movq $1, %rdi
    call get_static
    call f1_factorial
    add  $8, %rsp
    pop  %r9
    movq %rax, %r12
    # Function call end

    movq 24(%rbp), %r13
    imul  %r12, %r13
    movq %r13, %rax
    jmp end1_factorial
if_else1_end:
end1_factorial:
    movq %rbp, %rsp
    add  $8, %rsp
```

```
    pop  %rbp
    ret

# Compiler generated functions
...
```

The next example allocates two arrays and concatenate them using the copy statement.

```
type A = array of int;

var a:A, b:A, c:A;
var i:int, j:int;

allocate a of length 10;
allocate b of length 10;

for i = 0; i < |a|; { i++; j++; } do
    a[i] = j;

for i = 0; i < |b|; { i++; j++; } do
    b[i] = j;

allocate c of length |a| + |b|;
copy a, c;
copy b, 0, c, |a|, |b|;

for i = 0; i < |c|; i++ do
    write c[i];
```

This time everything related to memory has been generated. This is in built-in functions and can be seen in the full version in appendix B.

```
.section .data
    offset_table:
    .quad 3
    .quad -16
    .quad -40
    .quad -48

.global main
.section .text
main:
    # Preamble
    push  %rbp
    push  $0
    movq %rsp, %rbp
```

```
    push  $0
    movq %rbp, %r9
    sub  $40, %rsp

    # Assigning default values
    movq $0, -16(%rbp)
    movq $0, -24(%rbp)
    movq $0, -32(%rbp)
    movq $0, -40(%rbp)
    movq $0, -48(%rbp)

    # Init memory
    call meminit

    # Main scope code

    movq $0, -40(%rbp)
    movq $10, %rdi
    movq $8, %rsi
    movq $1, %rdx
    call allocate_array
    movq %rax, -40(%rbp)

    movq $0, -48(%rbp)
    movq $10, %rdi
    movq $8, %rsi
    movq $1, %rdx
    call allocate_array
    movq %rax, -48(%rbp)
    movq $0, -24(%rbp)

while1_loop:
    # while start
    cmp  $0, -40(%rbp)
    je address_null_err
    movq -40(%rbp), %r12

    movq -24(%rbp), %r13
    cmp  8(%r12), %r13
    mov $0, %r13
    setl %r13b
    cmp  $0, %r13
    je while2_end
    # while body
    movq -40(%rbp), %r12
    movq -24(%rbp), %r13
```

```
    movq %r12, %rdi
    movq %r13, %rsi
    call array_index
    movq -32(%rbp), %r14
    movq %r14, 16(%r12, %r13, 8)

    movq -24(%rbp), %r12
    inc %r12
    movq %r12, -24(%rbp)

    movq -32(%rbp), %r12
    inc %r12
    movq %r12, -32(%rbp)
    jmp while1_loop
while2_end:
    movq $0, -24(%rbp)

while3_loop:
    # while start
    cmp $0, -48(%rbp)
    je address_null_err
    movq -48(%rbp), %r12

    movq -24(%rbp), %r13
    cmp 8(%r12), %r13
    mov $0, %r13
    setl %r13b
    cmp $0, %r13
    je while4_end
    # while body
    movq -48(%rbp), %r12
    movq -24(%rbp), %r13
    movq %r12, %rdi
    movq %r13, %rsi
    call array_index
    movq -32(%rbp), %r14
    movq %r14, 16(%r12, %r13, 8)

    movq -24(%rbp), %r12
    inc %r12
    movq %r12, -24(%rbp)

    movq -32(%rbp), %r12
    inc %r12
    movq %r12, -32(%rbp)
    jmp while3_loop
```

```
while4_end:

    cmp  $0, -40(%rbp)
    je address_null_err
    movq -40(%rbp), %r12
    cmp  $0, -48(%rbp)
    je address_null_err
    movq -48(%rbp), %r13

    movq 8(%r12), %r12
    add  8(%r13), %r12
    movq $0, -16(%rbp)
    movq %r12, %rdi
    movq $8, %rsi
    movq $1, %rdx
    call allocate_array
    movq %rax, -16(%rbp)
    # Start copy
    movq -40(%rbp), %r12
    cmp  $0, %r12
    je address_null_err

    # Check valid to copy
    movq -16(%rbp), %rdi
    movq 8(%r12), %rsi
    dec  %rsi
    call array_index

    # Perform copy
    add  $16, %rdi
    movq %rsi, %rdx
    inc  %rdx
    imul  $8, %rdx
    movq %r12, %rsi
    add  $16, %rsi
    call memcopy
    # Copy done
    # Start copy
    movq -48(%rbp), %rsi
    movq $0, %r8
    movq -16(%rbp), %rdi
    cmp  $0, -40(%rbp)
    je address_null_err
    movq -40(%rbp), %r12
    movq 8(%r12), %rcx
    cmp  $0, -48(%rbp)
```

```
    je address_null_err
    movq -48(%rbp), %r12
    movq 8(%r12), %rdx
    call memcopyfrom
    # Copy done
    movq $0, -24(%rbp)

while5_loop:
    # while start
    cmp  $0, -16(%rbp)
    je address_null_err
    movq -16(%rbp), %r12

    movq -24(%rbp), %r13
    cmp  8(%r12), %r13
    mov $0, %r13
    setl %r13b
    cmp  $0, %r13
    je while6_end
    # while body

    movq -16(%rbp), %r12
    movq -24(%rbp), %r13
    movq %r12, %rdi
    movq %r13, %rsi
    call array_index

    movq 16(%r12, %r13, 8), %rdi
    call int2string
    movq %rax, %rdi
    call write

    movq -24(%rbp), %r12
    inc  %r12
    movq %r12, -24(%rbp)
    jmp while5_loop
while6_end:

    mov %rbp, %rsp
    pop %rbp
    ret
# Compiler generated functions
...
```

10 Conclusion

Aside from the discussed errors and potential problems, the compiler works as intended. It compiles most test files to correctly executable assembly programs, and it does so in a reasonable speed. Overall we see the end product as a successful implementation of a compiler for the Kitty language with extra extensions and features. It might have been better to skip some features, like peephole optimization and instead ensure some others like liveness worked better.

Signatures

Date: 23-5-2019

Anton Nørgaard

Bjørn Glue Hansen

Thor Skjold Haagensen

11 References

Andrew W. Appel, Maia Ginsburg - Modern Compiler Implementation in C - Cambridge University Press (2004)

A Factorial Code Emit

```
.section .data
    offset_table:
    .quad 0
    .quad 0

.global main
.section .text
main:
    # Preamble
    push %rbp
    push $0
    movq %rsp, %rbp
    push $0
    movq %rbp, %r9

    # Init memory


    # Main scope code


    # Function call start
    push %r9
    push $5
    movq %r9, %rax
    call f1_factorial
    add $8, %rsp
    pop %r9
    movq %rax, %r12
    # Function call end

    movq %r12, %rdi
    call int2string
    movq %rax, %rdi
    call write

    mov %rbp, %rsp
    pop %rbp
    ret
.type f1_factorial, @function
f1_factorial:
    # Preamble
    push %rbp
```

```
    push %rax
    movq %rsp, %rbp
    push $8
    movq %rbp, %r9

    # Function body


    movq 24(%rbp), %r12
    cmp $0, %r12
    mov $0, %r12
    sete %r12b
    cmp $1, %r12
    je or1_true

    movq 24(%rbp), %r12
    cmp $1, %r12
    mov $0, %r12
    sete %r12b
    cmp $1, %r12
    je or1_true
    movq $0, %r12
    jmp or2_end
or1_true:
    movq $1, %r12
or2_end:
    cmp $0, %r12
    je else1_start

    # If statement

    movq $1, %rax
    jmp end1_factorial
    jmp if_else1_end

else1_start:


    # Function call start
    push %r9

    movq 24(%rbp), %r12
    dec %r12
    push %r12
    movq $1, %rdi
    call get_static
```

```
    call f1_factorial
    add  $8, %rsp
    pop  %r9
    movq %rax, %r12
    # Function call end

    movq 24(%rbp), %r13
    imul  %r12, %r13
    movq %r13, %rax
    jmp end1_factorial
if_else1_end:
end1_factorial:
    movq %rbp, %rsp
    add  $8, %rsp
    pop  %rbp
    ret

# Compiler generated functions

# function for checking array index
# %rdi start address
# %rsi index
.type array_index @function
array_index:
    # Check address
    cmp $0, %rdi
    je address_null_err

    # check index
    cmp $0, %rsi
    jl array_index_err

    cmp 8(%rdi), %rsi
    jge array_index_err

    ret

array_index_err:
    mov $1, %rax                # sys_write
        mov $1, %rdi            # fd stdout
        lea err2out, %rsi       # string to dest index
        mov $20, %rdx           # lenght of message
        syscall
        mov $60, %rax           # sys_exit
        mov $2, %rdi            # array out of bounds
        syscall
```

```
address_null_err:
    mov $1, %rax                        # sys_write
        mov $1, %rdi                    # fd stdout
        lea err5out, %rsi               # string to dest index
        mov $20, %rdx                   # lenght of message
        syscall
        mov $60, %rax                   # sys_exit
        mov $5, %rdi                    # null pointer used
        syscall

divide_by_zero_err:
    mov $1, %rax                        # sys_write
        mov $1, %rdi                    # fd stdout
        lea err3out, %rsi               # string to dest index
        mov $15, %rdx                   # lenght of message
        syscall
        mov $60, %rax                   # sys_exit
        mov $3, %rdi                    # null pointer used
        syscall

# Get Static Link after n jumps
# %rdi number of jumps
# %rbp start static Link
# %rax
.type get_static, @function
get_static:
    mov %rbp, %rax

get_static_loop:
    mov (%rax), %rax
    dec %rdi
    jg get_static_loop          # Jump if %rdi > 0

    ret

# Convert int to string and saves in buffer
# %rdi int to convert
# %rax length of string
.type int2string, @function
int2string:
    push %r12
    push %r13
    push %r14

    # Local variables
```

```
    mov $buffer, %r12  # Current byte in buffer
    add $buffer_size, %r12
    sub $1, %r12
    mov $0, %r13                # Length of string
    mov $10, %rsi               # Constant divisor
    mov %rdi, %rax

    movb $0x0A, (%r12)          # add newline
    dec %r12
    inc %r13

    cmp $0, %rdi
    setl %r14b                  # Mark if negative
    jg is_positive
    je is_zero

    neg %rdi
    mov %rdi, %rax

is_positive:
    mov $0, %rdx

    div %rsi

    add $0x30, %rdx
    movb %dl, (%r12)
    dec %r12
    inc %r13

    cmp $0, %rax
    je int2string_end

    jmp is_positive

is_zero:
    movb $0x30, (%r12)
    dec %r12
    inc %r13

int2string_end:
    cmp $1, %r14
    jne int2string_not_negative

    movb $0x2D, (%r12)
    dec %r12
    inc %r13
```

```
int2string_not_negative:
    inc %r12
    movq %r12, string_start
    mov %r13, %rax

    pop %r14
    pop %r13
    pop %r12

    ret

# Write buffer to stdout
# %rdi size of buffer
.type write, @function
write:
    # move arg to proper registers
    mov %rdi, %rdx

    # write to stdout
    mov $1, %rax
    mov $1, %rdi
    mov string_start, %rsi
    syscall

    ret

.section .data
    string_start: .quad 0

    err1out:
        .ascii "meminit: error allocating memory\n" # length 33
    err2out:
        .ascii "array out of bounds\n"  # lenght 20
    err3out:
        .ascii "divide by zero\n"       # lenght 15
    err4out:
        .ascii "non-positive length for allocating array\n" # length 40
    err5out:
        .ascii "use of null pointer\n"  # lenght 20
    err6out:
        .ascii "memory out of bounds\n" # lenght 21

.section .bbs
    .equ buffer_size, 30
    .lcomm buffer, buffer_size
```

B ArrayExample Code Emit

```
.section .data
    offset_table:
    .quad 3
    .quad -16
    .quad -40
    .quad -48

.global main
.section .text
main:
    # Preamble
    push %rbp
    push $0
    movq %rsp, %rbp
    push $0
    movq %rbp, %r9
    sub $40, %rsp

    # Assigning default values
    movq $0, -16(%rbp)
    movq $0, -24(%rbp)
    movq $0, -32(%rbp)
    movq $0, -40(%rbp)
    movq $0, -48(%rbp)

    # Init memory
    call meminit

    # Main scope code

    movq $0, -40(%rbp)
    movq $10, %rdi
    movq $8, %rsi
    movq $1, %rdx
    call allocate_array
    movq %rax, -40(%rbp)

    movq $0, -48(%rbp)
    movq $10, %rdi
    movq $8, %rsi
    movq $1, %rdx
    call allocate_array
    movq %rax, -48(%rbp)
```

```
    movq $0, -24(%rbp)

while1_loop:
    # while start
    cmp  $0, -40(%rbp)
    je address_null_err
    movq -40(%rbp), %r12

    movq -24(%rbp), %r13
    cmp  8(%r12), %r13
    mov $0, %r13
    setl %r13b
    cmp  $0, %r13
    je while2_end
    # while body
    movq -40(%rbp), %r12
    movq -24(%rbp), %r13
    movq %r12, %rdi
    movq %r13, %rsi
    call array_index
    movq -32(%rbp), %r14
    movq %r14, 16(%r12, %r13, 8)

    movq -24(%rbp), %r12
    inc  %r12
    movq %r12, -24(%rbp)

    movq -32(%rbp), %r12
    inc  %r12
    movq %r12, -32(%rbp)
    jmp while1_loop
while2_end:
    movq $0, -24(%rbp)

while3_loop:
    # while start
    cmp  $0, -48(%rbp)
    je address_null_err
    movq -48(%rbp), %r12

    movq -24(%rbp), %r13
    cmp  8(%r12), %r13
    mov $0, %r13
    setl %r13b
    cmp  $0, %r13
    je while4_end
```

```
    # while body
    movq -48(%rbp), %r12
    movq -24(%rbp), %r13
    movq %r12, %rdi
    movq %r13, %rsi
    call array_index
    movq -32(%rbp), %r14
    movq %r14, 16(%r12, %r13, 8)

    movq -24(%rbp), %r12
    inc  %r12
    movq %r12, -24(%rbp)

    movq -32(%rbp), %r12
    inc  %r12
    movq %r12, -32(%rbp)
    jmp while3_loop
while4_end:

    cmp  $0, -40(%rbp)
    je address_null_err
    movq -40(%rbp), %r12
    cmp  $0, -48(%rbp)
    je address_null_err
    movq -48(%rbp), %r13

    movq 8(%r12), %r12
    add  8(%r13), %r12
    movq $0, -16(%rbp)
    movq %r12, %rdi
    movq $8, %rsi
    movq $1, %rdx
    call allocate_array
    movq %rax, -16(%rbp)
    # Start copy
    movq -40(%rbp), %r12
    cmp  $0, %r12
    je address_null_err

    # Check valid to copy
    movq -16(%rbp), %rdi
    movq 8(%r12), %rsi
    dec  %rsi
    call array_index

    # Perform copy
```

```
    add  $16, %rdi
    movq %rsi, %rdx
    inc  %rdx
    imul  $8, %rdx
    movq %r12, %rsi
    add  $16, %rsi
    call memcopy
    # Copy done
    # Start copy
    movq -48(%rbp), %rsi
    movq $0, %r8
    movq -16(%rbp), %rdi
    cmp  $0, -40(%rbp)
    je address_null_err
    movq -40(%rbp), %r12
    movq 8(%r12), %rcx
    cmp  $0, -48(%rbp)
    je address_null_err
    movq -48(%rbp), %r12
    movq 8(%r12), %rdx
    call memcopyfrom
    # Copy done
    movq $0, -24(%rbp)

while5_loop:
    # while start
    cmp  $0, -16(%rbp)
    je address_null_err
    movq -16(%rbp), %r12

    movq -24(%rbp), %r13
    cmp  8(%r12), %r13
    mov $0, %r13
    setl %r13b
    cmp  $0, %r13
    je while6_end
    # while body

    movq -16(%rbp), %r12
    movq -24(%rbp), %r13
    movq %r12, %rdi
    movq %r13, %rsi
    call array_index

    movq 16(%r12, %r13, 8), %rdi
    call int2string
```

```
    movq %rax, %rdi
    call write

    movq -24(%rbp), %r12
    inc %r12
    movq %r12, -24(%rbp)
    jmp while5_loop
while6_end:

    mov %rbp, %rsp
    pop %rbp
    ret
# Compiler generated functions

# function initializes the heap memory region
.type meminit @function
meminit:
    mov $12, %rax              # sys_brk
    mov $0, %rdi               # get start address
    syscall

    mov %rax, %rdi
    mov %rdi, heap_start
    mov %rdi, heap_currpos
    mov %rdi, lowspace

    add data_size, %rdi
    mov %rdi, highspace

    mov $12, %rax              # sys_brk
    add data_size, %rdi        # allocate heap
    syscall

        cmp %rdi, %rax                     # if not equal then error getting me
        jne meminit_err

    # Make sure new memory is zero
    mov $0, %rax
    mov heap_start, %rdi
    mov data_size, %rsi
    shr $3, %rsi
    call memstore

        ret

meminit_err:
```

```
        mov $1, %rax                    # sys_write
        mov $1, %rdi                    # fd stdout
        lea err1out, %rsi               # string to dest index
        mov $33, %rdx                   # lenght of message
        syscall
        mov $60, %rax                   # sys_exit
        mov $6, %rdi                    # out-of-memory err code 6
        syscall

#function that checks if requested bytes of heap space can be aquired.
#reqeusted space must be passed in %rdi
#uses data_size and heap_currpos
#returns 1(true) or 0(false) in %rax if there is enough/not enough space

.type memcheck @function
memcheck:
        mov data_size, %r8
    add heap_start, %r8
        sub heap_currpos, %r8       # subtracting current position in the heap

    xor %rax, %rax
        cmp %rdi, %r8                   # with the total size to get remaini
        setge %al                       # set rax (1 byte reg) to 1 if enoug

        ret

# function for expanding heap space
# We expand heap space by a factor 2 up to a limit of 1gb
# size of memory in rdi
.type memexpand @function
memexpand:
    push %rbx

    mov heap_start, %rbx
    cmp lowspace, %rbx
    je memexpand_resize

    # move data to low-space
    push %rdi
    call garbagecollection
    pop %rdi
    mov heap_start, %rbx

memexpand_resize:
    push %r12
        movq data_size, %r12
```

```
memexpand_loop:
        shl $1, %r12                            # calculating new size
        cmp data_limit, %r12            # comparing new size with upper limit
        jg memexpand_limit_err

    cmp %r12, %rdi              # keep expanding if we need more space
    jg memexpand_loop

    # new stuff
    mov %r12, %rdi
    shl $1, %rdi
    add %rbx, %rdi              # new brk address

    mov $12, %rax               # sys_brk
    syscall

    cmp %rax, %rdi              # if not equal we could not expand heap
    jne memexpand_err

    # Make sure new memory is zero
    mov $0, %rax
    mov highspace, %rdi
    mov %r12, %rsi
    sub data_size, %rsi
    shr $3, %rsi
    call memstore

    movq %r12, data_size
    add %rbx, %r12
    mov %r12, highspace

    pop %r12
    pop %rbx

        ret

memexpand_err:
        mov $1, %rax                    # sys_write
        mov $1, %rdi                    # fd stdout
        lea err6out, %rsi               # string to dest index
        mov $21, %rdx                   # lenght of message
        syscall
        mov $60, %rax                   # sys_exit
        mov $6, %rdi                    # out-of-memory err code 6
        syscall
```

```
memexpand_limit_err:
        mov $1, %rax                        # sys_write
        mov $1, %rdi                        # fd stdout
        lea errlimit, %rsi                  # string to dest index
        mov $20, %rdx                       # lenght of message
        syscall
        mov $60, %rax                       # sys_exit
        mov $6, %rdi                        # out-of-memory err code 6
        syscall

# Function for removing garbage in heap
.type garbagecollection @function
garbagecollection:

    mov lowspace, %rsi
    mov %rsi, %rdi
    mov highspace, %rax

    cmp heap_start, %rax            # if low-space will be to-space, swap
    cmove %rax, %rdi
    cmove %rsi, %rax

    push %rdi

    # set to-space
    movq %rax, heap_start
    movq %rax, heap_currpos

    # move from-space -> to-space
    call iterate_stack              # add from stack
    call iterate_heap               # scan to-space
    pop %rdi                        # get old start address
    mov data_size, %rsi
    call iterate_temps              # change temporary results

    # make sure unused space is 0
    mov $0, %rax
    mov heap_currpos, %rdi
    mov heap_start, %rsi
    add data_size, %rsi
    sub %rdi, %rsi
    shr $3, %rsi
    call memstore

    ret
```

```

# function for allocating memory on the "heap"
# size of memory requested be passed in %rdi
# this is needed to store the metadata
# returns adress to start of memory in %rax

.type memalloc @function
memalloc:
        push %rdi                                               # pushing rdi to

    cmp data_size, %rdi             # check if impossible to
    jg memalloc_expand              # fit in current space

        call memcheck                           # Calling memcheck with argument
                                                                # return

        cmp $1, %rax                                    # comparing result from
        je memalloc_finalize

        # lav garbage collection hvis fejler for g memory her
    call garbagecollection

    movq (%rsp), %rdi
    call memcheck

    cmp $1, %rax
    je memalloc_finalize

memalloc_expand:
    # minimum required size of new heap
    movq heap_currpos, %rdi
    subq heap_start, %rdi           # current bytes being used
    addq (%rsp), %rdi               # min bytes needed
        call memexpand                          # will exit program on failure

memalloc_finalize:
        pop %rdi                                                # popping alloca
        movq heap_currpos, %rax                 # start of allocated memory, thi
        add %rdi, heap_currpos                  # adding allocated size

        ret

# Iterate all pointer variables on the stack
# Local
# %rdi Static link of current frame
# %r12 Content of variable on stack
```

```
# %r13 Address of offset_table
# %r14 Count in offset_table
.type iterate_stack , @function
iterate_stack:
    push %rbp
    push %r12
    push %r13
    push %r14
    push %r15

    mov %rbp , %rdi

get_meta:
    lea offset_table(%rip), %r13     # address of offset_table
    addq -8(%rdi), %r13              # address of functions info

    movq (%r13), %r14                # get number of variables

next_var:
    dec %r14
    jl previous_frame                # jump if r14 is lower than 0

    add $8, %r13                     # increment offset_table pointer
    movq (%r13), %r15                # save offset in r15
    lea (%rdi, %r15, 1), %r12        # get address of the next variable

    push %rdi

    movq (%r12), %rdi                # address to heap as first argument
    call memfromptr                  # call function for variable
    movq %rax, (%r12)                # replace content with new address to heap

    pop %rdi

    jmp next_var

previous_frame:
    cmp $0, (%rdi)
    cmovne 8(%rdi), %rdi
    jne get_meta

    # epilogue
    pop %r15
    pop %r14
    pop %r13
    pop %r12
```

```
    pop %rbp
    ret

# Iterate all pointers on the heap
# Local
# %r12 scan, current address in heap
# %r13 counter for loops
# %r14 address to offset_table
.type iterate_heap, @function
iterate_heap:
    push %r12
    push %r13
    push %r14
    movq heap_start, %r12

next_memory:
    cmp heap_currpos, %r12
    je iterate_heap_end

    movq (%r12), %r13

    cmp $2, %r13
    je iterate_array

    cmp $3, %r13
    je iterate_record

    # cmp $0, %r13              # should never happen
    # je iterate_heap_end

    # array of non-pointer values
    movq 8(%r12), %r13
    imul $8, %r13
    add $a_header, %r12         # add header to current address
    add %r13, %r12              # add array size to current address
    jmp next_memory

iterate_array:
    movq 8(%r12), %r13
    add $a_header, %r12

iterate_array_loop:
    dec %r13
    jl next_memory

    movq (%r12), %rdi
```

```

    # call function for variable
    call memfromptr

    movq %rax, (%r12)

    add $8, %r12
    jmp iterate_array_loop

iterate_record:
    movq 8(%r12), %r14
    movq 8(%r14), %r13          # count in offset_table

iterate_record_loop:
    dec %r13
    jl iterate_record_end

    mov %r12, %rdi
    addq 16(%r14, %r13, 8), %rdi
    push %rdi

    mov (%rdi), %rdi

    # call function for variable pointer
    call memfromptr

    pop %rdi
    movq %rax, (%rdi)

    jmp iterate_record_loop

iterate_record_end:
    addq (%r14), %r12
    add $r_header, %r12

    jmp next_memory

iterate_heap_end:
    pop %r14
    pop %r13
    pop %r12
    ret

# Iterate temporary result registers
# %rdi old from-space start
# %rsi old size
```

```
.type iterate_temps, @function
iterate_temps:
    push %rbx
    push %r8

    # redirect %r12
    xor %rbx, %rbx

    mov %r12, %r8
    sub %rdi, %r8
    jl iterate_temps_r13

    sub %rsi, %r8
    jge iterate_temps_r13

    movq (%r12), %r12

iterate_temps_r13:
    # redirect %r13
    xor %rbx, %rbx

    mov %r13, %r8
    sub %rdi, %r8
    jl iterate_temps_r14

    subq %rsi, %r8
    jge iterate_temps_r14

    movq (%r13), %r13

iterate_temps_r14:
    # redirect %r14
    xor %rbx, %rbx

    mov %r14, %r8
    sub %rdi, %r8
    jl iterate_temps_r15

    subq %rsi, %r8
    jge iterate_temps_r15

    movq (%r14), %r14

iterate_temps_r15:
    # redirect %r15
    xor %rbx, %rbx
```

```
    mov %r15, %r8
    sub %rdi, %r8
    jl iterate_temps_end

    subq %rsi, %r8
    jge iterate_temps_end

    movq (%r15), %r15

iterate_temps_end:
    pop %r8
    pop %rbx
    ret

# Function for pointer on from-space
# %rdi address in from-space
# %rax new address on to-space
.type memfromptr, @function
memfromptr:
    push %r12
    push %r13

    xor %rax, %rax
    cmp $0, %rdi
    je memfromptr_end

    mov %rdi, %r13
    movq (%r13), %r12              # first value in header

    # Check if header is an address
    cmp $3, %r12
    cmovg %r12, %rax
    jg memfromptr_end              # header is not an id, so must be an address

    call memsize
    push %rax

    # copy from-space to to-space
    movq heap_currpos, %rdi        # dst address
    mov %r13, %rsi                 # src address
    mov %rax, %rdx                 # bytes to copy
    call memcopy

    # update header on from-space
    movq heap_currpos, %rax
```

```
    movq %rax, (%r13)

    # update next in to-space
    pop %r12
    addq %r12, heap_currpos

memfromptr_end:
    pop %r13
    pop %r12

    ret

# function for getting size of an entry on heap
# %rdi address of entry
# %rax size in bytes
.type memsize, @function
memsize:
    push %r12
    movq (%rdi), %r12

    xor %rax, %rax
    cmp $0, %r12
    je memsize_end

    movq 8(%rdi), %rax

    cmp $3, %r12
    je memsize_record

    # else array
    imul $8, %rax
    add $a_header, %rax
    jmp memsize_end

memsize_record:
    movq (%rax), %rax
    add $r_header, %rax

memsize_end:
    pop %r12
    ret

# Effective function for copying bytes of any size
# %rdi dst address
# %rsi src address
# %rdx bytes to copy
```

```
.type memcopy @function
memcopy:
    mov %rdx, %rcx
    # shr $3, %rcx
    cld
    rep movsb

    ret

# Effective function for copying constants to array
# %rax constant
# %rdi start address
# %rsi element count
.type memstore @function
memstore:
    mov %rsi, %rcx
    cld
    rep stosq

    ret

# Copy from index of one array to index of another array
# %rdi dst start of array
# %rsi src start of array
# %rdx number of iterations
# %rcx dst start index
# %r8  src start index
.type memcopyfrom @function
memcopyfrom:
    push %r12
    push %r13

    mov %rdi, %r12
    mov %rsi, %r13

    # check dst interval
    mov %rcx, %rsi
    call array_index

    add %rdx, %rsi
    dec %rsi
    call array_index

    # check src interval
    mov %r13, %rdi
    mov %r8, %rsi
```

```
    call array_index

    add %rdx, %rsi
    dec %rsi
    call array_index

    # set parameters for copying
    mov %r12, %rdi
    add $a_header, %rdi
    imul $8, %rcx
    add %rcx, %rdi

    mov %r13, %rsi
    add $a_header, %rsi
    imul $8, %r8
    add %r8, %rsi

    imul $8, %rdx
    call memcopy

    pop %r13
    pop %r12

    ret

# function for allocating array
# %rdi elements
# %rsi element size
# %rdx array type, 2 = pointers, 1 otherwise
# %rax return start address of array
.type allocate_array @function
allocate_array:

    cmp $0, %rdi
    jle allocate_array_err

    # allocate space
    push %rdi
    push %rdx
    imul $8, %rdi           # bytes for array
    add $a_header, %rdi     # extra space for metadata
    call memalloc
    pop %rdx
    pop %rdi

    # add metadata
```

```
    movq %rdx, (%rax)        # array type
    movq %rdi, 8(%rax)       # size of array

    ret

allocate_array_err:
    mov $1, %rax                        # sys_write
        mov $1, %rdi                    # fd stdout
        lea err4out, %rsi               # string to dest index
        mov $40, %rdx                   # lenght of message
        syscall
        mov $60, %rax                   # sys_exit
        mov $4, %rdi                    # array out of bounds
        syscall

# function for checking array index
# %rdi start address
# %rsi index
.type array_index @function
array_index:
    # Check address
    cmp $0, %rdi
    je address_null_err

    # check index
    cmp $0, %rsi
    jl array_index_err

    cmp 8(%rdi), %rsi
    jge array_index_err

    ret

array_index_err:
    mov $1, %rax                        # sys_write
        mov $1, %rdi                    # fd stdout
        lea err2out, %rsi               # string to dest index
        mov $20, %rdx                   # lenght of message
        syscall
        mov $60, %rax                   # sys_exit
        mov $2, %rdi                    # array out of bounds
        syscall

address_null_err:
    mov $1, %rax                        # sys_write
        mov $1, %rdi                    # fd stdout
```

```
        lea err5out, %rsi               # string to dest index
        mov $20, %rdx                   # lenght of message
        syscall
        mov $60, %rax                   # sys_exit
        mov $5, %rdi                    # null pointer used
        syscall

divide_by_zero_err:
    mov $1, %rax                        # sys_write
        mov $1, %rdi                    # fd stdout
        lea err3out, %rsi               # string to dest index
        mov $15, %rdx                   # lenght of message
        syscall
        mov $60, %rax                   # sys_exit
        mov $3, %rdi                    # null pointer used
        syscall

# Convert int to string and saves in buffer
# %rdi int to convert
# %rax length of string
.type int2string, @function
int2string:
    push %r12
    push %r13
    push %r14

    # Local variables
    mov $buffer, %r12  # Current byte in buffer
    add $buffer_size, %r12
    sub $1, %r12
    mov $0, %r13                # Length of string
    mov $10, %rsi               # Constant divisor
    mov %rdi, %rax

    movb $0x0A, (%r12)          # add newline
    dec %r12
    inc %r13

    cmp $0, %rdi
    setl %r14b                  # Mark if negative
    jg is_positive
    je is_zero

    neg %rdi
    mov %rdi, %rax

```

```
is_positive:
    mov $0, %rdx

    div %rsi

    add $0x30, %rdx
    movb %dl, (%r12)
    dec %r12
    inc %r13

    cmp $0, %rax
    je int2string_end

    jmp is_positive

is_zero:
    movb $0x30, (%r12)
    dec %r12
    inc %r13

int2string_end:
    cmp $1, %r14
    jne int2string_not_negative

    movb $0x2D, (%r12)
    dec %r12
    inc %r13

int2string_not_negative:
    inc %r12
    movq %r12, string_start
    mov %r13, %rax

    pop %r14
    pop %r13
    pop %r12

    ret

# Write buffer to stdout
# %rdi size of buffer
.type write, @function
write:
    # move arg to proper registers
    mov %rdi, %rdx

```

```
    # write to stdout
    mov $1, %rax
    mov $1, %rdi
    mov string_start, %rsi
    syscall

    ret

.section .data
    string_start: .quad 0
    heap_start: .quad 0
        heap_currpos: .quad 0
    lowspace: .quad 0
    highspace: .quad 0
    # Limit of 500 MB
    data_limit: .quad 0x20000000
        # Initial size of 4096 bytes,
    # the usual size of a virtual memory page
    data_size: .quad 0x1000

    err1out:
        .ascii "meminit: error allocating memory\n" # length 33
    err2out:
        .ascii "array out of bounds\n"  # lenght 20
    err3out:
        .ascii "divide by zero\n"       # lenght 15
    err4out:
        .ascii "non-positive length for allocating array\n" # length 40
    err5out:
        .ascii "use of null pointer\n"  # lenght 20
    err6out:
        .ascii "memory out of bounds\n" # lenght 21
    errlimit:
        .ascii "1 GB limit exceeded\n"  # lenght 20

.section .bbs
    .equ buffer_size, 30
    .lcomm buffer, buffer_size
    .equ r_header, 16
    .equ a_header, 16
```