Bachelor Project in Compiler Construction

Falcon

May 2019

Report from group 3:

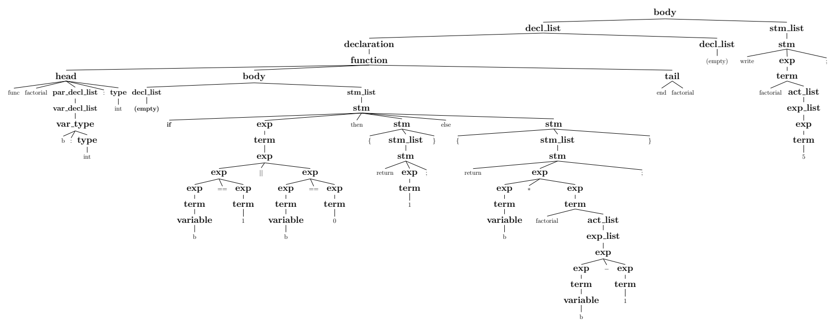


Figure 1: AST for a the Simple program.

0.1 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 1: Example of a valid program requiring multiple type passes.

```
var a : my_custom_int_type;
var b : a;
type my_custom_int_type = int;
b = 42;
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.

0.1.1 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.

0.1.2 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.

0.1.3 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.

0.1.4 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.

0.1.5 `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.

0.1.6 `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 2: 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.

0.1.7 `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.

0.1.8 `head` node

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

Figure 3: 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.

0.1.9 `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.

0.1.10 `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.

0.1.11 `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	id + int	str + str	int + id
	int + int			
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.

0.1.12 `statement` node

0.1.12.1 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.

0.1.12.2 if-statements

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

0.1.13 `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.

0.1.14 `variable` node

0.1.14.1 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.

0.1.14.2 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.

0.1.14.3 record access variable

$\langle$variable$\rangle$: $\langle$variable$\rangle$ **.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.

0.2 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
12	F_ShortCircuitAND	✓
13	F_ShortCircuitOR	✓
14	F_SimpleStructuralEquiv	✓
15	O_AbsoluteValueTest	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 number 11: Fails because a record is not a tuple but rather a set, where the ordering matters.

Comments to test number 15: : Fails during the parsing phase.

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	✓

0.3 Supplementary Tests

0.3.1 Function tests

0.3.1.1 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.	✓

0.3.1.2 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 14. The type checker will correctly yield an error on the input if the two return statements are flipped however.

Listing 2: function containing incorrect return statement.

```
func a () : int
    if (false) then{
       return 5;
    }else{
       return true;
    }
end a
```

0.3.2 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.	✓

0.3.3 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.	✓

0.3.4 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`	✓

1 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.

1.1 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.

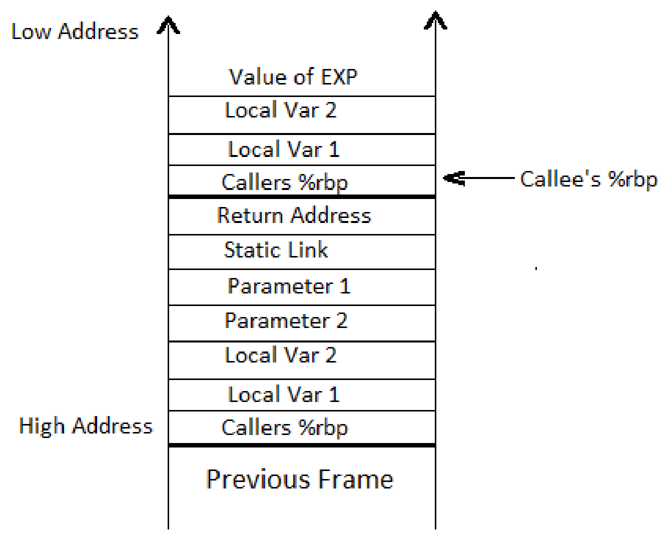


Figure 4: Stack Frame.

1.2 Code Templates

1.2.1 Template for declarations

1.2.1.1 Variable Declaration:

Listing 3: Variable Declaration.

```
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.

1.2.1.2 Function Declaration:

Listing 4: Function Declaration template.

```
func_start:
    pushq %rbp
    movq %rsp, %rbp

    code for variable declarations
    code for function body
func_end:
    movq %rbp, %rsp
    popq %rbp
    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.

1.2.1.3 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.

1.2.2 Template for statements

1.2.2.1 Return Statement:

Listing 5: Return statement template.

```
code for <expression>
pushq <exp>
popq %rax
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.

1.2.2.2 Write Statements:

Listing 6: Write statement template.

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

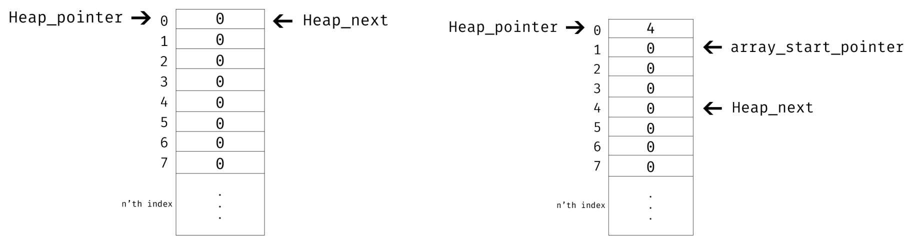
When writing to ”stdout” the ”printf” function is simply called with the right arguments.

1.2.2.3 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 5: Example heap with an array allocation of length 4.

`array_start_pointer` denotes the index assigned to the array.

1.2.2.4 Assignment Statement:

Listing 7: Assignment statement template.

```
code for <expression>
code for finding the right frame pointer and offset
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.

1.2.2.5 If Statement:

Listing 8: If statement template.

```
    code for <expression>
    cmp "<expression>—result", 1
    jne end_if
    code for <statement1>
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.

1.2.2.6 If Else Statement:

[caption=If-else statement template.]

```
    code for <expression>
    cmpq "<expression>—result", 1
    jne else_part
    code for <statement1>
    jmp end_if_else
else_part:
    code for <statement2>
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.

1.2.2.7 While Statement:

Listing 9: While statement template.

```
while_start:
    code for <expression>
    cmp "<expression>—result", 1
    jne while_end
    code for <statement>
    jmp while_start
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.

1.2.3 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.

1.2.3.1 Expressions:

Listing 10: Expression template.

```
code for <expression1>
"place result on stack"
code for <expression2>
"place result on stack"
"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.

1.3 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.

1.3.1 Functions

1.3.1.1 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.

1.3.1.2 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.

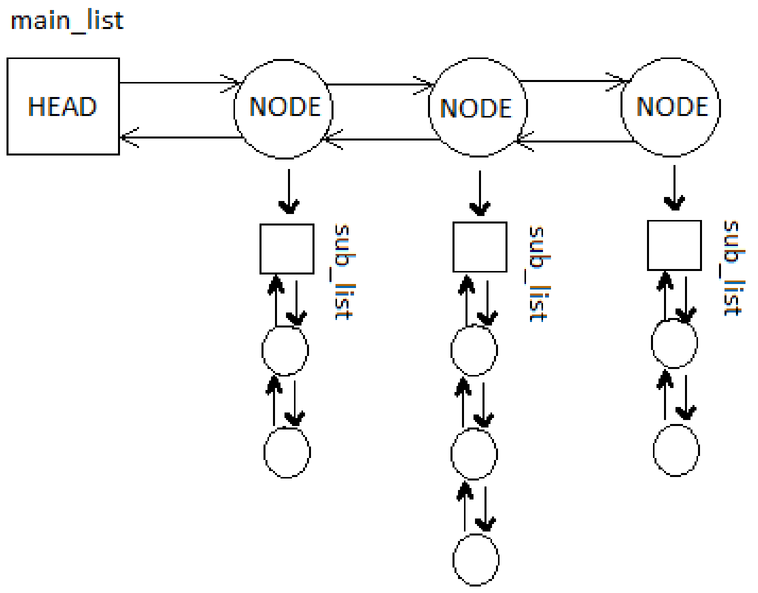


Figure 6: linked list of sub-linked lists that contain the code.

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.

1.3.1.3 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”.

1.3.1.4 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.

1.3.2 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.

1.3.2.1 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.

1.3.2.2 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.

1.3.2.3 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.

1.3.2.4 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.

1.3.3 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.

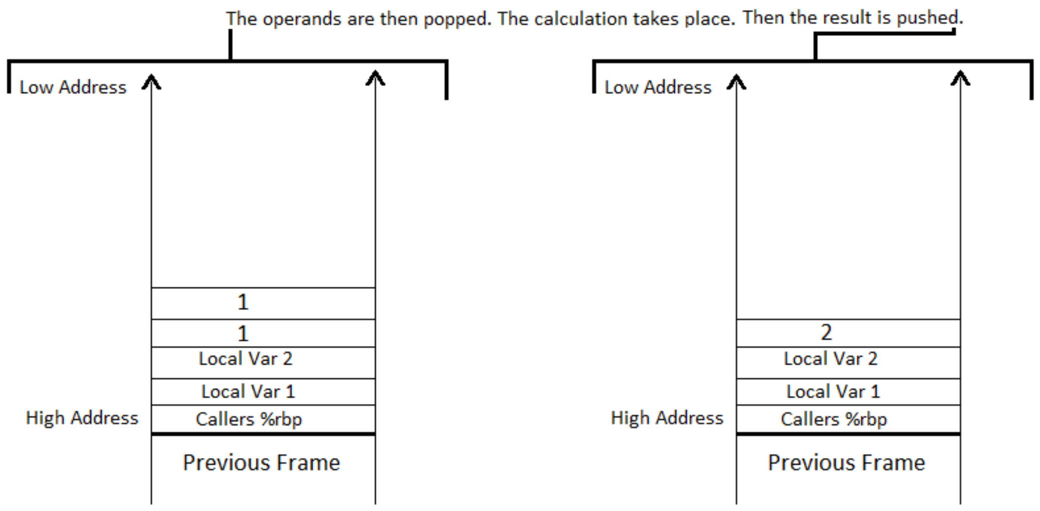


Figure 7: Stack Frame Expression Operation.

1.3.4 Term

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

1.3.4.1 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.

1.3.4.2 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.

1.3.4.3 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.

1.3.4.4 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.

1.3.4.5 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.

1.4 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

2 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.

2.1 Example Code

Listing 11: O_simple_string.src

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

Listing 12: 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 13: O_ArrayLength.src

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

Listing 14: 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 15: O_Factorial.src

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

write factorial(5);
```

Listing 16: Compiler emit of O_Factorial 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

factorial_0:
pushq %rbp
movq %rsp, %rbp

pushq $1
pushq 24(%rbp)
popq %rbx
popq %rax
cmpq %rbx, %rax
je TRUE_0
pushq $0
jmp END_0
TRUE_0:
pushq $1
END_0:

pushq $0
pushq 24(%rbp)
popq %rbx
popq %rax
cmpq %rbx, %rax
je TRUE_1
```

```
pushq $0
jmp END_1
TRUE_1:
pushq $1
END_1:

popq %rbx
popq %rax
orq %rbx, %rax
pushq %rax

popq %rax
movq $1, %rbx
cmpq %rax, %rbx
jne else_part_0
pushq $1
popq %rax
jmp end_factorial_0
jmp end_if_else_1
else_part_0:
pushq $1    #push param for called function
pushq 24(%rbp)
popq %r11
popq %rbx
subq %rbx, %r11
pushq %r11      #end of sub expression
movq 16(%rbp), %r15
push %r15
call factorial_0
addq $8, %rsp    #discard parameter
addq $8, %rsp    #discard parameter
pushq 24(%rbp)
movq %rax, %rbx
popq %rax
imulq %rbx, %rax
pushq %rax
popq %rax
jmp end_factorial_0
end_if_else_1:
end_factorial_0:
movq %rbp, %rsp
popq %rbp
ret

main:
pushq %rbp
movq %rsp, %rbp
pushq $5    #push param for called function
pushq %rbp     #push address of static link
call factorial_0
addq $8, %rsp    #discard parameter
addq $8, %rsp    #discard parameter
movq %rax, %rbx     #moving return value in rax to rbx
leaq printf_format_int(%rip), %rdi
movq %rbx, %rsi
movq $0, %rax
call printf
```

```
movq %rbp, %rsp
popq %rbp
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.

3 Conclusion

In this work, we present a working compiler. We briefly explained the tools bison and flex and then introduced our grammar. We then took the grammar in bison in which we then built a parser and used it to parse source code. We then created a weeder and a type-checker in which we wanted to ensure program correctness to a certain degree. We explored why it wasn't possible to fully guarantee program correctness from a compiler standpoint. Moreover, we successfully created a working type-checker. We explored code generation and the working templates that can be used to emit sequential assembly code from valid programs. We laid the foundation for code generation and managed to emit working assembler code that can be compiled and ran on 64-bit Linux machines.

We took a look at the challenges we faced along the way and the extensions that we've created to improve upon the compiler. We laid the foundations for peephole optimization to be able to optimize the emit code drastically in the future. All in all, we managed to explore the whole ecosystem of a compiler from start to finish. We did face issues along the way and had to resolve them. We presented and showed by testing our compiler in various ways along the way and managed to get a clear picture of where our implementation is currently at.