Bachelor Project in Compiler Construction

Falcon

May 2019

Report from group 3:

Gabriel Howard Jadderson : gajad16 : #438023
Jens Kofoed Laurberg : jlaur16 : #439403
Siver Sabah AL-khayat : sialk16 : #439492

1 Introduction

In this report in which we explore the implementation of a compiler. Compilers encompass a vast area of computer science from data structures, graphs, to programming.

We introduce the report for our compiler named `Falcon`. Our compiler is able to compile falcon inputs into X86 Assembly/GAS code. This report describes the set of phase that our compiler encompasses. The phases consist of a parsing phase, including a lexicographical analysis, type checking and finally code generation. During these phases, an abstract syntax tree is built from the user input that is to be compiled. The abstract syntax tree is filled with valuable information during the different phases and finally converted into Assembly code.

1.1 Clarifications

1.2 Limitations

The obvious limitations in our compiler is the things that have not been implemented yet. Such as records and some of the array functionality. Beyond that we discovered some extensive error in the compiler. For example the compiler analyses the code in a sequential manner which can give issues when a variable is declared after the scope it should be able to be used within.
There is also an issue with end labels for a specific case of nested functions which is explained further in the report.

1.3 Extensions

1.3.1 Syntactic Sugar

Post increment of variables has been added, together with operator assignments such as + =.

1.3.2 Strings

Strings are an important part of any programming language. Therefore, we have chosen to extend our grammar to handle strings, similarly to how the Java programming language does.

Under the hood, a string structure contains an array of characters, together with the length of the string. Strings are resolved during compile time, along with their corresponding lengths. In order to achieve this, we have introduced a new terminal token `'string'`, which is added to the `<type>` rules, to introduce a new string type. The scanning for string literals is done in flex, by creating a

char buffer and populating it for each string. A token is then returned with the pointer to the buffer.

The full description of the string grammar is in section 2.1.6

1.3.2.1 String Concatenation

String literal concatenation is evaluated when parsing the abstract syntax tree (AST) during compile time. This means that we can only do string concatenation of string literals for now, and that concatenation of dynamically created strings is not implemented.

The following is a valid example:

Listing 1: Valid program of literal strings and literal string concatenation.

```
1  var s : string;
2  s = "Compiler Says:";
3  write s;
4  write  "Hello, " + "World.";
```

Listing 2: Invalid program of dynamic string concatenation.

```
1  var s : string;
2  var s1 : string;
3  s = "Hello, ";
4  s1 = "World.";
5  write s + s1;
```

1.3.3 Peephole optimization

Our compiler includes a Simple implementation of a peephole optimizer. Our peephole optimizer can reduce four computationally expensive memory instruction into two. Since our compiler does not generate an intermediate representation, we are limited to parsing our own emit code and analyzing it. This may cause problems in the future, and certainly limits our peephole-optimization capabilities.

We had to halt the development of the peephole-optimizer due to time constraints. However, we have managed to reduce make a working example.

1.4 Implementation Status

1.4.1 Strings

Strings have been tested to work, but some structural changes of the emit code must be handled properly.

1.4.2 Peephole-optimizer

The peephole optimizer functions correctly, however, further development on intermediate representation is highly essential before the peephole-optimizer can can be used to a greater extent.
The current implementation of the peephole-optimizer works in some cases, however due to the changes in the emit code, the peephole optimizer is substantially limited.

1.4.3 Parser

1.4.4 Type Checker

Everything Works as expected, however, further development on Booleans, records, and strings is required.

1.4.5 Code Generation

The code generator can generate code for basic programs, run recursive programs, and assign variables from other scopes through the static link. Declarations of the language are implemented except for records. Expressions, statements and terms of the language are implemented. It all works on the stack without the use of temporaries. Thus we do not get the chance to implement liveness analysis, control flow graphs or register-allocation algorithms such as colouring by simplification. Since temporaries are not used, the stack will contain all the information that will be stored over the long term and short term. The only place where registers are used is the case when an expression tree is calculated, but this will only be for a short isolated period where the need for caller-save or callee-save conventions is not needed. This is explained further in section 5
There is a couple of issues. First of all, there is a case where the function end label is slightly the wrong string. This is explained in the section 5.
Another important notable issue is the fact that the compiler evaluates the source code sequentially, thus when an expression within a scope uses a variable from the parent scope that is defined after the function definition, the information about the variable is not set, such as its offset to its frame-pointer. This results in the error that the offset of the variable is not known thus since it is initialized by 0 as default, there will be a reference to the wrong variable. This can be fixed by evaluating all declarations that are not function declaration first and then following that. The function declarations should be evaluated within all ”body” nodes of the AST.

2 Parsing and Abstract Syntax Trees

2.1 The Grammar

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

Figure 1: Falcon full Grammar, part 1.

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

Figure 2: Falcon full Grammar, part 2.

To depict legal inputs for the compiler, we use a context free grammar. The complete grammar is seen in both Figure 1 and Figure 2.

The grammar consists of a set of rules, called productions. The left-hand side of each production is a variable, and can be expanded into the right-hand side. For a string to be in the language of acceptable inputs for the compiler, i.e. being a syntax-error-free 'kitty' program, it must be possible to derive the start symbol from the input. This is done by replacing the right-hand side of one or more productions, with the left-hand side, until only the start symbol remains.

2.1.1 Body

The `<body>` node is the body of a function. `<body>` is furthermore the start symbol of the grammar, meaning the whole user program syntactically is inside an implicit main function. It can be expanded into a list of declarations and a list of statements, as seen in Figure 3.
In the AST, `<body>` will point to a node that represents a `<decl_list>` node and a `<statement_list>` node.
`<decl_list>` can either be empty, or a list of one or more declarations. This causes the declaration of variables, to happen in the beginning of the function, before the statements of the function.
`<statement_list>` is a list of one or more statements.

⟨body⟩ : ⟨decl_list⟩ ⟨statement_list⟩

Figure 3: Body Grammar.

2.1.2 Statement

Figure 4 illustrates productions of the non-terminal `<statement>`, containing 9 different right-hand sides.
The `<return>` symbol defines a return statement, which purpose it is to return an expression at the end of a function call.
The `write` variable is used for outputting a result of an expression.
The `allocate` statement is used for allocating memory for a record, and the purpose of `allocate <variable> of length <expression>` is to allocate memory for an array.
Following the `if <expression> then`, with the possibility of an additional `else` token and a `<statement>`, when parsing conditional statements. Here the grammar indicates that the expression after the `if` will be evaluated before jumping into the `<statement>` in either the `if` or `else`. Thus a node indicating an `if statement` in the AST, points to `<expression>` and `<statement>` nodes.
Finally the grammar for `while <expression> do <statement>` and a block of statement list `<statement_list>` is defined.

⟨statement⟩ : **return** ⟨expression⟩ **;**
| **write** ⟨expression⟩ **;**
| **allocate** ⟨variable⟩ **;**
| **allocate** ⟨variable⟩ **of length** ⟨expression⟩ **;**
| ⟨variable⟩ = ⟨expression⟩ **;**
| **if** ⟨expression⟩ **then** ⟨statement⟩
| **if** ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩
| **while** ⟨expression⟩ **do** ⟨statement⟩
| { ⟨statement_list⟩ }

Figure 4: Statement Grammar.

2.1.3 Expressions

The `<expression>` production can either derive a `<term>`, or be expanded into an `<expression> <op> <expression>` node as seen in Figure 5. In the latter case, `<op>` is a terminal, which can be replaced with one of 12 operators. The 12 operators consists of four mathematical operators (addition, subtraction, multiplication and division), two Boolean operators (and, or), and the relational operators (greater than, lesser than, greater or equal to, lesser or equal to), the equality operator (==), and the non equality operator (!=).

⟨expression⟩ : ⟨expression⟩ **op** ⟨expression⟩
| ⟨term⟩

Figure 5: Expression Grammar.

2.1.4 Terms

The grammar for the variable `<term>` is seen i Figure 6.
`!` `<term>` is used for the negation of the terms true and false.
`|<expression>|`, can denote the size of an array or the absolute value of a `num`.
`num` is a literal number, e.g. 42
`null` is the standard value for a reference variable (array and record).

⟨term⟩ : ⟨variable⟩
| ⟨string⟩
| **id** **(** ⟨act_list⟩ **)**
| **(** ⟨expression⟩ **)**
| **!** ⟨term⟩
| `|` ⟨expression⟩ `|`
| **num**
| **true**
| **false**
| **null**

Figure 6: Term Grammar.

2.1.5 Type

The `<type>` non-terminal can derive the fundamental data types of the compiler. The data types includes `<id>`, which is an identifier for a variable, `int`, `bool`, `string`, array and record which is similar to a struct.
The grammar for the types is seen in Figure 7.

⟨type⟩ : **id**
| **int**
| **bool**
| **string**
| **array of** ⟨type⟩
| **record of** { ⟨var_decl_list⟩ }

Figure 7: Type Grammar.

2.1.6 String Grammar

To allow for string support in the falcon language, a new type called 'string' is added to `<type>` and the `<string>` grammar is incorporated into `<term>`, such that a string is a term. The **string** terminal token is similar to the **id** terminal token. In fact, the only difference is the name and the scanning implementation in flex.
String concatenation achieved by overloading the + token as can be seen in Figure 8.

⟨string⟩ : **string**
| **string** + **string**

Figure 8: String Grammar.

2.2 Use of the flex Tool

The tool Flex is used during the scanning phase of our compiler. The tool Bison is used during the parsing phase. The two phases are intertwined such that Flex scans for tokens, then returns directly to bison, bison handles the input immediately after flex returns a value to bison. At this point, Bison generates the specific node in the AST, then again flex scans the next item and the process is repeated.
The tokens of the language are first defined in the Bison File, Bison then generates a header containing the definitions etc. Flex then includes the generated C header file and knows which tokens to reference to. Most of the terminal symbols of the grammar, such as 'if', 'bool' and 'false', that is their actual string values are directly formulated in Flex, so these symbols can be used as tokens by bison. Only the terminal symbols are defined in the flex file, and all of the non-terminal symbols of the grammar are handled using bison. The terminal symbols of the grammar, defined in the flex file, consists of arithmetic and logic operators, and keywords used in the language. Regular expressions are used to define the terminal symbols 'id', which is a string value, and 'num' which is an integer value. The regular expressions used to define 'id' are placed last in the

flex file since all the keywords from the language have the same features as 'id' does. 'id' is thus placed last, to capture all the possible id string values that are not reserved for keywords in the language. The regular expression that returns the 'id' string value, is designed to capture all strings that begin with a letter, and only contains letters or numbers.

2.2.1 Start conditions

The use of exclusive start conditions in Flex, gives the possibility to catch part of the input, which is syntactically different than the rest. Therefore one-line comments, multi-line comments and strings are implemented in Flex using exclusive start conditions. Being exclusive means that rules with no start condition will not be active at the same time as the rules qualified with the start conditions.

2.2.1.1 Comments in general

Comments are important for the practicality of the language. They can be found in almost every popular language, and they're beneficial for storing rich information directly within the source code. Falcon supports two types of comments, namely one-line comments and multi-line comments. Comments are handled during the scanning phase in flex, and are completely ignored by the compiler throughout the entire compilation process.

2.2.1.2 one-line comments

To declare a one-line comment, the pound sign `'#'` is used as the beginning of the one-line comment. The comment is ended by a newline character.

2.2.1.3 multi-line comments

Multi-line comments allow for documentation and for complex textual structures that are hard or cumbersome to achieve with just one-line comments. To declare a multi-line comment a `'(*'` is used as the beginning of a multi-line comment and `'*)'` closes the comment. As the name indicates, such comments may run over several lines, though it may also be closed on the same line it is started. These comments can also be nested. The implementation of comments is done during flex in which whenever the initial `'(*'` is encountered we switch to a new state, the new state ignores everything besides subsequent `'(*'` strings, once the string `'(*'` is encountered a simple counter is incremented and decremented when a matching closing `'*)'` string is encountered. This allows the comments to be nested. The state is terminated when the counter reaches zero indicating that we've matched the last `'*)'`.

2.2.1.4 String scanning

Scanning for strings starts off by matching `'"'` in the initial state. Then we enter the string state, in the string state we allocate a buffer for the string and append the characters into the buffer. The buffer can grow dynamically and is amortized, allowing for large strings to be scanned. The state is then terminated and flex is brought back to the initial state when the `'"'` is encountered within the string state. The string is returned to bison similarly to the 'id'.

2.3 Use of the bison Tool

2.3.1 Implementation of the grammar

Each variable of the grammar is implemented in Bison together with the belonging substitution rules. The different variables of the grammar (along with extensions) are: `string`, `function`, `head`, `tail`, `type`, `par_decl_list`, `var_decl_list`, `var_type`, `body`, `decl_list`, `declaration`, `statement_list`, `statement`, `variable`, `exp`, `term`, `act_list`, and `exp_list`. Bison goes through the input it gets from Flex, and tries to match it with the substitution rules. If the input string, i.e. the user program, can be reduced to the start state, which is 'body' in the grammar, using the substitution rules defined in Bison, then the string (i.e. the user program) is in the language defined by the grammar.

Tokens are defined in Bison and consists of all the terminal symbols of the grammar. All the non-terminal symbols, i.e. the variables of the grammar, are defined as types in Bison and can be seen as all the left-hand sides of the substitution rules of the grammar.

2.3.2 Using bison together with c

Each time Bison encounters a string from Flex, consisting of variables and terminals, that matches a right-hand side of a substitution rule, Bison can either reduce the string of variables and terminals to be the left-hand side of the substitution rule, or in some cases, Bison can choose to shift, meaning looking at the next token, to see if a longer right-hand side of a substitution rule can be matched. In the case when Bison chooses to reduce a set of tokens to be a variable, Bison executes a piece of c-code, which creates a node to the abstract syntax tree and pops the inspected tokens from the stack.

Each type in Bison is also defined as a 'typedef' in the header file 'tree.h'. An example is seen in 'Listing 3' which shows the 'typedef' of 'EXP'.

Listing 3: Expression node.

```
1  typedef struct EXP
2  {
3     int lineno;
4     enum
```

```
5     {
6        exp_timesK, exp_divK, exp_plusK, exp_minusK, exp_equalityK,
7        exp_non_equalityK, exp_greaterK, exp_lesserK,
             exp_greater_equalK,
8        exp_lesser_equalK, exp_andK, exp_orK, exp_termK
9     } kind;
10    union
11    {
12       struct { struct EXP *left; struct EXP *right; } op;
13       struct TERM* term;
14    } val;
15    struct type_info* type;
16    struct code_info* cg;
17 } EXP;
```

An 'EXP' can be one of 13 different kinds as stated in line 4 through 9 of 'Listing 1'. For each different kind that the 'typedef' 'EXP' can take, a corresponding function is defined (giving 13 functions in this case), which returns a pointer to a newly allocated 'EXP' with kind correctly specified, together with possibly additional information as seen in line 10 through 14 in 'Listing 3', such as pointers to two additional 'EXP's' if the 'EXP' for example is the arithmetic binary operation 'expression + expression'. An example of one of the functions that return a pointer to an 'EXP' is seen below in 'Listing 4'.

Listing 4: Function example from tree.c

```
1  EXP* make_EXP_plus(EXP * left, EXP * right)
2  {
3      printf("made + at line: %d\n", lineno);
4      EXP* e = NEW(EXP);
5      e->lineno = lineno;
6      e->kind = exp_plusK;
7      e->val.op.left = left;
8      e->val.op.right = right;
9      return e;
10 }
```

The 'make_EXP_plus' allocates a new 'EXP', sets the kind to specify that it is a binary operator that adds two expressions together, and stores the two expressions that are to be added in an 'EXP' 'left' and an 'EXP' 'right' which are in the struct. The way that the grammar is handled in Bison, 'tree.c' and 'tree.h' is relatively similar for all the variables and substitution rules. For each kind of variable, or left-hand side of the grammar, a 'typedef' is defined which also specifies all the different kinds of right-hand sides the variable can take. Furthermore for all the different kinds of right-hand sides of the grammar, a function is specified in 'tree.h' and defined in 'tree.c'. An example of how these 'typedefs' and functions are used by Bison is given in 'Listing 5' below.

Listing 5: Bison expressions example.

```
1  exp : exp '*' exp
2      {
3          $$ = make_EXP_times($1,$3);
4      }
```

```
5       | exp '/' exp
6       {
7           $$ = make_EXP_div($1,$3);
8       }
9       | exp '+' exp
10      {
11          $$ = make_EXP_plus($1,$3);
12      }
```

We see in the example 'Listing 5' in line 9 through 12, that when Bison encounters a string consisting of a type of 'exp', a '+' token, and then another 'exp' type, that Bison can choose to reduce, and thus call the 'make_EXP_plus' function, and in the process give the correct 'exp's' as arguments to the function, by popping the left and right expression of the operator from the stack. The procedure executed by Bison is similar for all the right-hand sides of the grammar, only the tokens and types looked at are different, along with which function Bison calls, and which tokens, if any, from the stack that should be given as arguments to the function.

2.3.3 Shift/reduce conflicts in bison

Some of the right-hand sides of the substitution rules from the grammar are proper subsets of another substitution rule from the grammar. When Bison encounters such a subset, it can choose to reduce the string under analysis to a variable, or shift and look ahead, and possibly reduce an even greater string. This choice leads to a shift/reduce conflict in Bison.

A shift/reduce conflict occurs in the handling of the 'statement' variable when bison encounters the string: 'if <expression> then <statement> else <statement>'. Here bison can choose to reduce the 'if <expression> then <statement>' part of the string to a 'statement', or shift and move to another state as seen in 'Listing 6'. Shift/reduce conflicts are handled with shifting in bison by default, which is the correct handling in this case.

Listing 6: Shift/reduce conflict in Bison.

```
1   tIF exp tTHEN statement
2       {
3           $$ =  make_STM_if($2, $4);
4       }
5       | tIF exp tTHEN statement tELSE statement
6       {
7           $$ = make_STM_if_else($2, $4, $6);
8       }
```

Bison produced two shift/reduce conflicts which we resolved by adding the option '%expect 2' which is used in bison to silence warnings, however, the expected action for these two conflicts is to shift anyway, which bison does by default, therefore, we can tell bison to expect two conflicts.

2.3.4 Precedence

Precedence is an important concept, when making the grammar unambiguous. If precedence is not used in context with 'EXP', then various legal syntax trees can be constructed from the same input. As an example, multiplication needs to have higher precedence than addition, so that a mathematical expression is understood correctly. In bison '%left' is used to indicate which tokens have left precedence. 'Listing 7' shows how we have chosen to declare precedence in bison.

Listing 7: Precedence in Bison.

```
1 %left tOR
2 %left tAND
3 %left tEQUALITY tNON_EQUALITY
4 %left '<' '>' tLESSER_EQUAL tGREATER_EQUAL
5 %left '+' '-'
6 %left '*' '/'
```

To give one set of characters a higher precedence, than another set in bison, we declare the precedence of the first mentioned set of characters last.
Multiplication and division has the same precedence, and also has left precedence. We have that multiplication and division has the highest precedence, since we want a mathematical expression such as '$1 + 2 * 3$' to be treated the same as the expression '$1 + (2 * 3)$. We declare the precedence of multiplication and division last, using '%left'.
Addition and subtraction has the next higher precedence, and also has left precedence.
We also let $<$, $>$, $<=$, and $>=$ have left precedence, and to have the third highest precedence. This is because we want the mathematical expression '$3 < 2+2$' to be treated the same way as the expression '$3 < (2 + 2)$'.
The relational operators equality ($==$) and non-equality ($! =$) has the fourth highest precedence, and also has left precedence.
At last, we have that the logical operation 'and' (&&) has higher precedence than the logical operation 'or' (||).

2.4 Abstract Syntax Trees

An abstract syntax tree (AST), which is constructed by the compiler during the parsing phase, is a structural representation of some user input.
An example is given below in Figure 9 which depicts the code in Listing 8:

Listing 8: Simple falcon program.

```
1 func c () : bool
2   return true;
3 end c
4
5 if (true) then write 0; else write 1;
```

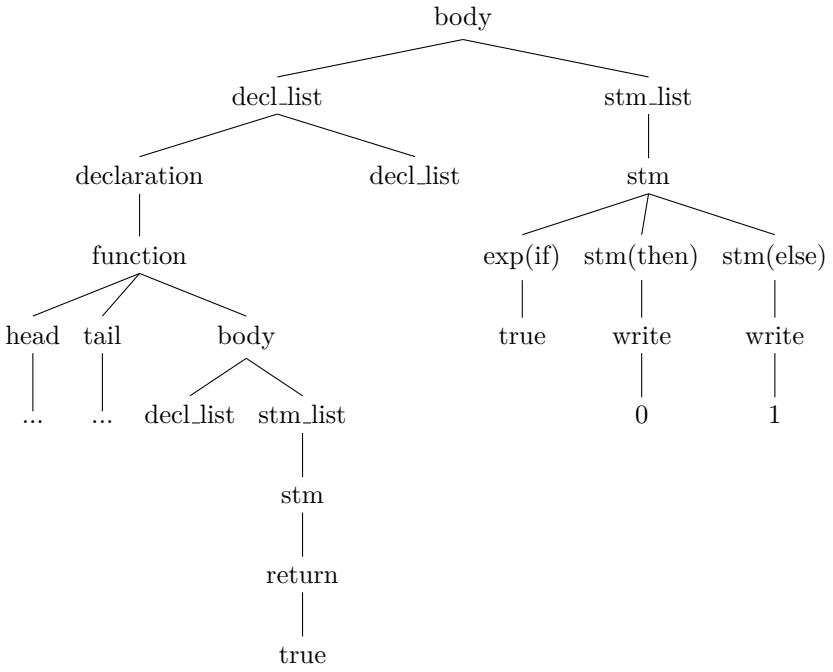


Figure 9: AST for a simple falcon program.

An AST indicates that the whole program can be expanded from, or rewritten to the start symbol. Figure 10 depicts how `decl_list` together with `stm_list` can be rewritten to the start symbol `body`.

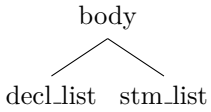


Figure 10: AST for a the `<body>` non-terminal.

2.4.1 Common characteristics of the nodes of the AST

When 'Bison' reduces a right-hand side of a rule, it calls a C function which creates a node to the AST, which represents the left-hand side of that same rule. `'tree.c'` contains a function, used by 'Bison', for each rule, and additionally a function for each kind of operator for the expression rule:

⟨expression⟩ : ⟨expression⟩ **op** ⟨expression⟩

In the AST of the Falcon compiler, only non-terminals are represented as nodes, because the terminals such as semicolon ";" or parentheses "(...)" don't convey any useful information to the remaining phases of the compiler.
When the right-hand side of a rule is reduced to the left-hand side, and made into a node in the AST, the node retains information of the variables of the reduced right-hand side.
Common for all nodes in the AST, is that they contain three pieces of information, namely `'lineno'`, `'type_info'` and `'code_info'`.
`'lineno'` is used by the type checking phase of the compiler, to give supplementary debugging information, in the form of a line number of a possible error, in the user-made program.
The `'type_info'` information is used by the type checking phase of the compiler. The `'code_info'` information is used by the code generation phase of the compiler.

2.4.2 Special nodes of the AST

2.4.2.1 Function

In addition to the common characteristics of a node in the AST, the `'function'` node holds a pointer to a symbol, and a pointer to a symbol table. The symbol table pointer is used by both the type checking phase, and the code generation phase of the compiler, and it represents the new scope that is created on account of the function.

2.4.2.2 Variable declaration list

The `'var_decl_list'` node contains a supplementary array.

2.4.2.3 Declaration

The `'decl'` node contains a supplementary string which is used to hold a variable name. This is convenient since a `'decl'` node always involves a variable.

2.5 Syntactic Sugar

2.5.1 Increment of variable

The use of increment by 1 is often used in programming languages. The increment by 1 of the value of some variable `a` in the `falcon` languages without syntactic sugar, is seen below.

Listing 9: Increment by 1 in falcon.

```
1  a = a + 1;
```

2.5.2 Adding value to variable

The following are all evaluated to an assignment statement.

⟨statement⟩ : ⟨variable⟩ + = ⟨expression⟩ **;**
| ⟨variable⟩ − = ⟨expression⟩ **;**
| ⟨variable⟩ ∗ = ⟨expression⟩ **;**
| ⟨variable⟩ / = ⟨expression⟩ **;**
| ⟨variable⟩ ++ **;**
| ⟨variable⟩ −− **;**

Figure 11: syntactic sugar of assignment statement extensions.

2.6 Weeding

Between the parsing phase and the type-checking phase, our weeding phase begins and runs to weed-out invalid function names. We also wanted to ensure that the compiler halts after the weeding phase and before the type-checking phase if any error should occur. This allows the compiler to quickly identify problems rather than moving to the next phases and spending a lot of time on those.

2.6.1 Function names

Our compiler will throw an error when the function `<head>` identifier and the `<tail>` identifier are not equal, thus the user must explicitly specify a correct and matching id for both the function head and tail. Below are examples of legal and illegal function declarations for Falcon.

Listing 10: Legal function declaration of head and tail

```
1 func Hello() : <type>
2   ...
3 end Hello
```

Listing 11: Illegal function declaration of head and tail

```
1 func Hello() : <type>
2   ...
3 end World
```

This can be achieved inside the `<Function>` node itself, However, we've chosen to do this checking in the `<tail>` node instead. Our implementation relies on passing down the function pointer to the tail node of the function, such that the name of the function can be retrieved from the `<head>` node and compared

effectively. If the comparison results in an inequality then an error is printed to the user.
Furthermore, an error counter is incremented, once we are done weeding-out, we will check if the error counter is greater than zero, if it is then the compiler will halt. This is done because we want to present all the errors and warnings to the user in one go, instead of halting immediately after a single error has been encountered. If we halted immediately we would have presented the user with the first occurring error then the user must fix the error before the user could see the next errors in their program. This is rather cumbersome and we wanted Falcon to be as user-friendly as possible.

2.6.2 Function Return Types

We've chosen to move checking for function return types to the type checker instead and we have also chosen to allow functions to return from the main scope. This is because we wanted the programmer to be able to specify the return code back to the operating system.

2.7 Test

2.7.1 Parser tests

The parser parses all test inputs correctly, with the notable exception of `O_AbsoluteValueTest.src`. The test fails due to our simple implementation of absolute values, which do not account for complex operations. Due to this, the parser mistakes the logic symbol `or` (||) with nested absolute values. A fix could be to implement absolute value similarly to multi-line comments by using start conditions. However, we had to focus our efforts elsewhere since the benefits did not outweigh the amount of work required to implement it.

2.7.1.1 Precedence

2.7.2 Weeder tests

The weeder successfully recognizes mismatched function head and function tail, and prints error messages accordingly.

3 The Symbol Table

To efficiently store variable names, function names, and proper types when compiling, we use a symbol table structure. The symbol table is, in its core, a hash table that uses chaining to resolve conflicts.

3.1 Scope Rules

Throughout the report We refer to the outermost scope as the `"root scope"`. Figure 12 shows an example of a scoping tree. Each node (illustrated as a box) depicts a symbol table. There is only one root scope, which in the figure is the top box. All scopes, which are not the root scope, points to exactly one other scope (this is illustrated by a black arrow in the figure). All the values in a symbol table are available to the symbol table itself, and to any symbol table which points at it. Thus, the values of the root scope is available to any other scope.

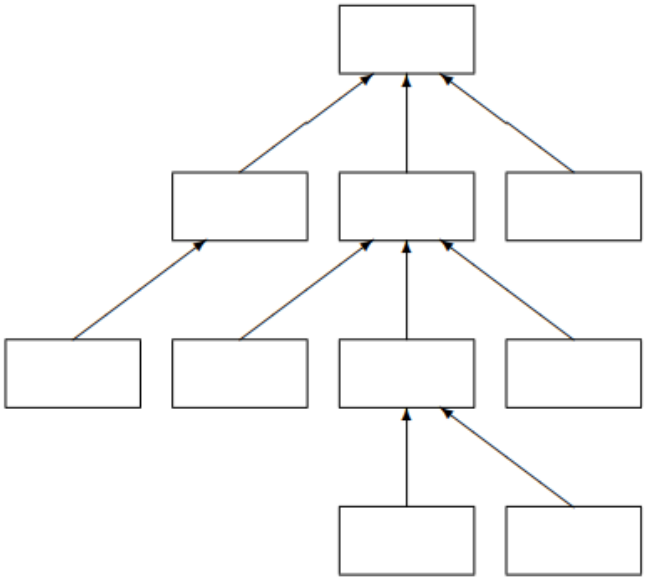


Figure 12: Scoping tree illustration.

3.2 Symbol Data

The symbol table used in our compiler practically functions as a structure, that includes an array of symbols, and a pointer to an inner-more scope.

The content of the symbol table, i.e. the symbol structure, is implemented in C, as shown in Listing 12.

Listing 12: `SYMBOL` struct from symbol.h

```
1  typedef struct SYMBOL
2  {
3     char *name;
4     void* value;
5     struct SYMBOL *next;
6
7     char* func_label;
8  } SYMBOL;
```

'`name`' contains the string to be inserted and the '`value`' pointer contains a '`type_info`' struct from the type-checker.

'`func_label`' is only used in the code generation phase and was inserted because we needed to store a '`function_label`' in the symbol table, and we didn't want to overwrite the `type_info` struct which was already stored inside from type-checking.

To calculate the index, '`name`' is hashed using our own string hashing algorithm.

In case of a collision in the array, the symbol 'next' seen in line 6 of Listing 6, is used to resolve the conflict using chaining.

3.3 The Algorithm

3.3.1 String Hashing

In order to store strings (`name`) in our symbol table as keys, we must have a good hash function that returns a hash-code for a string.

For each characters in the evaluated string, except the last, we take the ASCII value, add it to the partial result, which is initialized to be zero, and then shift the partial result one position to the left. Finally, we add the ASCII value of the last character to the partial sum, but this time we don't make a shift. The partial result after this point is the hash value, which we return. This method is called the shift-add multiplication.
A regular implementation of the above hash function requires each character in the string to be computed, and thus leading to a run-time of $O(n)$. This is impractical for large strings, however, in our case we have chosen to bound the length of the input string to some constant (31), such that we only compute the first 31 characters e.g. just like the C programming-language standard, thus achieving a string hashing speed of $O(1)$.
Lastly our hash function returns an unsigned integer instead of an int, to prevent the hash value overflowing into a negative value, since we also use the hash return value as index in the array.

3.3.2 Putting a symbol into a symbol table

When putting a symbol into a symbol table, we first check whether the symbol table exists. If it does, as is expected, we use the name field of the symbol to get a hash value, using the hashing method described above. If the entry in the hash table, where the symbol hashes to, is empty, the symbol is inserted here. If the entry is not empty, we recursively go through the linked symbols, until we get to the last linked symbol, and then we link the new symbol, to this last symbol of the chain.

3.3.3 Getting a symbol from a symbol table

When getting a symbol from a symbol table, we first check whether the symbol table exists. If it does, as is expected, we get the hash value from the symbol we are looking for, using the hashing method described above. We then look at the entry in the hash table, where the symbol would hash to, and if that entry contains the symbol we are looking for, it is returned. If we don't find the symbol, we recursively go through the linked symbol table that is pointed to, starting with our initial symbol table, using the same method, until we can return the sought after symbol, or if the symbol is not found after reaching the root scope, return null.

3.4 Test

3.4.1 Hash function

To test the hash function, we evaluate the returned values for various different strings.

Test string	Resulting value	Expected value
"kitty"	3369	3369
"ABC"	495	495
"" (empty string)	0	0

3.4.2 Linking symbol tables

The testing of the linking of symbol tables is conducted by making several linked symbol tables, and then checking if the number of symbol tables in the chain of symbol tables matches the expected length.

Test case	Works as intended
Case 'Linking of one symbol table to another': We create a symbol table `'root'`, and a symbol table `'table1'`. We then make `'table1'` point to `'root'`, and assess the length of the chain of symbol tables starting at `'table1'`, which should be 2.	✓
Case 'Linking of 4 symbol tables': We create the symbol tables `'root'`, `'table1'`, `'table2'` and `'table3'`. We make `'table3'` point to `'table2'`, `'table2'` point to `'table1'` and `'table1'` to `'root'`. We then check the number of symbol tables in the chain of symbol tables starting at `'table3'`, which should be 4.	✓

3.4.3 Symbol insertion

We insert a symbol into a symbol table in various ways, retract it, and measure its values.

Test case	Works as intended
Case 'insertion of a symbol': We want to test that if we insert a symbol into a symbol table using the `'putSymbol'` function from `'symbol.c'`, that the same symbol is correctly inserted into the hash table of the symbol table, and can be retrieved again. We give the symbol the name `"kitty"`, and let the void pointer `'value'` of the symbol be an int with the value 42. We know that `"kitty"` yields the hash key 3369, meaning the symbol should be found in index 3369 % `HashSize` (which in this case is 317) = 199 of the array of the symbol table. We manually look up the symbol in that index of the array, and check that the value of that symbol is 42.	✓
Case 'Update of existing symbol': We insert a symbol into a symbol table, where a symbol with an identical name already exists. We want the insertion of the new symbol to only change the value of the already existing symbol, to be the same as that of the new symbol, and not insert an entirely new symbol into the linked list. We use the `'putSymbol'` function from `'symbol.c'` to insert a symbol into a symbol table. We give the symbol the name `"kitty"`, and set the void pointer `'value'` of the symbol to be an int with the value 42. We then use `'putSymbol'` to insert another symbol with name `"kitty"` into the same symbol table, this time with the value 21. We get the symbol found in the 199^{th} index of the array of the symbol table, and evaluate its value, which should be 21.	✓
Case 'Adding symbol to the linked list': We want to test whether a symbol added to a slot in the hash map of the symbol table, where a symbol with a different name already exists, is correctly added to the linked list. We manually add a symbol with the name `"kitty2"` to the 199^{th} slot of the hash table of the symbol table. We then use `'putSymbol'` to add a new symbol with the name `"kitty"` (since we are sure this name yields a key that hashes to the 199^{th} index). We make the void pointer `'value'` of the new symbol to be an int with the value 5. We then test the value of the symbol linked to the symbol in 199^{th} index of the array of the symbol table, which should be 5.	✓

3.4.4 Symbol Retrieval

We want to test whether symbols added to symbol tables in various ways, using the `'putSymbol'` function from `'symbol.c'`, can be found using the `'getSymbol'` function from `'symbol.c'`.

Test case	Works as intended
Case 'retrieval of a symbol': We add a new symbol to a symbol table using `'putSymbol'`. We let the symbol have the name `"kitty"`, and set the void pointer `'value'` to an integer with the value 15. We then use `'getSymbol'` in which we give the string `"kitty"` and the symbol table as arguments to the function. We then evaluate `'value'` of the retrieved symbol, which should be 15.	✓
Case 'retrieval of a symbol in a linked list': We want to test whether we can find a symbol, which is part of a linked list structure. To test this We manually insert a symbol, with the name `'kitty2'` into the 199^{th} index in the array of a symbol table, and set the void pointer `'value'` of this symbol to be an int with the value 2. Then we use `'putSymbol'` to put a new symbol with name `"kitty"` (which we are sure hashes to the 199^{th} index of the array of the symbol table). We let the value of the new symbol be an int with the value 7. We now use `getSymbol` to retrieve the symbol with name `"kitty"` from the symbol table, which value should be 7.	✓
Case 'two symbols with identical names in different symbol tables': We make a symbol table `'table1'`, and link it to another symbol table `'root'`. We insert a symbol into `'root'`, and another symbol with an identical name, but with a different value, into `'table1'`. When retrieving a symbol from `'table1'`, using the identical name as key, we want to make sure it's the one with the correct value, i.e. the symbol residing in `'table1'`.	✓

3.4.5 Larger symbol table structure

We want to test if our compiler is able to correctly handle a larger, to a slight extend more complicated symbol table structure. We chose to simulate the scoping tree seen in Figure 12.

Test case	Works as intended
Case 'Handling structure from Figure 11': We make up the scoping tree from Figure 11, by creating a new symbol table `'root'`. We then create 3 new symbol tables: `'left_table'`, `'middle_table'` and `'right_table'`, and make them point to the `'root'` symbol table. We then create 4 new symbol tables: `'left_table2'`, `'middle_left_table'`, `'middle_middle_table'` and `'middle_right_table'`, and we link `'left_table2'` to `'left_table'`, and the other 3 symbol tables to `'middle_table'`. Lastly we create two more symbol tables: `'middle_middle_left_table'` and `'middle_middle_right_table'`, and link both to `'middle_middle_table'`. We now place a symbol in each of the symbol tables, and make all the symbols have different values. We then check that only the appropriate symbols are visible from within each symbol table. So, for example the symbols in `'root'`, `'left_table'` and `'left_table2'` (which together makes the 'left branch' of the scope tree) are the only symbols visible from `'left_table2'`. The symbols in `'root'` and `'left_table'`, are the only symbols visible from in `'left_table'` and so on.	✓

4 Type Checking

Type-checking is very important as this is where the rules of the language are established and enforced.

It is very important to emphasize that type-checking only contributes to program correctness and does not provide any guarantee that a program is correct nor incorrect.

Our compiler only utilizes a static type-checking system, which means that run-time type-errors are left to the programmer to resolve, And therefore, errors such as division by zero are not handled during compile time, but rather during run-time.

Below is only a subset of the most interesting decisions and noteworthy nodes.

4.1 Types

4.1.1 id

`id` is used as an identifier of a variable or a parameter. The Id type can be used by functions, variables, and proper types.

4.1.2 int

`int` represents integers, which are 64-bit signed integers.

4.1.3 bool

Booleans are supported by the compiler. They can be set to the values true and false, which technically is represented by 1 and 0 respectively. Additionally they can be set to any integer value, the idea was to make bit-set's easier to create and one can assign a boolean to be equal to 4 which automatically promotes the Boolean to a bitset of 4 Boolean's. Since the assigned value is a non-negative value the first four booleans in the bit-set are flipped to true. One would then be able to access a boolean similarly to an array.

4.1.4 array

The compiler supports arrays, which must consists of objects of the same type.

4.1.5 record

A record contains a list of variable declarations all of which reside in their own scope. The implementation is as follows. A new scope in the symbol table

is created. This scope is used to encompass the variable declarations inside the record. Which means that variables declared inside a record can only be accessed using the scope of the record itself. This particular scope is stored inside the `type_information` struct for later reference.

4.1.6 string

Strings are supported by our compiler. Strings are composed of any number of any character inside quotes. Our compiler allows for literal strings to be concatenated by the addition sign during the parsing phase, however, due to time constraints, the concatenation of dynamically allocated strings is not yet implemented.

4.1.7 null

The null type is similar to the null type in most languages such as Java. Importantly only records and arrays are allowed to be assigned to null.

4.2 Type rules

Below are notably particular type rules. Otherwise, the type rules are gradually described below in the algorithm section.

We allow booleans to be assigned integers, which may seem like a minor limitation at first, but our thinking strategy was to implement a bitset behaviour for booleans. We figured that booleans take way too much space and can be represented as bits, giving us 8 booleans in a char. In order to access them, we would allow booleans to be accessed similarly to arrays. Furthermore, we wanted to allow booleans to be assigned to integers such that that if we assign 4 to a boolean the first four booleans will be flipped to true, and -4 means the first four will be flipped to false. Due to time constraints we weren't able to finish this feature, but the remnants of the code still remain.

4.3 The Algorithm

After the parsing and weeding phase of the compiler, the abstract syntax tree is handed to the type checking phase. In which type-checking begins. Type-checking is done in 3 phases, the collection, calculation, and verification phase. The type checker initializes a new symbol table, which becomes the root scope. The symbol table is then updated to fit the scopes of the input program and filled with the identifiers of the program.
AST is recursively traversed, and the type is derived in each sub-node, and then the type is available to the parent node.
We will below use the word "primitive type" to denote a type that can be resolved to an int, bool, string, null, a record, and an array. Thus the `id` type is

not a primitive type.
Lastly, it is important to note that we cannot do type-checking in one phase since we might have variables declared later on in the source code that we've not resolved or reached in the AST yet.
Consider the following example program:

Listing 13: Example of a valid program requiring multiple type passes.

```
1 var a : my_custom_int_type;
2 var b : a;
3 type my_custom_int_type = int;
4 b = 42;
5 write b;
```

To resolve the types of `a` and `b` we must first resolve `my_custom_int_type` which is not possible since in order to resolve the last type we must first traverse a then b, and then finally be able to resolve a, but we cannot go back to a. Therefore, we require multiple type passes.

4.3.1 Type information

Every node of the AST (except for the tail node) includes a `type_information` struct, which is used by the type checker. When the type checker traverses the tree, it updates the `type_information` of all the nodes. All nodes return a `type_information`. Since the traversing of the AST is done recursively, once the type of a child node is derived, the execution flow will be returned to the parent node. The parent returns its appropriate type based on the information from the children.

4.3.2 Recursive type derivation

The type checker recursively derives the types once they've been inserted into the symbol table. Recursive type derivation can be seen as a linked list of identifiers in which the last element will eventually contain a primitive type. The type information struct contains a child node called `type_info_child` which is a child of the type. The child may contain a child of its own, and thus the traversal to a child whose value is null denotes that we've reached the primitive type. To summarize in order to derive the primitive type, we must recursively follow the children until the primitive type has been found.

4.3.3 Collection phase

In the first sub-phase of the type checker, which is the collection phase, all variables, proper types, and function declarations are inserted into the symbol table. This information is stored inside the nodes of the AST for later use by

the remaining type checking phases and the code generation phase. Nodes that are important to the collection phase will be discussed below.

4.3.4 Calculation/verification phase

In the calculation and verification phase of the type checker, we traverse the AST again and resolve the types that have already been inserted into the symbol table. To create a correct symbol table.

4.3.5 `type` node

During the collection phase the `type` node allocates a new `type_information` and fills it with the information corresponding to the type of the `type` node. For records, a new scope is created and the variable declarations of the record are recursively traversed with the newly created scope.

4.3.6 `declaration` node

During the collection phase the declaration node,

The following is the type declaration grammar. Type declaration allows the programmer to derive a new type based on a primitive type. It can be seen as an alias in other programming languages.

type id = ⟨type⟩ **;**

Figure 13: Type declaration grammar.

During the Collection phase, we first derive the type of the ¡type¿ node. Then the type is inserted as a child, and then the whole node containing the child is now put in the symbol table. Thus when we come back in the calculation/verification phase, we will resolve the type and assign it correctly. It proceeds in the following manner, it checks whether it can retrieve a symbol from the symbol table using the node `id` as key. It recursively obtains the 'true' type associated with the identifier, i.e. recursively examines nested ids until the 'original' type is found. It then puts a `type_information` containing the information on the type into the symbol table, using the id as a key.
If the declaration node can be expanded into a function, the type checker will check if the function id can be found in the symbol table, and print an error otherwise.

4.3.7 `function` node

When the type checker reaches a `function` node during the collection phase, it creates a new scope to the symbol table tree, and saves a pointer to that scope inside the `function` node.
Subsequent nodes will then use the new scope. In the function node, the return type is also calculated, the body node returns a type_information as well and is compared to the functions own type, if there's a mismatch an error message is printed out. If the body node of the function does not include a return statement, the type checker writes a warning, but does not exit.

4.3.8 `head` node

func id (⟨par_decl_list⟩ **):** ⟨type⟩

Figure 14: Function head grammar.

In the collection phase, a similar approach is used as to the type declaration node above. First, we collect the type and assign it to a child. Then during the verification and calculation phases, we come back to resolve the type correctly and update the symbol table to reflect the correct type.

4.3.9 `var_type` node

Again the var_type node uses a similar approach to the function head and type declaration by first assigning a child in the collection phase then in the verification/calculation the correct derivation of the type is calculated. It proceeds in the following manner. It checks whether it can retrieve a symbol from the symbol table using the node id as a key. If no symbol can be retrieved or the node doesn't contain an id, an error is returned. The compiler checks whether the value, i.e. the `type_information`, of the retrieved symbol has already been verified, in which case we don't need to examine further. Finally, we want to insert a symbol into the symbol table, using the id as the key, and the `type_information` of the type as value. We want to be able to handle the case of nested identifiers. We recursively derive the type and update the symbol table.

4.3.10 `var_decl_list` node

In this node, we create an array and insert types in order. Functions and records can then use this array. The function head node traverses a par_decl_list which in turn calls this node and along the way an array is created, and when the control reaches the function head node again the array is fully populated

with all the parameter declarations, these are then stored in the functions own `type_information` for later processing and retrieval. A similar process occurs with records.

4.3.11 `expression` node

During the calculation phase, the type checker calculates the types of all expressions and sub-expressions. The information on the type of the expression is stored in the corresponding node of the AST. To do this, it is necessary to obtain a `type_information recursively` from both the left and right expression of the node. These are used to check whether the expression involves an operation on two compatible types. Our compiler supports two kinds of expression operations: i.e. the mathematical operations, and the Boolean operations. The allowed mathematical expressions are seen Table 1 (in which 'str' stands for string). Performing mathematical operations on other type combinations will lead the type checker to give an error. The allowed Boolean expressions are seen in Table 2 (in which 'str' stands for string, 'arr' stands for array, 'rec' stands for record). Performing Boolean operations on any other type combination will lead the type checker to report an error. If the expression node involves a (legal) mathematical operation, the type checker will create a new `type_information` and set its information to reflect that the node is an integer type, and store it in the expression node of the AST. The same technique is used if the node involves a Boolean operator, only in this case the `type_information` will reflect that the node is of type `bool`.

multiplication	id $*$ id	id $*$ int	int $*$ id	int $*$ int
division	id / id	id / int	int / id	int / int
addition	id + id int + int	id + int	str + str	int + id
subtraction	id $-$ id	id $-$ int	int $-$ id	int $-$ int

Table 1: Legal mathematical expressions.

equality (==)	id == id, int, bool, str, rec, arr bool == id, int, bool null == id, rec, arr, null rec == id, rec, null	int == id, int, bool str == id, str arr == id, arr, null
non-equality (=!)	——"——	
greater (>)	id > id, int, bool bool > id, bool, int	int > id, int, bool
lesser (<)	——"——	
greater-or-equal (>=)	——"——	
lesser-or-equal (<=)	——"——	
and (&&)	——"——	
or (||)	——"——	

Table 2: Legal Boolean expressions.

4.3.12 `statement` node

4.3.12.1 assignment

⟨statement⟩ : ⟨variable⟩ = ⟨expression⟩ ;

The assignment statement is handled differently then the rest of the statements, in particular because we need to make sure that records are assigned properly. During the calculation/verification phase we will recursively derive the types of both the expression and the variable. Once those types have been derived they are compared to each other, if they're not the same type, they are both checked and allowed according to their type:

- booleans and integers and vice versa are allowed to be assigned to each other.
- strings can only be assigned to strings. This also means that the null type is not allowed to be assigned to strings.
- Arrays and records can be assigned the null type.

If, however, the types are equal, then they can be assigned without further checking with notable exceptions of arrays and records.
For arrays, the requirements are that their underlying type is the same. E.g. An array of strings cannot be assigned to an array of integers.
For records, we must first lookup their variable declarations in their scopes and then compare the type of each element in record A with the type of the element of record B. The comparisons are made in order, which means that records are not sets but rather a list of types. If a mismatch between the types of any of

the elements is found, an error is produced.

4.3.12.2 if-statements

Only expressions of the type `bool`, `int` are allowed in if statements.

4.3.13 `term` node

When the type checker reaches a function call term node, It proceeds as follows. The type checker verifies that the variable is indeed a function. That the amount of arguments specified in the function call matches the those specified in the head of the function, And that each type (similarly to assigning records described earlier) are precisely the same, any mismatch will trigger an error. The only case where the types are allowed to differ is if either the function call argument or function head parameter is a record or an array, and the corresponding parameter/argument is null.

4.3.14 `variable` node

4.3.14.1 id variable

⟨variable⟩ : **id**

When the compiler reaches a `id variable` node, it is known due to our grammar that we have already traversed all declarations and, therefore, we must be inside a statement, expression, or a variable. At this point, all we have to do is look up the id in the symbol-table recursively and return the type back up. We do not have to put any symbols and update the symbol table because we've already traversed all the declaration nodes.

4.3.14.2 array indexing variable

⟨variable⟩ : ⟨variable⟩ [⟨expression⟩]

When the compiler reaches a `array indexing variable` node, the compiler looks up recursively the type of both the expression and the variable nodes.
The two requirements here is that the expression must be of type integer. This requirement can be extended to strings as well to allow string indexing like

python and many other languages. However, this is not in the current implementation.
The second requirement is that the variable must be an array. Otherwise, a type error is produced.
If the requirements are met, the compiler looks up the type of the array itself and return that back up.

4.3.14.3 record access variable

⟨variable⟩ : ⟨variable⟩ **.id**

When the compiler reaches a `record access variable` node, the compiler looks up recursively the type of the variable and ensures that the type is a record.
If its a record, the scope of the record is used to look up the `id` in the symbol table and return that back up.

4.4 Test

#	Test case	Works as intended
1	C_ErrAssignToType	✓
2	C_ErrFuncParamsInvalidType	✓
3	C_ErrFuncParamsTooFew	✓
4	C_ErrFuncParamsTooMany	✓
5	C_ErrInvalidToken	✓
6	C_ErrTypeLoop	Fail
7	C_ErrUnmatchedBeginComment	✓
8	C_NullWrong	✓
9	C_ReturnInMainScope	✓
10	F_FuncParamsEvalOrder	✓
11	F_RecordIsTupleOrSet: Fails because a record is not a tuple but rather a set, where the ordering matters.	Fails
12	F_ShortCircuitAND	✓
13	F_ShortCircuitOR	✓
14	F_SimpleStructuralEquiv	✓
15	O_AbsoluteValueTest: Fails during the parsing phase.	Fail
16	O_AbsTest	✓
17	O_ArrayComparisonsA	✓
18	O_ArrayComparisonsB	✓
19	O_ArrayIndex	✓
20	O_ArrayLength	✓
21	O_ArrayOfOwnType	✓
22	O_ArrayOfRecords	✓
23	O_Assoc	✓
24	O_BinarySearchTree	✓
25	O_Comments	✓
26	O_Factorial	✓
27	O_FuncCallAsParamA	✓
28	O_FuncCallAsParamB	✓
29	O_FuncModifyingParams	✓
30	O_FuncRedefinedInItself	✓
31	O_FuncRedefinedReturnType	✓
32	O_FuncRedefinedType	✓
33	O_FuncReturnRecord	✓
34	O_Function	✓
35	O_IfThen	✓
36	O_Knapsack	✓

Comments to test 6: The typechecker does not terminate. This case is not implemented correctly in the typechecker. A possible fix would be to flip a boolean once the type is resolved.
Comments to test

37	O_KnapsackNoComments	✓
38	O_LargeExpTreeA	✓
39	O_LargeExpTreeB	✓
40	O_LargeExpTreeC	✓
41	O_MultiDimArray	✓
42	O_MultipleTypecheckPassesA	✓
43	O_MultipleTypecheckPassesB	✓
44	O_MultipleTypecheckPassesC	✓
45	O_NullCorrect	✓
46	O_RecordComparisonsA	✓
47	O_RecordComparisonsB	✓
48	O_RecordsWithArray	✓
49	O_Recursion	✓
50	O_SimpleRecord	✓
51	O_StaticLink ✓	
52	O_StaticLinkA	✓
53	O_StaticLinkB	✓
54	O_TypeJumpScope	✓
55	O_WhileDo	✓
56	R_ErrOutOfBounds1	✓
57	R_ErrOutOfBounds2	✓
58	R_ErrRuntimeDiv0	✓
59	R_ErrRuntimeNegArraySize	✓
60	R_ErrRuntimeNullPointer	✓
61	R_ErrRuntimeOutOfMem	✓

4.5 Supplementary Tests

4.5.1 Function tests

4.5.1.1 Function calls

We test function calls with different kinds of argument types, i.e. we test arguments and parameters of type `id`, `int`, `bool`, `string`, `record` and `array`, and also different number of arguments.

#	Test case	Works as intended
1	Test whether function calls with correct number of arguments, and correct argument types in relation to the function parameters yields no errors as expected.	✓
2	Test whether function calls with correct number of arguments, but incorrect argument types in relation to the function parameters yields errors as expected.	✓
3	Test whether function calls with incorrect number of arguments, both too few and too many, yields errors as expected.	✓
4	Test whether a function call using a variable, which is defined as something else than a function yields an error as expected.	✓

4.5.1.2 Function return statements

Return statements and return values for functions are tested with the types `id`, `int`, `bool`, `string`, `record` and `array`.

#	Test case	Works as intended
1	Test whether a function with 1 or more return statements, which returns correct values in relation to the function type, yields no error as expected.	✓
2	Test whether a function containing 1 return statement, which returns an incorrect value in relation to the function type, yields an error as expected.	✓
3	Test whether a function containing multiple incorrect return statements yields an error.	Failed
4	Test whether a function which includes no return statements yields a warning as expected.	✓

Comments regarding test 3:
Our type checker will only evaluate the type of the first return statement it sees, meaning it won't give any error if the first return statement in the function returns a correct type, no matter the number of incorrect subsequent return statements. An example of an input that should give errors, but doesn't is seen in Listing 12. The type checker will correctly yield an error on the input if the two return statements are flipped however.

Listing 14: function containing incorrect return statement.

```
1 func a () : int
2     if (false) then{
3         return 5;
4     }else{
5         return true;
6     }
7 end a
```

4.5.2 Record tests

We assign a record `'record1'` to a variable, that has been defined to be a record `'record2'`. We now test the following cases:

#	Test case	Works as intended
1	`'record2'` has the same amount and types of elements as `'record1'`, which should yield no errors.	✓
2	`'record2'` consists of a different amount of elements than `'record1'`, which should yield an error.	✓
3	`'record2'` consists of the same amount of elements, but the elements have different types than `'record1'`, which should yield an error.	✓

4.5.3 Expression tests

For each mathematical expression, we want to test whether all the legal expressions, which are depicted in Table 1, doesn't yield any errors as is expected, and also that all illegal expressions, which are mathematical expressions performed on type combinations not depicted in Table 1, does indeed yield errors as expected. We also want to test whether no Boolean expression, which are depicted in Table 2, doesn't yield any errors as expected, and that all the illegal expressions, which are all the Boolean operations performed on combinations of types not depicted in Table 2, does indeed yield errors as expected.

#	Test case	Works as intended
1	Test whether no legal mathematical expression yields any error, which they are expected not to.	✓
2	Test whether all kinds of illegal mathematical expressions yields an error, as is expected.	✓
3	Test whether no legal Boolean expression yields any error, which they are expected not to.	✓
4	Test whether all kinds of illegal Boolean expressions yields an error, as is expected.	✓

4.5.4 Statement tests

#	Test case	Works as intended
1	Return error if write statement includes either a record or an array.	✓
2	Return error if function variable is assigned any expression.	✓
3	Test whether if and if-else statements only accepts expressions of type `id`, `int` and `bool`	✓

5 Code Generation

The purpose of code-generation is to generate a intermediate representation (IR) or finished assembly code from the abstract syntax tree (AST) and the information put into the symbol table by the type-checker. The way to achieve this is by traversing the AST, similar to the type-checking phase, but in this case preferably only once. Thus in the process the AST can be evaluated and the code can be generated by help of the symbol table considering scopes, and variables.

5.1 Strategy

In our compiler we do not use any intermediate code we simply translate the AST directly to assembly code. This results in a different method than would be expected by translating the AST to an intermediate code. Mainly liveliness analysis and register allocation would be a part of the back end, but in our case it is not, thus we have to handle the code generation in relation to the stack frame (aka. activation record) differently than what could be expected in a compiler with intermediate representation between the back end and front end. The activation record consists of the base-pointer that points to the start of the frame then the local variables. When expressions are calculated the values from, for example the variables are pushed to stack, popped when the expression is calculated then the result is pushed and popped where ever it is needed, either for evaluation or variable assignment or outputting to user. Thus registers are only used when a calculation is happening and expression results will be pushed to stack and popped when the inevitable use of the result occur. Thus we do not have the need for saving registers before or after a new stack frame is made (caller and callee save).
Following the parameters for the next stack frame are pushed to stack which includes the static link and then the new stack frame is created.

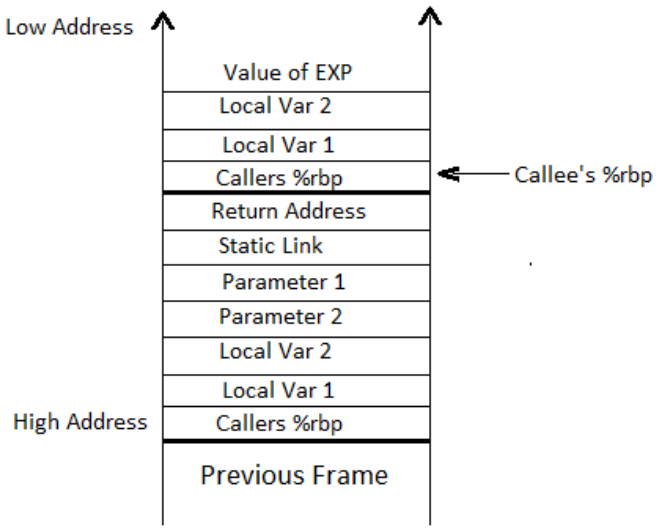


Figure 15: Stack Frame.

5.2 Code Templates

5.2.1 Template for declarations

5.2.1.1 Variable Declaration:

Listing 15: Variable Declaration.

```
1 Push 0;
```

When a variable is declared there is simply made room for the variable on the stack by pushing the value 0 to the stack. This implicates that all declared variables will be initialized with the value 0, thus we avoid any errors of using unassigned variables after the declaration which could be a problem if we simply move the stack pointer to make room for the variable, and do not keep in mind that a random value could be stored in the uninitialized memory area.

5.2.1.2 Function Declaration:

Listing 16: Function Declaration template.

```
1 func_start:
2     pushq %rbp
3     movq %rsp, %rbp
4
5     code for variable declarations
```

```
6       code for function body
7   func_end:
8       movq %rbp, %rsp
9       popq %rbp
10      ret
```

When a function is declared a start label is created. After the declarations and statements of the function a return statement is created that pushes the return value to the register %rax, which always will be used directly after the function call since following the grammar a function can only be called by derivation from an expression non-terminal, this will be explained with more detail in the section "Algorithm 6.3". It can also be seen that the prologue and epilogue happens by default in every function to establish the correct stack frames for each function that is entered into and returned from.

5.2.1.3 Array Declaration:

Declaration of an array happens the same way by the same code as for variable declarations. Mainly the "code_gen_VAR_TYPE" functions handles declarations and pushing the initialization value of 0 to the stack. Thus the array pointer is seen as a variable and will later point to a place in the heap wherein the first value of the array resides.

5.2.2 Template for statements

5.2.2.1 Return Statement:

Listing 17: Return statement template.

```
1  code for <expression>
2  pushq <exp>
3  popq %rax
4  jmp func_end
```

When it comes to return statements we know that the expression of the return statement has been pushed to stack by convention and thus it will be enough to pop the value from stack to the %rax register.

5.2.2.2 Write Statements:

Listing 18: Write statement template.

```
1  code for <expression>
2  code for inserting arguments to write
3  call printf
```

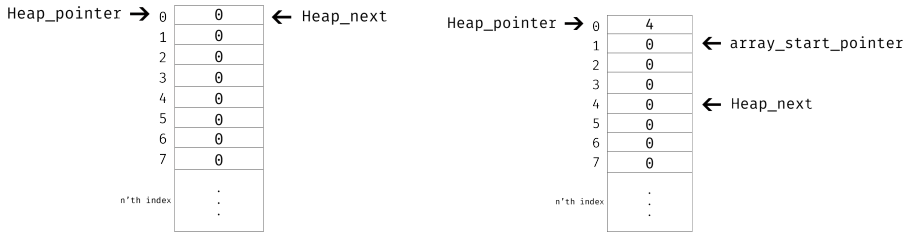
When writing to ”stdout” the ”printf” function is simply called with the right arguments.

5.2.2.3 Allocate Length Statement:

We first generate the code for the expression and pop the result from the stack. The expression contains the allocation length.
We will then generate code for the variable and retrieve the variable's offsets on the stack to reference back to the variable.
The value of `heap_next` is retrieved and then we add the value to the `heap_pointer` which is our heap, since we want to go to the next free memory location.

Once we have resolved the new memory location, we will insert the allocated length into the first element.
The memory location is then incremented by 1 going to the next location in memory we will then assign the variable the current memory location and update `heap_next` correspondingly

Below is a visual example with an allocation length of 4:



(a) Initial heap layout

(b) Heap layout after allocation.

Figure 16: Example heap with an array allocation of length 4.

`array_start_pointer` denotes the index assigned to the array.

5.2.2.4 Assignment Statement:

Listing 19: Assignment statement template.

```
1 code for <expression>
2 code for finding the right frame pointer and offset
3 mov "<expression>-result", "variable offset"(%rbp)
```

When assigning a value to a variable, it is assume that the value is simply pushed to the stack, or that we can expect it in the %rax register when the variable is assigned to the result of a function. Since all variables reside within the stack frame we simply have to keep track of the offset from the base pointer.

5.2.2.5 If Statement:

Listing 20: If statement template.

```
1       code for <expression>
2       cmp "<expression>-result", 1
3       jne end_if
4       code for <statement1>
5   end_if:
```

As seen on the template above if statements are relatively simple code generated for the expression, then it is compared to 1 (which is our representation of true in our convention). Then the control flow jumps to the correct label depending on the result of the comparison.

5.2.2.6 If Else Statement:

[caption=If-else statement template.]

```
1       code for <expression>
2       cmpq "<expression>-result", 1
3       jne else_part
4       code for <statement1>
5       jmp end_if_else
6   else_part:
7       code for <statement2>
8   end_if_else:
```

Here it is seen that the "if else" statement resembles the "if" statement closely. The difference is how ever that if the expression in the "if" statement is evaluated to false the "code for ¡statement2" is executed.

5.2.2.7 While Statement:

Listing 21: While statement template.

```
1   while_start:
2       code for <expression>
3       cmp "<expression>-result", 1
4       jne while_end
5       code for <statement>
6       jmp while_start
7   while_end:
```

For the while statement a loop have to be created that is done by evaluating the expression and treating it as the condition for the loop. If it still evaluates

to true, the statement executes once more, otherwise the loop will end.

5.2.3 Template for expressions

When it comes to the templates for the different expression that the compiler supports, the template is very similar for all the expressions. As stated earlier, all expressions pushes their results to the stack after popping the operands to specific registers and computing the specific registers with the operator. Thus all expressions and sub-expression can be exprected on the stack when needed.

5.2.3.1 Expressions:

Listing 22: Expression template.

```
1 code for <expression1>
2 "place result on stack"
3 code for <expression2>
4 "place result on stack"
5 "pop results from stack perform operation, push result to stack"
```

The latter template is a generalization that fits all the expression of the compiler, which are multiplication, division, addition and subtraction. This also hold for Boolean operations and comparison operators such as relational-equality, relational-inequality, relational-ordering (<,<=,>,>=), Boolean AND and Boolean OR.

5.3 Algorithm

The code generation phase is performed and managed by a couple of different algorithms that will be explained here. The main goal of all these algorithms is to have the logic that is the foundation for creating the final assembly code when a program is given to the compiler.

5.3.1 Functions

5.3.1.1 Function Label:

The algorithm and mechanism of creating the labels of a function may be one of the simplest algorithms in the code generation phase. To understand this part, we look into the function called "code_gen_FUNCTION" within the "code_gen.c" file.
First of all, when the control flow of the compiler is in the "code_gen_FUNCTION" function, it is because that we traverse the AST and the recursive traversal of the AST has reached a function. A unique function label has to be created because every label in the finished assembly code has to be unique for it not to

be confused with other labels. The way this is done is by appending a number to the function name. The number will then be incremented. We call this global integer for "global_func_lable_counter". Now when this unique label for the function has been created, and the global integer variable will be incremented so that the next function will have a unique label. We assume that function overloading could be implemented in the future. Thus it would be wise to allow functions to have the same function name and differentiate between them in the assembly code by having a unique label. This method just described will take a function name found in the AST node of the function "factorial" and now append an integer after an underscore and a colon. Thus the function label will look like "factorial_0:" as an example, and then this unique function label will be saved within the "SYMBOL" of the function. Specifically, the label will be saved in the variable "func_label".
Following an end label for the function has to be created, this is done in the "code_gen_TAIL" function of the "code_gen.c". This is done by getting the function name of the function, prepending the string "end_" and appending the number of the function. This number should be the same as the start label unique number. Now, this is where we run into a problem. First of all, when the number from the end label is created, it is calculated from the "global_func_lable_counter" and the "func_neasting_depth". The calculation works for nesting functions only, but it breaks when there are two function definitions within a function definition. This calculation of the unique end label counter is entirely unnecessary because the complete function name with the unique number could be retrieved from the "SYMBOL" of the function within the "SymbolTable" where it was saved by the time the start label was created. This is a very notable error in the code of the compiler and is a result of neglecting or forgetting to test all essential cases.

5.3.1.2 Separation of Functions:

In assembly code functions need to be represented in a specific way, which means that for example there cannot be nesting of functions even if the functions of the input language in the compiler is declared in a nesting manner. This can be handled in a couple of ways. The way it is handled in our compiler is by creating a linked list that contains a reference to separate linked lists in each node, as seen in the following figure.

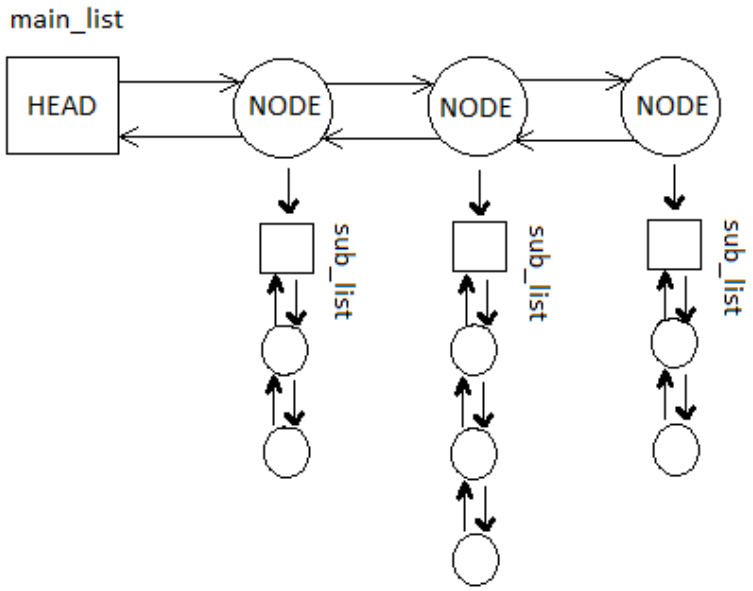


Figure 17: linked list of sub-linked lists that contain the code.

When entering the `code_gen.c` file the `main_list` is created. Each time some assembly code is produced, it is appended into the last `sub_list` in the `main_list`. When a new function node of the AST is reached, a new node in the main_list is created, and thus when the `linked_list_insert_function` function is called with one of the parameters being the string of newly produced assembly code, the assembly code string will be appended the last sub_list in the linked_list called main_list. Thus right before exiting the function node of the recursive AST traversal the function "linked_list_remove_function" is called with the parameter "main_list". This function removed the last node from the main_list and returns its sub_list of assembly code, thus when this sub_list is appended to the linked_list called "list" it will be appended to the list of code that is the final assembly code.

5.3.1.3 Variable and Parameter Offset:

When a variable is declared or parameter is given to a function that is called, there needs to be a accessible number that represents the offset of the given variable or parameter from the current base pointer, so it is possible to access the memory area that contain the requested or needed variable or parameter.
The compiler keeps track of the last variable offset from the scops, by having a stack that saves the last variable offset when entering a new scope and removing the last entry of the stack when exiting a scope. Thus when the compiler exits

a scope the last variable offset from the last declared variable of the given scope is saved in the stack, thus the compiler will know that the new variable will have an offset that is 8 bytes less that the previous offset saved in the offset stack when the compiler is working on a 64 bit system and 4 bytes less when the compiler is working on a 32 bit system.
When a variable is declared two different things happen. First of all a push instruction is added to the assembly code. This push instruction pushes the value 0 to the stack. There is two reasons this is done. There has to be made room for the variable on the stack, where all the variables and parameters reside. The variable is initialized with 0 which means that the user will not have to worry about error when referencing the variable because all variables are initialized.
Second of all The variable will be looked up in the "SymbolTable" and the offset of the variable will be saved in the "SYMBOL" within the variable "par_var_offset" that resides in the "type_info" struct.

When a parameter of a AST function node is meet a similare process is done, the only difference is that no assembly code is produced. The only thing is that the offset is saved to the "SYMBOL" in the "SymbolTable". Now the offset for a parameter is 8 bytes more in the positive direction on the stack than the last parameter, and the offset for parameters should account for that the last thing pushed to the called function is the static link. The parameters of a function is pushed before the function call in reversed order, thus the offset only need to be saved when the assembly code for the function is created, because the assembly code that pushed the parameters is handled by the caller function or scope, which is explained to a further extend in the "6.3.4".

5.3.1.4 Scope:

The scope of the part of the AST/code that is evaluated may be needed at different nodes, but not all nodes of the AST may have a pointer to the SymbolTable that represents the scope. Thus there is a need for the "code_gen.c" to save the scope when the recursive traversal of the AST happens. The whole algorithm for saving the scope happens in the "code_gen_FUNCTION" function. When the code generator starts it sets the pointer "main_scope" in the "data" struct to point at the root node of the "SymbolTable". Then the pointer "scope" in the "data" struct is set to point at that root node also, because it is assumed that any program starts in the global scope. When the traversal of the AST meets a function node, the current scope, which is saved in the "scope" pointer, is put into the local pointer "old_scope", to save the previous scope when we need to return back to it. Then the "scope" pointer of "data" struct is set to be the scope of the function. before exiting the function node in the AST the pointer "scope" is reset to be the the old scope.
So what simply happens is that the "scope" pointer that represents the current

scope, is set to the new scope when the entering a function and reset to the old scope when exiting a function. This is important because code that reside within the function body may need to have access to the current scope when generating assembly code for that function body.

5.3.2 Variables

When a variable or parameter is met the offset of the variable or parameter in relation to the frame pointer is needed. Thus before the offset from the frame pointer can be used the correct frame pointer needs to be found. The way this is done is by either using the current base pointer or de-referencing the static link the correct number of times before the correct frame pointer is found.

5.3.2.1 Scope Count:

The way the correct number of time the static link should be de-referenced is gotten by looking at how many times the compiler has to "step into" a new scope to find the specific symbol that it is looking for. This represents the amount of times the static link has to be de-referenced. Thus if there is no need to de-reference the static link the compiler simply uses the current base pointer and add or subtracts the offset depending on what value is saved in the "SymbolTable". Sometime the only thing needed is to find the offset and the number of times to de-reference the static link, other times the value of the variable is the only thing needed thus it is pushed to the stack.

5.3.2.2 Assignment:

When assigning the result of an expression to a location in memory which could be a variable of parameter, what is needed is the amount of time the static link should be de-referenced and the offset of the variable or parameter. This these values are saved in the "data" struct which is passed on to most functions with in the "code_gen.c" file (this "data" struct could instead simply be a global pointer instead of a parameter to most functions in the file). Following the result of the expression that is assigned to the variable is popped from the stack where it can be expected and the correct static link is found with the offset to the variable or parameter, then the result of the expression is moved to the specific location of the variable or parameter.

5.3.2.3 Push Variable Value:

When a variable or parameter is found in an expression, the value of the variable or parameter is simply pushed to the stack. This again is done by finding the correct number of times to de-reference the static link and then the value found at the location of the correct frame pointer with the offset to the variable or

parameter, will be pushed to the stack as all value in an expression should be. Then this value will be used in the expression for what every it is needed.

5.3.2.4 Array indexing

Since the first element in the array is an is the length of the array itself we must account for this and set the offset accordingly. Therefore we add 8 bytes to the offset to account for this.

To look elements in the array we must reference the `heap_pointer` and look up the elements in there, by first adding the offset and the given index.

Our current implementation of array indexing returns a pointer or the actual value of the element being indexed. However, there are mismatches between the various nodes, and some expect a value instead of a pointer. Therefore, exceptional cases were added to such nodes where it is needed. However, due to time constraints, we were unable to finish the implementation.

5.3.3 Expressions

All expressions push their result on the stack when they are done.

The expression node calls its sub-expressions, and when the control flow is returned to back to the expression node itself, The subexpressions have already pushed their results on to the stack. In which case, all that is required is to pop the subexpressions, compute a result based on the expression type and push the result on the stack. For example:

$1+1$ will look like $< exp > op < exp >$ in the context free language. Thus both operands 1 and 1 are expected on the stack and popped to registers to perform the arithmetic addition operation. The result of this arithmetic addition operation will be pushed to the stack and thus it can be expected to be on the stack for the next evaluated expression or statement.

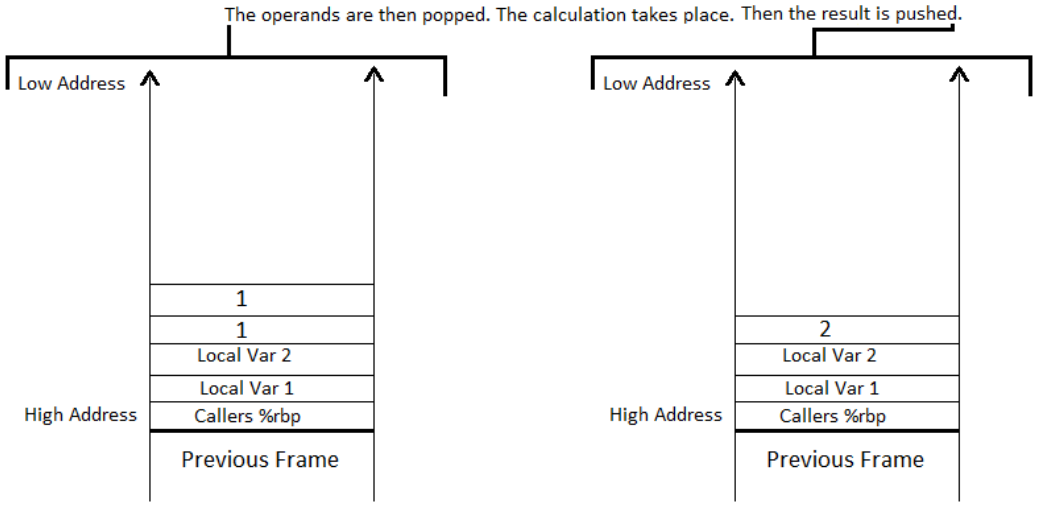


Figure 18: Stack Frame Expression Operation.

5.3.4 Term

All term nodes push their result on the stack, however, the nodes which resolve to `variable` are handled in the variable node itself.

5.3.4.1 Calling Functions:

When calling a function the parameters are pushed in reversed order because a stack frame expects that the first parameter will have the smallest offset to the base pointer and second will have a larger, and so on. Following that the correct static link needs to be pushed, thus again it is needed to find out which base pointer to push as static link, thus needing to know how many time to de-reference the static link and thus pushing the resulting frame pointer. After the function is called the static link should be discarded from the stack and the parameters should also be discarded. This can simply be done by incrementing the stack pointer with the correct offset for the system times the amount of parameters given to the now exited function.

5.3.4.2 Negation:

Negation is a very simple but important instruction. Here the expression that should be negated will be popped from the stack, a negation instruction will be performed, and the result will then be pushed into the stack.

5.3.4.3 Absolute Values:

The cardinality or the absolute value returns the length of an array, or if the type is an integer the value of the integer is returned instead. Integers are not supported for now.

A requirement for arrays is that the array must be allocated before requesting the absolute value of an array otherwise undefined behaviour may occur.

We want to store the length of the array on the heap as well, and preferably in a place where we can retrieve it easily. We have chosen to allocate an extra element on the heap when we allocate an array such that in total we allocate the length of the array + 1. The first element in the heap will contain the length of the array. The length of the array is stored at 1 - the address that the array points at. Thus all we do is subtract 1 from the array address to get length address. The length is then pushed on the stack.

5.3.4.4 Numbers:

When a number is meet with in an expression the value should be pushed to the stack, and also the same happen when a number is meet within an expression that is a parameter to a function call.

5.3.4.5 True and False:

Boolean values in the compiler are simply represented by integers 0 and 1, for False and True respectively. Thus when Boolean values are encountered in an expression these integer values are pushed like any other integer value.

5.4 Test

#	Test case	Works as intended
1	C_ErrAssignToType	✓
2	C_ErrFuncParamsInvalidType	✓
3	C_ErrFuncParamsTooFew	✓
4	C_ErrFuncParamsTooMany	✓
5	C_ErrInvalidToken	✓
6	C_ErrTypeLoop	Fail
7	C_ErrUnmatchedBeginComment	✓
8	C_NullWrong	✓
9	C_ReturnInMainScope: Returning from the main scope is not implemented in the code generation.	✓
10	F_FuncParamsEvalOrder	Fail
11	F_RecordIsTupleOrSet	Fail
12	F_ShortCircuitAND	✓
13	F_ShortCircuitOR	✓
14	F_SimpleStructuralEquiv	✓
15	O_AbsoluteValueTest	Fail
16	O_AbsTest	Fail
17	O_ArrayComparisonsA	Fail
18	O_ArrayComparisonsB	Fail
19	O_ArrayIndex	Fail
20	O_ArrayLength	✓
21	O_ArrayOfOwnType	✓
22	O_ArrayOfRecords	Fail
23	O_Assoc	Fail
24	O_BinarySearchTree	Fail
25	O_Comments	✓
26	O_Factorial	✓
27	O_FuncCallAsParamA	Fail
28	O_FuncCallAsParamB	Fail
29	O_FuncModifyingParams	Fail
30	O_FuncRedefinedInItself	✓
31	O_FuncRedefinedReturnType	Fail
32	O_FuncRedefinedType	✓
33	O_FuncReturnRecord	Fail
34	O_IfThen	✓
35	O_Function	✓
36	O_Knapsack	Fail

37	O_KnapsackNoComments	Fail
38	O_LargeExpTreeA	✓
39	O_LargeExpTreeB	✓
40	O_LargeExpTreeC	Fail
41	O_MultiDimArray	Fail
42	O_MultipleTypecheckPassesA	✓
43	O_MultipleTypecheckPassesB	Fail
44	O_MultipleTypecheckPassesC	Fail
45	O_NullCorrect	✓
46	O_RecordComparisonsA	✓
47	O_RecordComparisonsB	Fail
48	O_RecordsWithArray	Fail
49	O_Recursion	✓
50	O_SimpleRecord	Fail
51	O_StaticLink: This fails because of the sequential evaluation order of the compiler	Fail
52	O_StaticLinkA	Fail
53	O_StaticLinkB	Fail
54	O_TypeJumpScope	✓
55	O_WhileDo	✓
56	R_ErrOutOfBounds1	Fail
57	R_ErrOutOfBounds2	Fail
58	R_ErrRuntimeDiv0	Fail
59	R_ErrRuntimeNegArraySize	Fail
60	R_ErrRuntimeNullPointer	Fail
61	R_ErrRuntimeOutOfMem	Fail

6 Emit

We use a simple `string_builder` data structure to build strings for the emit phase. This proved to be very beneficial because it allowed us to write the instructions for every node and gives us more control and visibility of each instruction.

At the end of each node a string is generated and inserted into our linked_list.

6.1 Example Code

Listing 23: O_simple_string.src

```
var s : string;
s = "Hello, World";
write s;
```

Listing 24: Compiler emit of O_simple_string program.

```
.data
printf_format_int: .string "%d\n"
printf_format_string: .string "%s\n"
printf_format_nl: .string "\n"
.align 8
heap_pointer:
.space 16394
heap_next:
.quad 0
label_0: .string "Hello, World"

.bss
.text
.globl main
main:
pushq %rbp
movq %rsp, %rbp
pushq $0
leaq label_0(%rip), %rbx
pushq %rbx
popq %rcx
movq %rcx, -8(%rbp)
pushq -8(%rbp)
leaq printf_format_string(%rip), %rdi
popq %rsi
movq $0, %rax
call printf
movq %rbp, %rsp
popq %rbp
end_main:
```

As we can see above, all literal strings are stored in the data section and resolved by their label pointer whenever they're referenced.

Listing 25: O_ArrayLength.src

```
var a : array of int;
allocate a of length 7;
write |a|;
```

Listing 26: Compiler emit of O_ArrayLength program.

```
.data
printf_format_int: .string "%d\n"
printf_format_string: .string "%s\n"
printf_format_nl: .string "\n"
.align 8
heap_pointer:
.space 16394
heap_next:
.quad 0

.bss
.text
.globl main
main:
pushq %rbp
movq %rsp, %rbp
pushq $0   #pushing empty value for later assignment of variable
pushq $7
popq %rcx
pushq %rbp     #push address of static link
popq %rbx
movq $heap_next, %r14
movq (%r14), %r14
movq $heap_pointer, %r15
addq %r14, %r15
movq %rcx, (%r15)
addq $8, %r15
movq %r15, -8(%rbx)
movq $heap_next, %r14
movq (%r14), %r14
inc %rcx
movq %rcx, %rax
movq $8, %r13
mulq %r13
addq %rax, %r14
movq $heap_next, %r15
movq %r14, (%r15)
pushq -8(%rbp)
popq %rax
subq $8, %rax
movq (%rax), %r14
pushq %r14
popq %rbx
leaq printf_format_int(%rip), %rdi
movq %rbx, %rsi
movq $0, %rax
call printf
movq %rbp, %rsp
popq %rbp
end_main:
```

First we push 0 to create space in the stack for the variable. then we generate code for the allocation length, and put the allocation length inside the heap, incrementing by 8 bytes to offset for a single element and then storing the address of the array to the new pointer. Then the heap counter is incremented. When we're calculating the cardinality or the absolute value, we will lookup the pointer pointed to by the array, substract 8 bytes to the offset and get a the length.

Listing 27: O_Factorial.src

```
1  func factorial(n: int): int
2      if (n == 0) || (n == 1) then
3          return 1;
4      else
5          return n * factorial(n-1);
6  end factorial
7
8  write factorial(5);
```

Listing 28: Compiler emit of O_Factorial program.

```
1  .data
2  printf_format_int: .string "%d\n"
3  printf_format_string: .string "%s\n"
4  printf_format_nl: .string "\n"
5  .align 8
6  heap_pointer:
7  .space 16394
8  heap_next:
9  .quad 0
10 .bss
11 .text
12 .globl main
13
14 factorial_0:
15 pushq %rbp
16 movq %rsp, %rbp
17 pushq $1
18 pushq 24(%rbp)
19 popq %rbx
20 popq %rax
21 cmpq %rbx, %rax
22 je TRUE_0
23 pushq $0
24 jmp END_0
25 TRUE_0:
26 pushq $1
27 END_0:
28 pushq $0
29 pushq 24(%rbp)
30 popq %rbx
31 popq %rax
32 cmpq %rbx, %rax
33 je TRUE_1
34 pushq $0
35 jmp END_1
```

```
36  TRUE_1:
37  pushq $1
38  END_1:
39  popq %rbx
40  popq %rax
41  orq %rbx, %rax
42  pushq %rax
43  popq %rax
44  movq $1, %rbx
45  cmpq %rax, %rbx
46  jne else_part_0
47  pushq $1
48  popq %rax
49  jmp end_factorial_0
50  jmp end_if_else_1
51  else_part_0:
52  pushq $1    #push param for called function
53  pushq 24(%rbp)
54  popq %r11
55  popq %rbx
56  subq %rbx, %r11
57  pushq %r11       #end of sub expression
58  movq 16(%rbp), %r15
59  push %r15
60  call factorial_0
61  addq $8, %rsp    #discard parameter
62  addq $8, %rsp    #discard parameter
63  pushq 24(%rbp)
64  movq %rax, %rbx
65  popq %rax
66  imulq %rbx, %rax
67  pushq %rax
68  popq %rax
69  jmp end_factorial_0
70  end_if_else_1:
71  end_factorial_0:
72  movq %rbp, %rsp
73  popq %rbp
74  ret
75
76  main:
77  pushq %rbp
78  movq %rsp, %rbp
79  pushq $5    #push param for called function
80  pushq %rbp     #push address of static link
81  call factorial_0
82  addq $8, %rsp    #discard parameter
83  addq $8, %rsp    #discard parameter
84  movq %rax, %rbx     #moving return value in rax to rbx
85  leaq printf_format_int(%rip), %rdi
86  movq %rbx, %rsi
87  movq $0, %rax
88  call printf
89  movq %rbp, %rsp
90  popq %rbp
91  end_main:
```

As seen in the O_Factorial program assembly code the static link is de-referenced zero times because the function is a recursive function. This also means that there need to be pushed the correct static link when calling the function recursively, which is handle quite well.