A Compiler for an Imperative Programming Language

William Bundgaard
211000

Andreas Rosenstjerne Hansen
220298

Supervisor
Jakob Lykke Andersen
jlandersen@imada.sdu.dk

A thesis presented for the degree of
Bachelor of Computer Science

Department of Mathematics and Computer Science
University of Southern Denmark, SDU
Date: June 1, 2023

Contents

1 Introduction

In the sphere of computer science, a broad spectrum of programming languages exists, each with its unique set of functionalities. These functionalities are tailored to enchance and streamline the us of distinct programming paradigms such as imperative, functional, object-oriented, and generic programming. The design of these languages is carefully calibrated to serve specific purposes and solve unique challenges in the realm of programming.

A multitude of techniques can be deployed for implementing these languages e.g. translation into machine code and just-in-time (JIT) interpretation. These techniques greatly impacts the execution efficiency and compatibility of the programming language.

The primary objective of this project is to go deeper into the intriguing process of building a programming language from scratch, concentrating on the design and development of a compiler. This serves as the bridge between the high-level programming language and the low-level assembly or machine code that can be executed by the computer's hardware.

In order to accomplish this objective, we will write an imperative programming language and a corresponding compiler, which will convert our language instructions into x86-64 assembly code. The initial phase of the project will be focused on the development of a fully functional compiler that can handle a minimalistic subset of our language. This is to ensure that the foundational aspects of the language and compiler are reliable before additional features are implemented.

The initial version of the language will incorporate the most fundamental data types, including integers, booleans, arrays, along with a mechanism for defining custom types through a form of classes. Additionally, we will implement automatic memory allocation to abstract the memory management from the user, making the language easier to use. For booleans and integers, we will incorporate the most widely used basic logical and arithmetic operators to enable the users to manipulate these data types effectively. Basic control structures such as "if" conditions and "while" loops will be implemented to provide decision-making and iterative capabilities to our language. Furthermore, we will support nested functions to offer a more organised and modular programming environment, contributing to the readability and maintainability of the programs written in our language.

This report will cover the following topics:

1. Definition of the "Leif" programming langauge, with it accompanying grammar.
2. Construction of the abstract syntax tree (AST)
3. Building symbol tables to keep track of user elements, scopes, and other aspects
4. Type checking, where correct usage of the program's types is verified by traversing the AST multiple times.
5. Code generation
6. Liveliness analysis and register allocation, where as many memory accesses as possible are attempted to be removed by placing data in registers.
7. Peephole optimisation, in which unnecessary code is removed.
8. Emitting of the final assembly code

1.1 Related works

Our compiler is related to the compiler, SCIL "A Simple Compiler in a Learning Environment", written by Kim Skak Larsen.

We have been given permission to use SCIL in our own work, and have based some of our design choices on that of SCIL. This includes the structure of an AST, a visitor pattern to access said AST, printing of error messages, and some object design choices for both a symbol table and for intermediate code.

We have designed the language ourselves, and only used ideas from SCIL when we found that they were similar to what we were trying to do, or when a design choice we thought was smart.

As a result, our compiler's structure shares similarities with SCIL but is considerably larger, as it encompasses many features that SCIL does not provide.

2 Functionalities

Our objective is to establish what we refer to as a "minimum language". By this, we mean a compact language equipped with just the necessary functionalities to execute the majority of algorithms in a feasible manner. This includes the two simple types, integers and Booleans, along with the customary arithmetic, logical, and comparison operations applicable to them. For integers, operations such as addition, subtraction, multiplication, division, and modulus will be included, along with comparison operators for equality, inequality, and lexicographical ordering. For Booleans, we will have the logical "and" and "or" as well at "not", with only equality and inequality operators for comparison.

We also need variables to store data in. Control structures such as "if", "if-else", "while" and "break" statements. We also need functions, that can be nested, comprising both parameters and return types. It is also essential to support arrays of any type, and structures of any type. We also want nested scoping with local variables, functions and structures inside. We need to be able to assign to variables, parameters, array indices, and class attributes. Ultimately, we also want to include a print statement to observe our programs in action, producing observable output.

These are all functionalities we need in our "minimum language"

2.1 Specifications

Here we will additional specifications for the require components.

- **Types:**
  - **Simple types:**
    - **Integers:**
      These represent whole numbers and can be stored without the need of heap memory. Denoted by the keyword "int" for declarations.
    - **Booleans:**
      These embody the logical "true" and "false" and can be stored without the need of heap memory. Denoted by the keyword "bool" for declarations.

    - **Null:**
      The null type is solely intended for functions that return nothing. It will essentially be the type equivalent of nothing.
  - **Non-simple types:**
    - **Classes (structures):**
      Classes should be declared at the beginning of a scope. They can be identified by the keyword "class" followed by the assigned name and any number of variable declarations encased within curly brackets. Any number of classes can be declared in any scope. within this scope, and all the child scopes, the classes will function as types and can be declared using its assigned name as a keyword. An instance of the class can be created by using the keyword "new", followed by the class name. To access the class attributes, the dot (".") operator, followed by the attribute name, can be used on the class instance to either get or to assign to the attribute inside the class instance.
    - **Arrays:**
      Arrays can be of any type and is declared using any type name (both simple- and non-simple-, as well as other array types) followed by empty brackets. To create an array, the keyword "new" can be used followed by the type name of the array elements and a pair of bracket containing a non-negative integer expression. Arrays can be indexed into by adding a set of brackets with an integer expression inside, for both retrieving the value from and assigning to the index.

- **Variables:**
  Variables should be declared at the beginning of the scope, right after class declarations. Variables can be identified by the keyword "var" preceding them followed by the variable's type and then the variable's name. If more variables of the same type are needed, the user can add a comma after the first variable name and add a new variable name. This can be repeated as needed. Variable types can be any of the simple or non-simple types, including classes declared in the same or a parent scope. All variables will be initialised to 0 (Boolean's will be set to false). For non-simple type variables, they still need be assigned before use. If utilised in any expression (except assignment), it turns into an expression of its own type.

- **Operators:**

We require the following list of expression operators:

- **Arithmetic:**
  Arithmetic operators consists of "+" (addition), "−" (subtraction), "/" (division), "·" (multiplication) and "%" (modulus). These are all binary operators, requiring two integer expressions (one on either side) and returning an integer expression. "-" also functions an unary operator, requiring only one integer expression after it and returning an integer expression.
- **Logic:**
  The logic operators include "&&" (and), "||" (or) and "!" (not). The first two are binary operators that require two Boolean expressions and are themselves Boolean expressions. The last is a unary operator that requires a singular boolean expression after is and returns a boolean expression.
- **Comparison:**
  Comparison operators consist of "==" (equal), "≠" (not equal), "<" (less than), ">" (greater than), "≤" (less or equal) and "≥" (greater or equal). All are binary operations, and require two expressions on either side. The first two works with both integer and boolean expression (both sides must have the same type). While the rest only works with integer expressions. All of these are boolean expressions.
- **Parentheses:**
  Parentheses works by grouping expressions together so that everything within the parentheses must be evaluated in full before it is used outside the parentheses.
- **Indexing:**
  Indexing only works on expressions of some array type. The array expression comes first, followed by brackets with an integer expression inside. The integer expression will be evaluated and the result will be the index of the array that is returned. The expression type of this will be the type of the array's elements. Any indexed element also functions as a variable for assignment.
- **Dot:**
  The dot operator works on any expression of a declared class type. The class expressions comes first, followed by the "." operator, and then the name of the desired attribute from that class. This will access that attribute from the class instance. The expression type

will be the type of the attribute. Any attribute accessed using the dot operator also functions as a variable for assignment.

- **Functions:**
  Functions should be declared immediately after all variable declarations within a scope. They can be recognised by the keyword "function", followed by a return type, and then the function's name. The function's parameters (if any) can then be listed within parentheses, specifying their type and name in that order. There can be as many parameter as desired, separated by comma. The body of the functions then follow, which is a scope where new declarations can be made. The unique aspect of this scope is that the function's parameters are usable as variables and will be initialised to whatever values they are assigned during the function call. If the functions return type is "null", then there should be no return statement within the function. Otherwise, a return statement should be the very last statement in the function. This statement cannot be nested within any other scope or statement. The return statement must contain an expression of the functions return type. This expressions value it then returned to caller.

- **Function calls:**
  A function can be called by giving its name followed by a comma separated set of argument in parenthesis. The number of arguments should mach the given number of parameters for the function declaration, in that order, and must have the respective type of the parameter. This is an expression with the same type as the function return type.

- **Nested scopes:**
  A nested scope is a special statement. It is recognised by a whole new set of declarations and other statements encapsulated by curly brackets. Any declarations inside this nested scope will not be usable outside this scope. It functions as a whole new body.

- **Control structures:**
  Here are the control structures we want in our language:

  - **If:**
    The if statement is recognised by the "if" keyword followed by a boolean expression in parentheses, and then any statement. If the boolean expression evaluated to "true", then the nested statement will be executed, otherwise it will be skipped.

  - **If else:**
    The if else statement is recognised by the "if" keyword followed by a boolean expression in parentheses, then any statement, followed by the "else" keyword and then another statement. If the boolean expression evaluated to "true", then the first nested statement will be executed and the second skipped. Otherwise, the first will be skipped and the second executed.
  - **While:**
    The while statement is recognised by the "while" keyword followed by a boolean expression in parentheses, and then any statement. If the boolean expression is evaluated to "false" the nested statement will be skipped. Otherwise, it will be executed, and then control will return to the start of the while statement, and the process will be repeated.
  - **Break:**
    A break statement is recognised by the "break" keyword. Its sole function is to halt the innermost while loop it is contained within. When this occurs, any unexecuted statements within the while loop are skipped and control moves to the point right after the while loop. Break may only be used inside of a while loop.
  - **Return:**
    The return statement is recognised by the "return" keyword, followed by an expression of the same type as the function's return type.

- **Assignment:**
  Assignments are identified by a variable followed by an "=" and then an expression with the same type as the variable type. This operation assigns the value of the expression to the variable.

- **Print:**
  The print statement is identified by the keyword "print" followed by either an integer or boolean expression enclosed in parentheses. This operation outputs the value of the expression to the terminal when the program is executed.

3 Parsing and Abstract Syntax Tree Generation

For this project, we've utilised the Python libary PLY, or Python Lex-Yacc that provides a way to specify and recognise patterns in text. It has two modules, lex.py and yacc.py, both located in the ply package. The lex.py is responsible for the lexical analysis, where input text is broken down into tokens using regular expression rules. It offers an interface in the form of a token() function, which outputs the next valid token from the input stream. Yacc.py, on the other hand, handles syntax parsing based on a context-free grammar. It calls lex.py's token() function to retrieve tokens and implement grammar rules. The output is usually an Abstract Syntax Tree (AST), but this is user-defined and can vary according to specific needs. It can also be used in constructing one-pass compilers.

3.1 Grammar

Before developing the parser, we created our grammar for recognising our language. The language Leif is fully described by the grammar provided below. The grammar has 41 terminals and 38 non-terminals all combined in 86 rules.

Most of the terminal symbols are capitalised, except a few special ones, and will all directly evaluate into a string. The majority of these terminal symbols can only evaluate to a single string. However, a select few evaluate to a regex formula to recognise multiple strings of the same type. This includes the INT and BOOL which evaluate to a whole number and a boolean "true" or "false" respectively. IDENT evaluate to any name consisting of letters from the English alphabet, numbers and underscores, although they can't start with a number. Both COMMENT and COMMENTBLOCK identify comments in the language. The COMMENT recognises any string starting with a hashtag (not immediately followed by an asterisk) and ending with a new line (new line not included). The COMMENTBLOCK recognises a hashtag followed immediately by an asterisk, then any string of any symbols, including new lines until and including the next instance of an asterisk immediately followed by a hashtag. Then we have 'empty' that recognises the empty string and finally the 'newLine' that identifies new line symbols. Through all of this, although spaces and tabs function as separators between tokens, they are ultimately disregarded. Here are all the terminal symbols for the grammar:

```
IF -> "if"
ELSE -> "else"
WHILE -> "while"
FUNCTION ->"function"
VAR -> "var"
BREAK -> "break"
PRINT -> "print"
CLASS -> "class"
RETURN -> "return"
NEW -> "new"
_INT -> "int"
_BOOL -> "bool"
NULL -> "null"
IDENT -> [a-zA-Z_][a-zA-Z_0-9]*
INT -> \d+
BOOL -> "true" | "false"
PLUS -> "+"
MINUS -> "-"
MULTIPLY -> "*"
DEVIDE -> "/"
MODULO -> "%"
EQ -> "=="
NEQ -> "!="
LT -> "<"
GT -> ">"
LTE -> "<="
GTL -> ">="
AND -> "&&"
OR -> "||"
NOT -> "!"
LPAREN -> "("
RPAREN -> ")"
LCURL -> "{"
RCURL -> "}"
LBRAC -> "["
RBRAC -> "]"
ASSIGN -> "="
COMMA -> ","
SEMICOLON -> ";"
DOT -> "."

empty
  -> ""

newLine -> \n+

COMMENT -> \#.(?!\*).*

COMMENTBLOCK -> \#\*(((?!\*\#).)|\n)*\*\#
```

For the non-terminal symbols, have been classified up into three main categories. The overall scope structure, statements, and expressions. All the non-terminal symbols are presented in lower caps. We start with the overall scope structure.

The rule for the **program** is our starting point. This evaluates to a **body**. Essentially, a body represents a scope and each unique scope will possess a body. The body is composed of four key components, which are presented in order. These include class declarations, variable declarations, function declarations and finally, a list of all the statements for the scope. Each type of declaration is optional individually, and an arbitrary number of each can be declared, provided they adhere to the specified order. The list of statements, however, is not optional and must contain at least one statement. Similar to declarations, there is no limit to the number of statements that can be created.

The **class declarations** initiate the body. Class declarations are made using the "class" keyword, followed by an identifier for the class name. Inside curly brackets, users can declare an arbitrary number of attributes within each class, same way variables would be declared for a body.

Next we have the **variable declarations**. Variables are declared by the "var" keyword followed by a variable type and then one or more identifiers for the names of variables with that type. Variable declarations are separated by semicolons. The variable types may be any of the declared class names, hence the ordering.

The **function declarations** are declared with the "function" keyword, followed by a return type name, then an identifier for the function name. Then, within parentheses and separated by commas, is a list of all the function's parameters, and finally, a new body enclosed within curly brackets. The parameters iEach parameter is declared by a parameter type and then an identifier for the parameter name.

Finally, the body contains a **statement list**. This list must contain at least one statement, beyond that there is no limit to how many statements can be declared.

The grammar for the overall scope structure can be seen here:

```
program
  -> body

body
  -> optional_class_declaration_list
      optional_variables_declaration_list
        optional_functions_declaration_list
          statement_list

optional_class_declaration_list
  -> empty
   | class_declaration_list

class_declaration_list
  -> class_declaration
   | class_declaration class_declaration_list

p_class_declaration
  -> CLASS IDENT LCURL variables_declaration_list RCURL
```

```
optional_variables_declaration_list
  -> VAR variable_type variables_list SEMICOLON
   | VAR variable_type variables_list SEMICOLON variables_declaration_list

variable_type
  -> _BOOL
   | _INT
   | NULL
   | IDNET
   | variable_type LBRAC RBRAC

variables_list
  -> IDENT
   | IDENT COMMA variables_list

optional_functions_declaration_list
  -> empty
   | functions_declaration_list

functions_declaration_list
  -> function
   | function functions_declaration_list

function
  -> FUNCTION variable_type IDENT LPAREN
       optional_parameter_list RPAREN LCURL body RCURL

optional_parameter_list
  -> empty
   | parameter_list

parameter_list
  -> variable_type IDENT
   | variable_type IDENT parameter_list

statement_list
  -> statement
   | statement statement_list
```

The **statements** that define the functionality of a body are diverse, each

with their unique structure. When a "statement" appears on the right hand side of a rule, this can be evaluated to any of the 9 statements. Most of the simple statements that don't evaluate directly into new statements end with a semicolon, denoting the conclusion of a statement.

The **return** statement is meant to be used inside the body of a function. It uses the "return" keyword, followed by an expression that will be returned, and ends with a semicolon.

The **print** statement uses the "print" keyword, requires an expression to be printed in parenthesis, concluded with a semicolon.

The **assignment** statement is unique as it's meant to assign new values to variables or parameters. It takes any variable on the left side of an equal sign, followed by an expression whose value will be assigned to the variable.

A **variable** can be be either a standard variable identifiers, an attribute derived from an expression, or an index of an expression. Attribute are noted with an expression, followed by a dot, and then the identifier of the attribute. An index uses an expression with another expression in brackets. These variables will also be used in the expression rules, and can themselves be evaluated as an expression.

The **if then else** statement uses the "if" keyword, then an expression in parenthesis, followed by any statement, the "else" keyword, and a new statement.

The **if then** statement is like the if then else statement, except the "else" keyword and the last statements is omitted.

The **while** statement has a similar structure to the if then statement, except is replaces the "if" keyword with "while".

The **break** statement is simply the "break" keyword followed by a semicolon. This is meant to exit a while loop.

The **expression** statement allows a statement to be evaluated directly to an expression. Particularly useful, as the function call is an expression in this context. This statement is also ends with a semicolon.

Then, the most interesting statement, the **compound statement**. Essentially acts as a new body encapsulated within a pair of curly brackets. This allows the user to create new scope at will instead of limiting them to simple statement. This will be very useful in if-then statements, if-then-else statement and in while statements, as it enables the declaration of additional declarations or statements within them. For the while statement, when using

the compound statement as the nested statement, it offers a more versatile use of the break statement, which would otherwise be redundant.

Here is the grammar rules for all the statements:

```
statement
  -> statement_return
   | statement_print
   | statement_assignment
   | statement_ifthenelse
   | statement_ifthen
   | statement_while
   | statement_compound
   | statement_break
   | statement_expression

statement_return
  -> RETURN expression SEMICOLON

statement_print
  -> PRINT LPAREN expression RPAREN SEMICOLOM

statement_assignment
  -> variable ASSIGN expression PRAREN SEMICOLON

variable
  -> IDENT
   | expression DOT IDENT
   | expression LBRAC expression RBRAC

statement_ifthenelse
  -> IF LPAREN expression RPAREN statement
       ELSE statement

statement_ifthen
  -> IF LPAREN expression RPAREN statement

statement_while
  -> WHILE LPAREN expression RPAREN statement

statement_break
```

```
  -> BREAK SEMICOLON

statement_expression
  -> expression SEMICOLON

statement_compound
  -> LCURL body RCURL
```

There are nine distinct types of expressions. Any **expression** appearing on the right hand side of a rule can be evaluated to one of these nine types.

The **integer** expression just evaluates to any whole number, also know as an integer.

The **boolean** evaluated to either "true" or "false".

The **neg** expression uses the exclamation mark sign as a NOT symbol, followed by a new expression. This is meant to be used on boolean expression, negating the expression's value

The **negative** expression uses the hyphen sign, or a minus sign, followed by another expression. This is meant to be used on integer expression, negating their value.

The **identifier** expression is any variable (as described under the statement rules). They are meant to get the value from the variable.

The **call** expression represents a function call. it requires the function name, followed by by an optional expression list in parenthesis for the function's parameters. This is meant to invoke the respective function and retrieve the return value, if any.

The **optional expression list** can be evaluated to either nothing or to an expression list.

The **expression list** is a comma separated list of at any number of expressions, with at least one expression. These are meant to serve as the argument values to give to a function in a function call. There should be as many expressions as the called function requires. If the function takes no arguments, then the list should not be declared, and the optional expression list should evaluate to nothing.

The **binop** expression can be evaluated to one of 13 different binary operators in between two other expressions. Five of these are meant for arithmetic operation on integer expressions. Two are logical operators meant for boolean

expressions. Two are equality operators meant for both integer and boolean expressions (should not mix booleans an integers). The remaining four are comparison operators meant for integer expressions.

The **group** expression is essentially an expression enclosed in parenthesis. This ensures that whatever the nested expression's evaluation isn't unintentionally mixed with any external expression components. The nested impression will be evaluated independently from the outside.

Finally, the **new** expression. This takes a variable type to make a new installation of. This variable type should be one of the declared classes accessible inside the scope. Alternatively, the new expression can take a variable type name, followed by another expression enclosed in brackets. This is meant to create an array of the specified type, with a length given by the nested expression. The nested expression should be an integer expression.

Here is the grammar rules for all the expressions:

```
expression
  -> expression_integer
   | expression_boolean
   | expression_identifier
   | expression_call
   | expression_binop
   | expression_group
   | expression_neq
   | expression_negative
   | expression_new

expression_integer
  -> INT

expression_boolean
  -> BOOL

expression_neg
  -> NOT expression

expression_negative
  -> MINUS expression

expression_identifier
```

```
  -> variable

expression_call
  -> IDENT LPAREN optional_expression_list RPAREN

optional_expression_list
  -> empty
   | expression_list

expression_list
  -> expression
   | expression COMMA expression_list

expression_binop
  -> expression PLUS expression
   | expression MINUS expression
   | expression MULTIPLY expression
   | expression DEVIDE expression
   | expression MODULO expression
   | expression EQ expression
   | expression NEQ expression
   | expression LT expression
   | expression GT expression
   | expression LTE expression
   | expression GTE expression
   | expression AND expression
   | expression OR expression

expression_group
  -> LPAREN expression PAREN

expression_new
  -> NEW variable_type
   | NEW variable_type LBRAC expression RBRAC
```

3.2 Lexer

The lexer is an essential component of a compiler, responsible for transforming a source code into a stream of tokens that can be processed by the parser.

Its main job is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. Each token is a meaningful character string, such as a keyword, an identifier, or an operator.

The lexer processes the source code by identifying various tokens, including reserved words, identifiers, integers, booleans, arithmetic operators, comparison operators, and other special characters such as parentheses, brackets, and dots.

These tokens are defined using regular expressions to match specific patterns in the input source code. For example, the following code snippet shows the definition of an integer token.

to create a lexer using PLY, we need to use the **lex.lex()** function, which utilises Python's reflection or introspection feature to read the regular expression rules from the calling context to construct the lexer. Once we have constructed the lexer, we can control it using two primary methods, **lexer.input(data)** and **lexer.token()**

```
1  def t_INT(t):
2      r'\d+'
3      try:
4          t.value = int(t.value)
5      except ValueError:
6          error_message("Lexical Analysis",
7                        f"Integer value too large.",
8                        t.lexer.lineno)
9      if t.value > int('0x7FFFFFFFFFFFFFFF', 16):
10         error_message("Lexical Analysis",
11                       f"Integer value too large.",
12                       t.lexer.lineno)
13     return t
```

Looking at the above code, the regular expression **r'\d+'** is used to match any sequence of digits, which represents an integer value. The function then converts the matched string to an integer and checks if it exceeds the maximum allowed value, raising an error if needed.

When it comes to ignored characters and comments, the lexer is designed

to ignore certain chars, such as whitespace and comments, which has no impact on the execution of the program. The lexer also handles single-line and multi-line comments using the following code snippets:

```
def t_COMMENT(t):
    r'\#(?!\*).*'
    pass

def t_COMMENTBLOCK(t):
    r'\#\*(((?!\*\#).)|\n)*\*\#'
    t.lexer.lineno += t.value.count("\n")
```

The lexer has error handling to manage the cases where illegal characters or patterns are encountered in the source code. Here the function **t_error** is responsible for displaying an error message and skipping invalid chars.

```
def t_error(t):
    error_message("Lexical Analysis",
                  f"Illegal character '{t.value[0]}'.",
                  t.lexer.lineno)
    t.lexer.skip(1)
```

3.3 Parser (Yacc)

Parsing, the second phase of our compiler, is a crucial process in the compilation of a program. It takes as input the tokens produced during the lexical analysis stage and constructs an intermediate representation of the source program, typically in the form of a parse tree or an abstract syntax tree (AST). In the context of Leif, the Python module **yacc** is used to generate the parser. It uses a set of grammar rules defined in BNF, which is a formal way to express context-free grammars. a rules could like this.

```
<symbol> ::= <symbol-sequence>
```

To construct an Abstract Syntax Tree, **yacc** and **ply.yacc** work through a bottom up parsing strategy (specifically, LALR parsing). The grammar rules defined in the ply.yacc parser have associated actions that are executed when a rule is recognised. The parser rules create instances of different classes from the **AST** module, which represents nodes in the AST. The arguments to thses constructors are of expression **(t[i])**.

For example, in the rule:

```
def p_expression_binop(t):
    '''expression_binop : expression PLUS expression
                        | expression MINUS expression
                        | expression MULTIPLY expression
                        | expression DIVIDE expression
                        | expression MODULO expression
                        | expression EQ expression
                        | expression NEQ expression
                        | expression LT expression
                        | expression GT expression
                        | expression LTE expression
                        | expression GTE expression
                        | expression AND expression
                        | expression OR expression'''
    t[0] = AST.expression_binop(t[2], t[1], t[3],
        t.lexer.lineno)
```

The **expression_binop** node is then created in the AST when the parser recognizes a binary operation. The operator **t[2]** and the left and right operands (**t[1]** and **t[3]**) are passed to the **expression_binop** constructor. The line number is also passed for potential error reporting or debugging purposes. One should also note that the creation of the AST is entirely dependent on these actions associated with the grammar rules. The grammar itself only describes the structure of the language but doesn't inherently create an AST. The AST is built by the programmer-defined actions that create and connect the AST nodes when grammar rules are recognized by the parser.

Each grammar rule is defined by a docstring, that contains the appropriate notation. The function's body contains the action to perform when ever the rule is applied.

For instance, if we have a grammar rule for an arithmetic expression like this.

```
1 def p_expression_plus(p):
2     'expression : expression PLUS term'
3     p[0] = p[1] + p[3]
```

This rules states that an expression can be an expression followed by the "plus" token and a term. if this pattern is found in the token stream, that action is then to add the value of the first expression and the term. The result of this action then becomes the value of this expression.

There are some rules the code has to follow such as precedence- and Associativity. In this code, the precedence and associativity rules are used to guide the evaluation of complex expressions and statements. For example, the 'AND' and 'OR' tokens have left associativity and the 'NOT' token has right associativity. This means that in an expression with multiple 'AND' or 'OR' operators, the operations will be carried out from left to right. However, for expressions with multiple 'NOT' operators, the operations will be done from right to left.

Precedence rules in a parser determine how expressions involving more than one operator are evaluated. They dictate the order in which operations are carried out, helping avoid ambiguity in expressions. For example, in the mathematical expression $2 + 3 \cdot 4$, the multiplication operation is done first due to its higher precedence, and then addition is performed on the result. We have defined the precedence in this code using a tuple of tuples, where each internal tuple represents a precedence level, starting from the lowest. Each tuple contains two elements: associativity and a list of tokens at that level.

```
precedence = (
    ('left', 'LPAREN', 'RPAREN'),
    ('left', 'IF', 'ELSE'),
    ('left', 'AND'),
    ('left', 'OR'),
    ('right', 'NOT'),
    ('right', 'EQ', 'NEQ', 'LT', 'GT', 'LTE', 'GTE'),
    ('left', 'MODULO'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MULTIPLY', 'DIVIDE'),
    ('right', 'NEW'),
    ('right', 'LBRAC'),
    ('right', 'ASSIGN'),
    ('left', 'DOT')
)
```

The precedence tuple starts with the lowest precedence level. For instance, parentheses 'LPAREN', 'RPAREN' have the lowest precedence, and the dot operator 'DOT' has the highest. The tokens within the same internal tuple have the same level of precedence.

While we have the precedence rules decide the order of operations, associativity rules decide the order in which operations with the same precedence are executed. he associativity can be 'left' or 'right'. 'left': The operators are evaluated from left to right. For example, in the expression $a - b - c$, 'a' is subtracted from 'b' first, then 'c' is subtracted from the result. 'right': The operators are evaluated from right to left. For example, in the expression $a = b = c$, 'c' is assigned to 'b' first, then 'a' is assigned the value of 'b'. In the precedence definition, the first element of each tuple represents the associativity for that level.

3.4 AST Generation

We have defined several nodes for an abstract syntax tree (AST) that should cover most non-terminal symbols of the grammar. Combined with with the grammar, our associativity rules, lexer, and parser, we have issued specific

instructions to generate nodes for the tree for most reductions of the grammar rules.

3.5 The nodes

The creation of the tree occurs during the lexing and parsing of the language. Every node will be created alongside its relevant child nodes and any specific values, like integer values or variable names, if applicable. Each node is also given the line number for when the reduction rule from the grammar could be made.

Here we have a list of all the different node types of the tree with a description for each:

- **body:**
  The body (or scope) holds a list of class declarations, a list of variable declaration, a list of function declaration, and a list of statements. Class, variable, and function declaration lists can be None.
- **class_declaration_list:**
  This functions as a linked list, holding both a class declaration and the subsequent link in the class declaration list, if present.
- **class_declaration:**
  Contains the name of the declared class as well as a variable declaration list for its attributes.
- **variables_declaration_list:**
  Functions as a linked list, holding both a variable type name, a list of variable names, and the subsequent link in the variable declaration list, if present.
- **functions_declaration_list:**
  This functions as a linked list, holding both a function declaration and the subsequent link in the function declaration list, if present.
- **function:**
  Contains its return type name, its name, its parameter list (which can be None), and lastly, the body of the function. It should be noted that when the program as a whole is created, it is stored in the AST as a function called **main**. We refer to this as the main function.

- **parameter_list:**
  Functions as a linked list of parameters. It stores the parameter type name and the parameter's name, and then a link to the next parameter, if any.
- **variables_list:**
  This is a linked list of variable names, only storing the name of the variable.
- **statement_return:**
  Only holds the expression to be returns, and some meta parameters for later use.
- **statement_print:**
  Only holds the expression to be printed, and a meta parameters for later use.
- **statement_assignment:**
  This has a left hand side consisting of a variable and a right hand side of some expression.
- **statement_ifThen:**
  Contains an expression and a statement.
- **statement_ifThenElse:**
  Contains an expression and two separate statement. One for if the expression is true, and one for if it is not.
- **statement_while:**
  Holds an expression and a statement to repeat until the statement is false.
- **statement_break:**
  Essentially empty.
- **statement_list:**
  A linked list of statements, storing only a singular statement and a link to the next one.
- **expression_integer:**
  Stores the value of the integer. Stores its expression type as "int".
- **expression_negative:**
  Contains an expression to negate. Stores its expression type as "int", since it is only supposed to be used on integer expressions.

- **expression_boolean:**
  Stores the value of the boolean. Stores its expression type as "bool".

- **expression_neg:**
  Contains an expression to negate. Stores its expression type as "bool", since it is only supposed to be used on boolean expressions.

- **expression_index:**
  : Contains an expression to index into, and an expression to be used as the index. It also holds some metadata whether it is to be used for an assignment or as an expression since it can serve as both. The meta element will be filled out later.

- **variable:**
  Contains the name of the variable. Also holds some metadata for whether is is to be used for an assignment or as an expression since it can serve as both. The meta element will be filled out later.

- **dot_variable:**
  This holds an expression from which to get an attribute from as well as the attribute's name. It also includes some metadata, determining whether it should be used for assignment or as an expression, since it can serve both functions. The metadata will be completed at a later stage.

- **expression_call:**
  Contains the name of the function to call, and an optional list of arguments to use when calling the function.

- **expression_binop:**
  This node holds an expression for the left hand side and one for the right hand side. It also contains the operator that is applied to these two expressions. It uses this operator to determine its own expression type and what expression types can be used as input. This is used later and currently, it serves no immediate function.

- **expression_new:**
  This node holds the name of the class for creating a new instance. Its expression type is stored as the name of the instantiated class type.

- **expression_new_array:**
  This node holds the name of the type or class for creating a new array, as well as an expression to calculate array's size. Its expression type is stored as an array of the specified type.

- **expression_list:**
  This represents the list of arguments used in a function call. It functions as a linked list and stores the expression for the argument and a link to the following one, if any.

For any expression whose type was not named, it does hold a meta value for a type, however it is not known at the moment, and will be filled during the type checking process. Any expression node that already knows its type can fill this placeholder at this stage, but it will also only be used during type checking.

3.6 Visitor pattern

To access all the nodes in the AST in a orderly and simple manner, we've opted to use a visitor pattern. Every node class in the AST has a function called **accept** that takes a visitor as argument. The accept function will then call accept with the same visitor recursively on each child node within it, allowing the visitor to traverse the tree in a structured manner by only calling accept on the root node of the AST with that visitor.

Within every node, we invoke the relevant visitor functions with the node as its argument. We mainly do this before, after, and in between ever accept call of children nodes, so that the visitor will be able to do work both before and after every child is visited.

The given visitor must inherit from the **VisitorsBase** class, for it to work with the AST.

The code for the visitor base is quite short and is given here:

```
class VisitorsBase:

    def _visit(self, t, s):
        method = getattr(self,
            s + "_" + t.__class__.__name__,
            None)
        if method:
            method(t)

    def preVisit(self, t):
        self._visit(t, "preVisit")

    def preMidVisit(self, t):
        self._visit(t, "preMidVisit")

    def midVisit(self, t):
        self._visit(t, "midVisit")

    def postMidVisit(self, t):
        self._visit(t, "postMidVisit")

    def postVisit(self, t):
        self._visit(t, "postVisit")
```

This base class manages the various calls to the visitor functions. Every visitor then has the opportunity to have a **preVisit**, **preMidVisit**, **midVisit**, **postMidVisit** and **postVisit** function for every node in the tree. This is done by creating a function with one of the visitor function names, followed by an underscore, and then the name of the AST node. This function should accept the node as an argument. For instance, if we want to create a visitor function for a mid visit in the **body** node, we would create a function called **midVisit_body(self, t)**. We do not call every possible visitor function name in every node in the AST, only the ones considered relevant are called. If a specific visitor function isn't required for a particular visitor, the

function can simply be omitted, and simply be bypassed.

An example on how the visitor is used in an accept function can be seen here. This is the class for an if statement in the AST:

```
class statement_ifthen:
    def __init__(self, exp, then_part, lineno):
        self.exp = exp
        self.then_part = then_part
        self.lineno = lineno

    def accept(self, visitor):
        visitor.preVisit(self)
        self.exp.accept(visitor)
        visitor.midVisit(self)
        self.then_part.accept(visitor)
        visitor.postVisit(self)
```

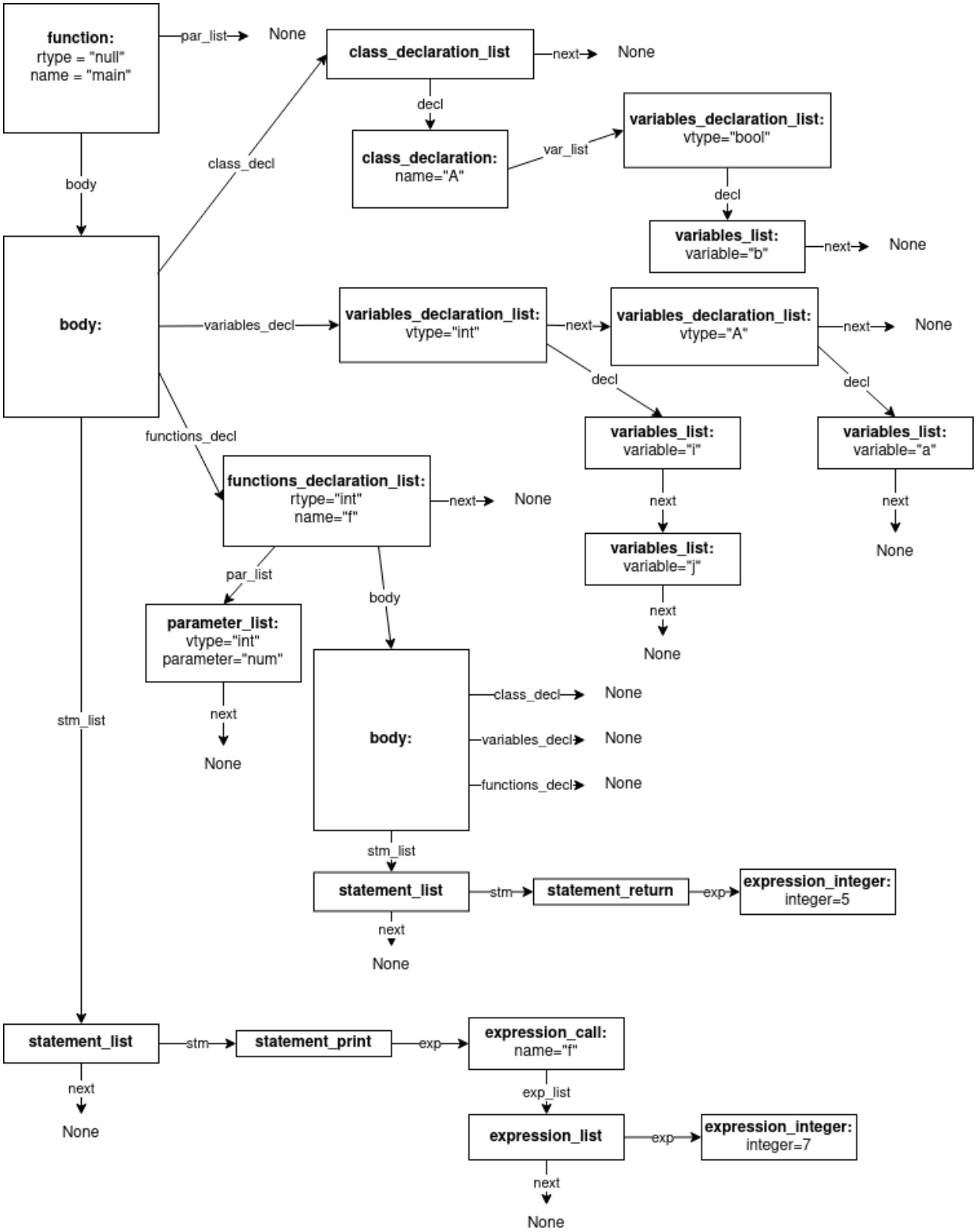
Here we use a pre-, mid- and post visit function for the visitor while calling the accept of both the expression (exp) and the nested statement (then_part), as they are child nodes of the if statements node. This way, the visitor can do some work both before, in between, and after both of the child nodes have been traversed.

We will use this visitor pattern quite a lot for the different phases of the compiler, where we both gather information about the code, check the legality of it, add more data to some nodes and transform the tree.

To give an example of how the AST would be generated, we have the written the following leif code:

```
1 class A{
2   var bool b;
3 }
4 var int i, j;
5 var A a;
6 function int f(int num){
7   return 5;
8 }
9 print(f(7));
```

Here we have a single class declaration, with one attribute. We have two integer variable declarations, and a declaration of our custom class. We also have an integer function declaration with a single parameter and single return statement. The only statement it the main body is a print statement that will run the function "f". From this code, an AST with the following structure would be generated:


function:
rtype = "null"
name = "main"
par_list
None
class_declaration_list
next
None
decl
class_declaration:
name="A"
var_list
variables_declaration_list:
vtype="bool"
decl
variables_list:
variable="b"
next
None
body
class_decl
body:
variables_decl
variables_declaration_list:
vtype="int"
next
variables_declaration_list:
vtype="A"
next
None
decl
decl
variables_list:
variable="i"
variables_list:
variable="a"
functions_decl
functions_declaration_list:
rtype="int"
name="f"
next
None
next
next
variables_list:
variable="j"
None
None
par_list
body
parameter_list:
vtype="int"
parameter="num"
next
None
stm_list
body:
class_decl
None
variables_decl
None
functions_decl
None
stm_list
statement_list
stm
statement_return
exp
expression_integer:
integer=5
next
None
statement_list
stm
statement_print
exp
expression_call:
name="f"
next
None
exp_list
expression_list
exp
expression_integer:
integer=7
next
None

3.7 Printing the tree

For every node in the AST, we have given them a 'toString' function that converts the data held by that node into a string. This works recursively, and aims to print the tree out as legal code, mostly corresponding to the original code. The structure of this generated code might be different that the original. We have aimed to covert the tree to a "pretty" version of the original program, with many line breaks and indentation. We have also added grouping parentheses to binop operators to show how the order was evaluated, and also added some comments to show where if, if-else, and while loops start and end.

3.8 Tests

To check if the AST (Abstract Syntax Tree) is being correctly constructed, one can do so by running the translator with the **-s** flag. This will print the AST in an embedded manner. That is, the output resembles the program that is input, but a few parentheses have been added to each node, so that all content of the tree, which is described by the root node, is encapsulated in a pair of parentheses. This way, there is no doubt about what belongs together.

Test Filename	Description	Expected	Result
nestedIfElse.leif	This test shows that if an nested if-then operation is made in the then-part of the if-then-else operation, the inner operation will inherit the else-part of the outer if.	Allowed	PASS
returnDecl.leif	returns a type declaration	Not allowed	PASS
combTypeDecl.leif	sets the type to be an expression	Not allowed	PASS
unfinMultiCom.leif	missing end of multi line comment	Not allowed	PASS
noStmList.leif	program without an action list	Not allowed	PASS
nestedFunc.leif	This test for correctness of nested functions	Allowed	PASS
wrgDeclOrder.leif	Trying to declare class, function and variables in the wrong order	Not Allowed	PASS
negationInt.leif	trying to print a negated int	Not Allowed	PASS
multiSingleCom.leif	Parsing a file with both a multi and single line comment	Allowed	PASS

4 Symbol Collector

In a compiler, a symbol collector is a critical component that, collects, organises and manages symbols. Identifiers used in a program, such as variable names, function names, class names, and so on. Symbols are collected during previous phase — the lexical analysis and syntax analysis phases, of a compiler. The symbol collector gathers all identifiers from the source code and puts them into a symbol table, which serves as a dictionary or lookup table of the identifiers used in the program. Each symbol entry in the table contains additional information such as the symbol's type, its scope (i.e., where in the code it is valid), and other attribute values depending on the specifics of the programming language. The symbol collector is also responsible for handling scopes. In many programming languages, symbols have a scope, which is the region of the program where a symbol is valid or can be accessed. For example, a variable declared inside a function is typically only accessible within that function; this is its scope. The symbol collector must properly handle such nested and non-nested scopes.

In the context of an Abstract Syntax Tree (AST), a visitor pattern might be used for the symbol collector. In this case, a visitor class traverses the AST, visiting each node, and collects and manages the symbols encountered along the way.

4.1 Symbol Table

We have a class for the symbol table called **SymbolTable**, defined in the same file as the visitor for the symbol collector. The symbol table holds four attributes: **_tab**, **_types**, **parent** and **is_function**. It is initialised with only a single argument for the parent attribute, which must be another symbol table (the parent one). The tab attribute is a dictionary mapping variable-, parameter- and function names to the respective data object for that name. This object will be of a class called **SymVal**. The type attribute is also a dictionary, but this one maps type names to another dictionary that maps attribute names to **SymVal** objects. This helps store the declared types and their attributes separately from the other categories, which is nice because of the structural difference in representation. The 'is' function attributes is just a boolean (initialises as false, but set to true by function visitor functions) that indicates whether the scope is the main scope of a function.

The **SymVal** class is used to represent variables, parameters and functions

in a similar way. It holds four attributes: **cat**, **info**, **rtype** and **metaVar**. The cat attribute is short for category, and can be one of three categories: **VARIABLE**, **PARAMETER** or **FUNCTION**. This helps differentiate between the three categories later. Attributes will have the same tag as a variable. The rtype attribute stores the data type stored in the variable or parameter, or the return type of a function. The info attribute, as well as metaVar, are meta data elements. The info element for both variables and function stores the AST node for either the varaible- or function declaration. For parameters and attributes the 'info' element is an integer showing the parameter, or attribute, offset for that parameter, or attribute, for that function, or class, (how many parameters was declared for the function before this one. Or what index the attribute will have for a class instantiating from the pointer). This is useful in code generation to calculate where on the stack a certain parameter is placed. The metaVar is not initialised here (it is none).

When the symbol collector visitor is first initiated, it creates a new symbol table without a parent. It stores this table in an attribute element as the 'current symbol table'. It also adds the 'int', 'bool' and 'null' types to this tables type list with no attributes for each of them.

Every time the visitor visits a body, it creates a new table with the current one as its parent, and sets the new table to be the current one. This does not happen if the body belong to a function. In that case the visitor creates the new table in the function to allow the parameters for the function to be collected correctly inside the functions scope. Every time a body is exited by the visitor, the current symbol table will be set to the parent of itself. This ensures that nesting is taken care of.

4.1.1 Collecting the symbols

- **Variables:**
  Variables are collected when the vistor is at a **'varibles_list'** node in the AST. The **'preVists_variables_list'** method is called, which creates a **'SymVal'** object with the category **'VARIABLE'** and inserts it into the current symbol table. It stores the AST node for the variable list in the 'info' element.

- **Functions:**
  The pre-visit method on the **function** AST node is responsible for inserting the function into the symbol table. It also readies process of

parameter counting, which is then recorded in the function node by the mid-visit method. It also creates the new table for this function before the parameters are visited and collected.

- **Parameters:**
  The parameters are saved much like the variables. They are collected in the pre-visit of the **parameter_list** node, and creates the SymVal object with the **'PARAMETER'** tag, but instead of storing the parameter list node in info, it stores its offset, or index in the linked parameter list.

- **Classes and attributes:**
  On pre-visit of the **class_declaration** AST node, it only propagates a list to its nested variable declarations. If these variable declarations receive this list, they append themselves to the list rather than being conventionally collected. They also recursively propagate the list to other embedded variable declarations and lists. Then, on post-visit of the class declaration, the visitor creates a dictionary of every attributes name and maps them to a new SymVal object similar to variables, except it inserts an unique index for every attribute into the 'info' element.

Within the same scope, we do not permit duplicate names for variables, parameters, or functions. A class name can coincide with the name of any parameter, function, or variable regardless of their scope, but two classes within the same scope must have distinct names. Lastly, it is not allowed for two attributes within the same class to have identical names. We enforce this by checking if the symbol is already used in the scope or class when we collect the different symbols. If the symbol was already used, we throw an error telling the user that a redeclaration was made, and indicating the line on which it occurred.

For the attributes, since we collect them together with the class declaration. If we detect any attribute sharing a name with another, an error is generated.

4.2 Scope Checking

In our symbol collector, we also check, for each symbol used, whether that symbol is accessible inside the current scope. This includes the symbols for variables, parameters, functions, types / classes and attributes.

For variables, parameters, functions and classes, we link the AST node holding the symbols to the respective symbol in the symbol table. For variables, parameters and functions, we use a lookup method from the current symbol table called **lookup**, that looks through the symbol table (not the type segment), and returns the first SymVal object, corresponding to the relevant symbol that is finds. If it does not exits, the method returns nothing, and we throw an error telling the user that the symbol was not declared. The lookup method checks if the symbol exists in this scope, and returns it if does. Otherwise it calls itself recursively on its parent. If there is no parent, then it returns nothing.

We have a similar method for looking up types called **type_lookup** that does the exact same thing, just on the type table segment. If the type does was not declared, we throw an error. Every time a type name is used, we do this check to see if it exist, and then we make a link between that AST node and the relevant dictionary for that class.

For attributes, we also have a **attribute_lookup** method on the symbol table. Given the potential for various classes to share attribute names and the inability to confirm the validity of the dot operator usage, we can't directly link the AST node to the appropriate SymVal object. We are, however, able to see if an attribute with the given name has been declared. If not, we can safely throw an error. If it has been declared, we make sure to link the current symbol table, so that we can look it up in the type checker.

Another thing, when it comes to expression calls, we assert that the number of arguments used in the call matches the number of parameters in the called function.

After having done all this, we know that every used symbol has been declared correctly.

4.3 Tests

Test Filename	Description	Expected	Result
classArr.leif	This test defines a Point class, creates an array of Points, assigns values, then prints.	Allowed	PASS
Shadowing.leif	The test defines two functions, each named shadowed_function. Prints different outputs due to scoping	allowed	PASS
shadowingVar.leif	This test demonstrates variable scope. A local variable named global_var shadows the global one.	allowed	PASS
varScope.leif	The test defines a global variable, x, and a local variable, x, in test_scope, demonstrating variable scoping.	allowed	PASS
symbolConflict.leif	This test wil raise an error since x is defined as a variable and function, it is allowed for the class to have the name x.	Not allowed	PASS

5 Type Checking

The type checking is implemented using the visitor pattern, extending the base class **VisitorBase**. The visitor pattern is utilised as it enables the addition of operations on the AST without modifying the nodes of the AST, allowing clean separation between the tree structure and operations on it.

In order to properly check types, we need to think about the current scope tracking. To do this it needs to be aware of the current scope of variables and functions. This can be managed by a **current_scope** attribute, which holds a reference to the current scope's symbol table. The symbol table holds the types of all current variables, allowing the checker to understand what types should be associated with each variable name.

For keeping track of the current function and while loop it is within. A current function stack and while nesting stack. the **current_function_stack** is used to handle return statements. When we enter a function, it is then pushed onto the stack, and when we exit, it is then popped. This ensures that when a return statement is encountered, it can be checked against the expected return type of the current function. The **while_nesting_stack** operates similarly, but for while loops this is used to ensure that break statements only appear within a loop.

In the visitor design pattern, the **Pre-vist** and **Post-vists methods** are a part of it and allows actions to be performed before and after visiting a node in the AST. For instance, preVisit_body and postVisit_body methods are called when entering and exiting a new scope, updating the current_scope to reflect this. Pre and post vists methods handle different checks at the stages of the tree traversal, such as type checking of expressions, assignments, and function calls, among others.

Should the type checker detect a type violation during the process, it calls the **error_message** function, to raise an error. which includes a descriptive message of the type violation and at which line it occurred.

The type checker is equipped with various methods to handle the type checking of different constructs in the language. Each of these methods is named according to the construct it's handling. These methods are triggered during the AST traversal when the associated construct is encountered. Each of these methods ensures that the type rules and constraints of its associated construct are strictly followed, raising an error if any violation is found.

5.1 Checking and setting the types

We have talked a lot about both expression types and variable types in the earlier chapters. We have also established the rules of how expressions can be used with respect to their types.

In the AST, we have given every expression AST node class a meta variables called **_type**. This is supposed to hold the type of the expression. We have already been able to determine some of the expression AST nodes type, since some nodes can only have one type. Some nodes, for example the one for variables, we have already determined its type in the symbol collector, since we had to find the variable from the symbol table anyway. For some expression nodes however, we have not been able to determine their type yet, since it may depend on the type of its children nodes. We have also not been able to detect illegal combination or usage of types in expressions. All of this is taken care of here.

For all the relevant and undetermined typed expression nodes, we have created a post visit visitor function to evaluate both the types of the children (for both legality and potential propagation), and then its own type. We make sure to always use a post visit functions to check types to allow all the children to be evaluated first.

For statements that depends on expressions of specific types, like if or while statements, we also check the types of these.

We also make sure that the variable on the left hand side, and the expression on the right hand side of an assignment statement have the same type.

For the binop operators, we have already determined the legal types that can be used on the left- and right hand side of the operator in the AST. We then check if the expressions then has these types here. The equality operators is not as trivial as the others, and can use both booleans and integers, but not at the same time. We therefore check if the expressions are the same type, and then if they are either integers or booleans.

For the unary operators we know exactly the type we can use, and we assert that type from the child node.

For indexes, we must determine whether the expression we index into is an array of some sort, and then if the index is an int. The return type will then be the expressions type with one less array nesting (since we can have arrays of arrays).

When dealing with dot expressions, we initially need to assert if the type of the expression we're accessing is a class that holds an attribute with the given name. To do this, we use the attribute lookup method on the symbol table associated with the current scope for the respective attribute name. This method returns a list of every type / class that has an attribute with that name. We then figure out if the left hand side of the dot expression has any of those types. If it does, we take that type and figure out the type of its attribute.

The print statement which is limited to only take booleans and integers, so we evaluate the type of its expression. The result is then assigned to the print statement for future use during code generation.

For break statements, we assess whether we are currently inside a while loop by examining the while_nesting_stack. If it is empty, we are not in a while loop. Otherwise, we link the break statement to the while loop at the top of the stack for later use.

For an expression call, we know the return type from the symbol collector. But we have to assert the types of its parameters, to make sure that they match the parameters for the called function. This is achieved by compiling the arguments into a list and then, during a post visit to the expression call, traversing this list in conjunction with the function's parameter list, comparing their types one by one.

When dealing with return statements, our first task is to verify its placement is valid. The return statement can only be legally positioned if it's the last statement in the function's statement list with a non-null return type. The AST node for the return statement hold a meta data attribute for its legality. This element is a boolean, and is initialised to false in the AST. We have a pre visit method on the statement list that checks the following things: whether the current scope aligns with a function's scope (remember that the symbol table has an is_function attribute), whether it's the last statement in the list, and whether the function's return type is not null. If all these conditions are true, we know that this statement must be a return statement. We confirm whether it is a return statement, and if it is, we mark its isLegal element to true. Then, during traversal to the return statement, we first check if it is legal, followed by checking whether the type of its expression aligns with the function's return type. In this process, the current function stack proves to be useful.

If at any stage the types does not match the expected type that we need, or if any other check failed, we throw an error with a small expression of what

went wrong and a line number.

5.2 Tests

Test Filename	Description	Expected	Result
WrgWhileSta.leif	This test is using an int in the guard of the while-loop .	Not Allowed	PASS
multiOperator.leif	Trying to have multiple '==' resovling this, means that it will evaulate the first and then the second is then comparing a bool to an int	Not allowed	PASS
wrgArrayAllo.leif	Tries to index an bool into an array	Not allowed	PASS
recursiveClass.leif	This test recursively creates a linked list of integers and prints values.	Allowed	PASS
wrgIfStatement.leif	This test is using an int in the guard of the if statement	Not allowed	PASS

6 Scope flattening

Now that we have gone through both the symbol collector and the type checker, we know that the program only contains legal code, and we have managed to link most AST nodes to relevant meta data. For the sake of

making the code generation easier, we have decided to "flatten out" the nested scopes of the different functions, including the main function. This mainly includes flattening out the symbol table. In this flat version of the symbol table, we must ensure that every variable, function and parameter gets a unique name, to eliminate any name confusion in the new "flat" symbol table. We no longer need to take care of classes or attributes, since we only care about the classes size and the attributes offset in the code generation. And the relevant AST nodes for classes and attributes have already been linked to enough meta data.

The new flat symbol table will be a list of all the functions (including the main function). Every function will then have a list of both its parameters and its variables. It will also have a orders list of all of its parent functions, so that we may calculate where to find parent parameters and variables on the stack later. Every variable will be stored with an offset relative to the other variables for that function. Parameters also have an offset, but this offset is independent of the variables offset. These offset will allow us to calculate the exact relative memory location of the variables and parameters for any function.

Recall the meta data attribute in the SymVal class called metaVar. This element will now be initialised for every symbol (not for types or attributes) as a link to its corresponding flat symbol table element.

7 Register allocation

Within a computer, there exist various types of memory. For our current discussion, we will focus on registers and main memory, which includes the stack and heap. Registers are the quickest memory to access, but they are limited in quantity and have a very small size. When the registers become insufficient to store all the required data, we resort to using the main memory. To be able to carry out any of the calculations and operations needed to execute a program, we must be able to store some intermediate values somewhere to be able to access and use them later. The most optimal, or fastest, way of storing an accessing these values would be if we could store as many as possible inside the registers before having to assort to main memory usage.

One possible way of storing and accessing these values would be to push them to the stack to store then, and pop them again when they are needed.

However, this method isn't optimal as it solely relies on the stack without benefiting from the faster speed of the registers.

An alternate strategy could involve filling the initial few values into the available registers and using the stack for the remaining data. This too isn't an ideal solution as the intermediate values might frequently exceed the limited number of available registers. Consequently, most of the values would end up on the stack regardless.

A more efficient approach involves first determining the total number of intermediate values that need to be stored, and then figure out how they can share the limited registers at our disposal. We can analyse the program to figure out exactly when we need to keep the individual values stored, so that two values that aren't needed simultaneously could share the same register. While this strategy is clearly more complex than merely assigning every value a space on the stack, it is more efficient.

To achieve this we need a few things. We first need to figure out exactly how many intermediate values we have to store. We can accomplish this by allocating intermediate registers for each value. We also need to know exactly where these values are to be accessed again. Following this, we need to analyse this information to figure out which values are live at the same time. This is called a liveliness analysis, and works by creating a graph with the intermediate registers / values as vertices, and drawing edges between values that are needed simultaneously.

The next step involves analysing this information to find an optimal method for intermediate values to share the registers. A colouring algorithm can be applied for this, where the colours would represent the actual registers. If we discover that there are more live values at any point that we have registers at hand, we must put the remaining on the stack. This solution significantly optimises the utilisation of registers for as many intermediate values as possible while only resorting to the stack for a minimal number of values. We have deiced that this last option is the one we are going to use in our compiler.

7.1 Intermidiate register distribution

Our way of distributing registers, including the liveliness analysis and the graph colouring might be a bit unorthodox. We have chosen to do all of this strictly before the code generation. This also means that the liveliness analysis is more of a pseudo liveliness analysis.

We have created a visitor for this called **ASTRegDistributor**. In the same document we have created a class called **intermediateRegister**. This class serves as both the distributed register, as well as the graph after the liveliness analysis.

The visitor itself has a function for the generation of new registers, as well as a function to "utilise" these registers. By "utilising" the registers, we are implying the process of updating a pseudo line number within the register object. The intermediary register holds two line numbers — the first one shows when the register was first utilised, while the second one reveals the time of the register's most recent use. Upon the initial "use" of a register, both line numbers are adjusted to the current pseudo line number. However, in any following uses, only the last used line number gets updated. The pseudo line number is just a number managed by the visitor, starting at 1 and increasing each time the "use" function is executed.

7.2 Liveliness Analysis

The visitor hold a list of all the distributed intermediate registers. Once every register has been distributed and "used", a liveliness analysis can be performed on them. In the class for the intermediate registers, there is a method meant for this. This method is called **addNeighboursConditionally**, it takes a list of intermediate register objects and compares the first and last usage of each register with its own to check if they are live simultaneously. If an overlap is detected, the examined register is added to a neighbouring list, kept by the current register. A for-loop is executed over every distributed intermediate register, invoking this method and passing the whole list of registers as an argument. This turns the list of registers into a graph where every register has a neighbour list of its neighbours. Two registers will now be neighbour if they are live concurrently.

Instead of giving the liveliness analysis its own method, we have combined it together with the colouring method in the visitor.

7.3 Graph Colouring

Graph colouring is a fundamental concept in graph theory. The idea is to assign labels (or "colours") to the vertices of a graph in such a way that no two adjacent vertices share the same colour. In the context of register allocation in compilers, graph colouring is used to assign storage locations

to variables. Each variable is represented by a vertex in the graph, and an edge connects two vertices if they are alive at the same time during program execution (i.e., their values are both needed). A colour represents a register, and the goal is to colour the graph so no two connected vertices share the same colour, meaning no two live variables share the same register.

The minimum number of colours needed to colour a graph G is its chromatic number. Determining the chromatic number of a graph is $\mathcal{NP}$-hard, meaning there's no known algorithm to solve it efficiently for all graphs.

We have created two different colouring algorithms for our compiler. The user can choose between then with the compiler flag -c followed directly by either a 0 or a 1. If this compiler flag is not used, or is used wrong, we default to -c 0.

7.3.1 The greedy algorithm

This algorithm is the default one, and can be explicitly used by setting the -c flag to 0.

We have here written the pseudo code for the greedy algorithm we have chosen. Every vertex $v(c, N)$ has an attribute c (initialised to 0), for its colour, and a list of all its neighbouring vertices. This is a simplified version of what we have implemented:

```
1  Colour(G):
2      for v in G.V:
3          C = NColours(v);
4          while v.c in C:
5              v.c = v.c + 1;
6
7  NColours(v):
8      C = [];
9      for n in v.N:
10         if not n.c in C:
11             C.append(C);
12     return C;
```

The Graph Colouring we have implemented works by in the first loop of the function, edges are created between concurrently used registers. This is accomplished with the **addNeighboursConditionally** method, which adds neighbouring registers based on their usage overlap. After we have established the edges, the function start to assign colours. It does this by looping over all the registers again. For each register, it collects the colours already used by its neighbours. This is done using a set comprehension which extracts the colour attribute of all neighbouring nodes where the colour has already been assigned. It then finds the smallest colour that's not used by any of its neighbours. It uses the **itertools.count** function to generate an infinite sequence of integers, starting from 0, and the next function with a generator expression to find the first colour that's not in the set of used colours. Once we then have found the smallest available colour it is then assigned to the register. If the colour assigned to the register is greater than the current maximum number of colours, the chromatic number is updated.

We choose this greedy algorithm because it has a $O(n^2)$ time complexity, where n represents the number of vertices in the graph. Since for any vertex, when we decide its colour, we first collect the colour from all of its neighbours, and finds the smallest colour not used by any neighbouring vertex. This has a time complexity of $O(n)$. And we do this for every vertex, giving us the total time complexity of $O(n^2)$. Since we know that finding the chromatic number of a graph is $\mathcal{NP}$-hard problem, we can deduce that this greedy algorithm may not give the most optimal result every time, and is essentially just an approximation algorithm. We accept this trade of to get the better time complexity, whereas it is generally assumed (however not proven) that $\mathcal{NP}$-hard problems can not be solved in polynomial time.

7.3.2 The lazy method

We also have another colouring algorithm with gives each one a unique colour. This method can be used by setting the -c compiler flag to 1.

This method aims to give a result in the fastest time possible, and with the laziest of algorithms. It has an $O(n)$ time complexity, due to it only iterating through every node once and giving them each a unique colour. This method does not need the liveliness analysis to create a graph over the intermediate registers, so we also save the computational power of skipping it.

This algorithm is clearly not an optimal solution to the colouring problem (or to our register distribution), and has mainly been created in order to test

our graph colouring algorithm. We will talk more about it in the test section.

7.4 The colours

The "colours" we assign the different intermediate registers translates to actual register names, or into stack locations. For the initial eight colours, we utilise registers from r8 to r15, in ascending order, with r8 representing the smallest, or first, colour that can be assigned to an intermediary register. If a register is allocated a "larger" colour beyond r15, it is assigned a specific location on the stack relative to the initial stack pointer's location. For the stack spilled locations, we want to allocate some space on the stack at the start of the program execution where these values can be stored. We can then save a pointer to this list in another register, allowing us constant access to these registers.

For the colouring of the stack values, we give the intermediate register objects a label stating that it is located on the stack. We also give it an offset so that we can calculate where on the stack, relative to the stack pointer to the intermediate stack values.

In the code generation, we can distinguish between actual registers, and stack locations for out intermediate registers.

For the allocation of the necessary stack space, we have a function in this visitor that returns the calculated chromatic number calculated, minus the number of registers at hand (would be 8).

7.5 Tests

Quicksort, a classic and efficient sorting algorithm, we have tested it by using our two graph colouring algorithms - the Greedy Algorithm and Lazy Colouring Algorithm.

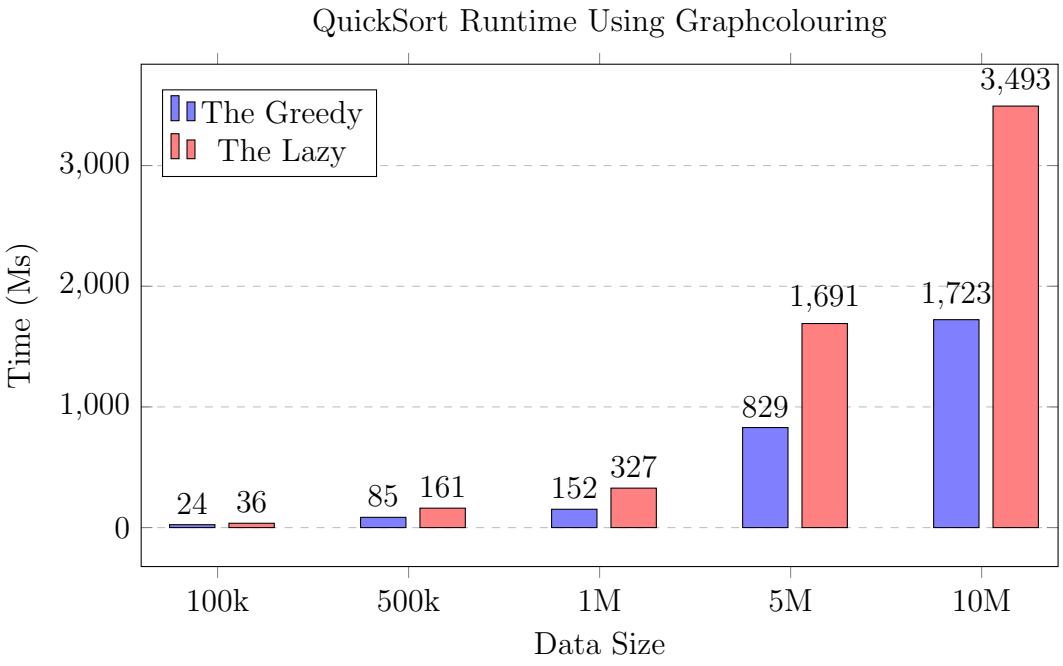
First, we need to generate a list of unsorted numbers, we do this by using this piece of code.

```
i = 0;
N = 100000000;
array = new int[N];
while (i < N) {
    array[i] = (i+5) * 33659939 % N;
    i = i + 1;
  }
quicksort(array, 0, N - 1);
```

where N is the data size input, the array filling part is $O(n)$. Then we apply the quicksort alogirthm on the list, time complexity of this operation is on average $O(n \log n)$. Once the list has been sorted the way we confirm it is by running this code after the sorting.

```
i = 0;
while (i < N - 1) {
    if (array[i] > array[i+1]){
      sorted = false;
    }
    i = i + 1;
}
```

It is iterating through the array and comparing each element with the next one. If any element is found to be larger than the one after it, the sorted variable is set to false. The time complexity of this operation is $O(n)$. Resulting in everything combined would be $O(n) + O(n \log n) + O(n)$.



The graph displaying the results of the test with the size of the data set used for the quicksort on the x-axis and the time taken to execute the sort on the y-axis. The graph shows two bars for each data set size, one for each graph colouring method. The blue bars represent "The Greedy" method, and the red bars represent "The Lazy" method. We can see that for each size of the data set, "The Greedy" method takes less time to execute than "The Lazy" method. For instance, for a data size of 10M, "The Greedy" method takes 1723 ms, while "The Lazy" method takes 3493 ms. This pattern appears to be consistent across all data set sizes.

8 Code Generation

The code generation is responsible for translating the AST, with all the meta data we have created, into something resembling x86-64 assembly instructions a bit more. This includes figuring out, roughly, which operations we need to use, as well as figuring out the memory usage and manipulation we need. The intermediate code will be represented as a list of instruction.

The instructions (**Ins**) will be represented as an **Operation**, followed by a list of argument (**Arg**), and possibly a comment. The operations can be any classical assembly operation like move, add, push and so on, but can

also be a label, or a meta operation. The arguments are represented by a **Target** and a **Mode**. The target is a memory location. It can either be a specific register, an intermediate register, or an intermediate integers or boolean. This is specified with one of the **TargetType** objects. The intermediate registers need another argument for which register (given from the register distribution), and the intermediate integers or booleans need an integer or boolean respectively. The target can also be a label, in which case a label name should be given as well. The mode specifies how the target should be addressed with an **AddressingMode** objects and possibly another argument. It can be addressed directly, indirectly relative to some given offset, or indirectly with another register as offset. Depending on the operation, this should be either one or two arguments. If the operation was a meta operation, then instead of an 'Arg' argument as an argument, it needs a **Meta** argument. This is mainly used to save the intermediate registers before and after a function, allocate some space on the stack or the heap, or to call the print statement.

We add these instructions to the list using a function in the visitor called **_app** (sort for append). It just append the given instruction to the list of instructions.

Here is an example on how to add the instruction for moving an intermediate integer into an intermediate register (Line breaks added for readability):

```
1  self._app(Ins(Operation.MOVE,
2      Arg(
3          Target(TargetType.IMI, t.integer),
4          Mode(AddressingMode.DIR)
5      ),
6      Arg(
7          Target(TargetType.REG, t.retReg),
8          Mode(AddressingMode.DIR)
9      ),
10     c=f"Moves integer into {t.retReg.name}"))
```

All of this will then be converted into actual x86-64 assembly instructions in the emitter.

8.1 Function decoupling

Just before we invoke the code generation visitor, we have another small visitor called the **ASTFunDecouple**. This visitors purpose it to decouple every function AST sub tree from each other and place them in a list. This includes the main function. We do this by creating a visitor, with only two visitor methods on the function AST node. The pre-visit adds the current function AST node to the list, and the mid-visit sets any nested function list to None. Please note that the nested functions withing a function is visited before the mid visit of the parent function is called. This ensures that every functions is added to the list, and that no functions can be nested withing each other.

The reason for doing this is to allow for a simpler code generation where every function appears in full, sequentially, and with no overlap. This seems to be better than having to take care of nested functions withing the intermediate code, or in the emitted code. In the symbol collector and type checker, we have already ensured the legality of these functions, with the flat symbol table preserving the parent-child relationships. Therefore, maintaining the nesting within the AST is no longer necessary.

When we want to call the code generation to generate the intermediate code, we iterate through the list of all the function and call the accept function with the same code generator visitor on each of them.

The code generator just append every command to the end of the code, so every function will just appear right after each other.

8.2 Intermediate code generation

We have used the visitor pattern to create a visitor for this called **ASTCodeGenerationVisitor**. For most AST nodes, we want to translate it into a set of instructions in a specific order. For some nodes, we can add all these instructions in a singular group, while others, we want to split them up, to allow for other instructions to be captured in between them. To create the instruction, we use both the general information from the nodes, together with the meta date we have added in the previous visitors. We also want to add some instructions to the start and end of the main file, to allocate the memory for the stack spilled intermediate registers. A pointer to this will be saved in rbx.

8.2.1 Expressions

For most of our expression nodes, we add all the instructions together in a singular bunch in a post visit method. Using the post visit method allows any nested nodes to be evaluated first. This is useful as we might need some of the values created by the nested nodes. All of these expressions can either use intermediate registers or stack locations, depending on the result from the register distributor. Since we cannot use stack locations in a lot of the instructions, we will be using rax, rcx and rdx (we will refer to these as the operator registers) for most of the actual operations, except move. Next, we will move the values of the intermediary registers into the required registers, perform the operation, and the outcome will be moved to one of the specified registers (commonly rax). We will then move the result into the correct intermediate register for later use. We do not use rbx, since rbx is the register we choose to hold the stack pointer to the stack spilled intermediate registers.

The intermediate integers and intermediate booleans just move which ever value they store to the given intermediate register (or stack location). Since we can use the move command with stack locations, we do not need to use our operator registers, and we can just move the value into the intermediate register. Booleans will be stores as a number, where 1 means true and 0 means false.

When a variable or parameter is used in an expression, we will calculate its location on the stack, and move its value into the given intermediate register. The calculation of its location will be explained under the variables section.

The unary negative operator for integers will uses the operation registers, rax and rcx. It moves the value of the integer expression from its intermediate result register into rcx, and 0 into rax. We do a subtraction instruction to subtract the value from 0. The result is put in rax and then moved to the intermediate result register for this node.

All of the binop operators for the arithmetic and logical operations work similarly with the operation registers, moving the relevant values into rax and rcx. Then the respective operation is inserted with its operation code, and rax and rcx as arguments. Then we move the result from rax into the intermediate result register.

The comparison binop operators also uses the operation registers for the arguments. IThey create two labels: one is used to leap to if the comparison returns true, and the other marks the end of this instruction block. We then

compare the two registers using the compare instruction, followed by a conditional jump based on the operator utilised, to the true label. Subsequently, we input the instruction for the false section, where we move 0 (false) into rax, and then include a jump instruction to the end label. Following this, we have the true label initiating the true segment. Here we move 1 (true) into rax, before reaching the end label. After this point, we move rax into the intermediate result register.

The unary not operator works very similar to the comparison, where we just compare the value with the intermediate boolean, true.

When creating a new instantiating of a class, we move the stored size (attribute count) for the class into rcx. We then call a meta instruction called **ALLOCATE_HEAP_SPACE**, which allocated a number of quadwords (every variable and value uses quadwords) equal to the value in rcx, on the heap. rax will then hold a pointer to the memory. rax is move into the intermediate result register.

Creating a new array is very similar. However, we move the value of the index into rcx instead.

When using the dot operator on a class object, we put the pointer to the object into rcx. We use the offset, stored as meta data in the attribute, to calculate the location from the heap pointer, which will then be indirectly relatively accessed and moved into rax. We then move rax into the intermediate return register.

For the index expression, we move the pointer into rax, and the index into rcx. We then move rax, accesed indirectly relative with the register, rcx, as offset, into rax. The result is now in rax, and is move to the intermediate result register.

The expression call is the only expression to be spit up into multiple visitor methods. We have a pre visit that saves the current intermediate registers. followed this, we have a post visit on the expression list (the argument list) that pushes their values to the stack. The values are pushed in reverse order due to the post visit. This is also the desired push order. After, we have a post visit on the expression call, that first calculates the base pointer for the called functions parent function. We explain this calculation later. Where it then calls the function, and expects the return to be in rax afterwards. Here we pop all the arguments and the parent base pointer before restoring the intermediate registers and then moving rax into the intermediate return register. The saving and restoration of the intermediate registers are meta instructions and are explained further in the emitter.

8.2.2 Statements

Some of the control structures are split up into multiple visits. This is to allow other statements and expressions to be evaluated in between their instruction.

The if statement uses two visitor methods. One in between the guard expression and the nested statement (mid visit), and one at the end (post visit). The one at the end only adds a label marking the end of the if statement. The mid visits takes the value from the guard expression and compares it to the value of true. It then has a jump if not equal to the end label. If it was equal to true, then it does not jump, and the nested statement will be executed before the if statement is naturally exited.

The if-else statement works similarly to the if statement, except it uses three visitors and two labels. It has both an end label, for both the 'then' and the 'else' part is done, and also one for the start of the 'else' part. Right before the else label, we add a jump to the end, so that if the 'then' part was executed, the 'else' part will be skipped. The first part (between the guard and the 'then' part) is much the same as the normal if statement, except the conditional jump, jumps to the 'else' part instead of the end.

The while statement is also a lot like the 'if' statement. The main difference is that before the guard, (in the pre visits) we add a start label for the while loop. And right before the end label, we add a jump to the start of the 'while-loop'. This way, the guard is evaluated again and then the conditional jump, and if the guard is still true, the nested statement will be executed repeatedly until the guard is not true any more.

The break statement is simple, and just adds a jump to the end of the 'while-loop'. Recall that we gave the break AST node a link to its respective' while-loop' in the type checker.

The return statement moves the value of the nested expression to rax. This works for the function return since the return statement is only allowed to be the last statement in a function, and we expect the return to be stores in rax after a function call.

The print statement moves the nested expressions value into rcx. Then it inserts a meta instruction called **CALL_PRINTF** with the value of the meta data element called **printType** from the AST node as an extra argument. The call print meta instruction will print which ever value lies in rcx, and will format the print as either an integer or a boolean dependent on the print

type given. This is handled in the emitter.

The assignment statement can generate three different sets of instructions dependent on weather we are assigning to a normal variable or parameter, an attribute or to an index in an array. If we assign to a normal variable or parameter, we will first calculate its location on the stack, and instead of getting the value, we will be moving the right hand side expressions result into the calculated location on the stack. If we are assigning into a dot variable, we will use the pointer given to the left hand side, and by calculating the index for the attribute with its offset, we can move the right hand side value into the indirectly accessed pointer with an offset of 8 times (one quadword) the given offset for the attribute. Here we are using both rax for holding the pointer, and rcx for holding the value of the right hand side. For index assignment, we need three registers to hold everything. We use rax to hold the pointer, rcx to hold the index and rdx to hold the value to assign with. We then move rdx into rax indirectly with an offset of 8 times (on quadword) rcx.

8.2.3 Variables and functions

Variables and parameters are stored on the stack around some base pointer for the functions whose scope they belong to. The current base pointer is stored in rbp. Every function has a base pointer. When a function is called, the arguments will be pushed to the stack in reversed order, Next, we place the base pointer of the function's parent function onto the stack. See the static link climb section to see where we find this pointer. When the function is then called, we push the current base pointer to the stack and move the current stack pointer into rbp as a new base pointer. Then we can allocate space for the functions variables on the stack as well. Now, from the base pointer, we can find any parameter by accessing the base pointer indirectly with an offset stated in the parameters meta information plus three (for the previous base pointer, the parent base pointer and one extra since the offsets start with 0). We of course multiply this offset by 8 (one quadword). Remember that the parameters were pushed in reverse order, so the first parameter is closets to the base pointer. To access a variable, we find it by accessing the base pointer indirectly with its offset (times 8) as an offset in the opposite direction as the parameters. At the end, we restore the stack pointer and pop the parent base pointer from the stack before giving control back to the caller. Remember that the return (if any) will be stored in rax by the return statement at the end.

8.2.4 Static link climb

To find parameters, variables or even base pointers to a parent function, we use what is called a static link climb. Since every function instantiating (except for the main function) has its parents base pointer stored on the stack right before the base pointer, we can keep following these pointers until we arrive at the parents instantiation. From here, we can access anything from that functions scope.

By knowing that a function can only call another function is that functions is declared in either its own, or one of its parents, scopes, and by knowing the level difference between the current function and the desired parent function, whose scope we would like to access, we can follow the static links once for every level difference. We calculate the level difference using the meta data in the flat symbol table, where every function has a list of all of its parents in order. If the scope we want to access it this scope, we just use its own base pointer.

When following a static link, we use rdx to not mess with rbp. We move the base pointer into rdx. Then for every level difference, we follow the parent pointer two quadwords in from of the element that rdx now points to. When we are done, rdx will have the base pointer to the desired scope.

9 Emit

The emitter does not follow the visitor pattern.

The emitter, which is also a class, is initialised with the intermediate code generated from the code generator, and the number of extra intermediate registers that we need to allocate to the stack.

It holds a list of x86-64 Assembly instruction as strings. At initialisation, this list will be empty. It also holds a lot of meta data for how the code should be formatted when inserted, called **instruction_indent** (size of indent of instructions), **instruction_width** (for how long an instruction should be before its comment appears) and **max_width** (for how long a line may be including the comment). It has a variable called **lbl_counter**, initialised to 0, but stores how many labels we have created.

It has a function called **getLbl**, that generates a new label, using the lbl_counter, increments the counter, and returns the label. This is used whenever we need

to create a label in the emitter that was not created before hand.

We have three methods for adding a new instruction to the list. The **_raw** method takes a string, and will just append this string into the code list. This is mostly used for line breaks by calling _raw with an empty string. The **_lbl** method takes a string, meant to be a label, and is added to the list with a semi colon appended to the string. This is meant to add labels. Then we have **_ins** that takes two strings, an instruction and a comment. The ins will add spaces equal to instruction_indent, then the instruction, then spaces until the instructions has the length of instruction_width. Lastly, the comment is appended. If the comment reaches beyond what is allowed, the comment will be split and appear on the next line, indented to meat up with the original comment indent.

We have a method for getting the code called **get_code**. It joins all the lines together with newlines in between, and return the code.

Then, the most important function, **emit**, is the function that will translate the intermediate code into the actual instructions. It first calls the program prologue, then, for every instruction in the intermediate code, it calls another function called **_dispatch** with the intermediate instruction. Lastly, it adds the program epilogue.

The function prologue adds some formats for print that we use for the print statement, and for a single error check on heap memory.

The epilogue adds a function to call on the memory allocation error, that prints the formatted string for bad memory allocation, and halts the program.

9.1 Translate instructions

The **_dispatch** functions takes an intermediate instruction, formats it into the x86 assembly version (this may evaluate to multiple instructions) and appends it to the instruction list.

The dispatch function itself looks at the operation, and dependent on that, it invokes one of multiple methods. If the intermediate instruction is a simple one that can be converted directly into a single instruction, we call a function, **_simple_instruction** with the intermediate instruction as input. For any non simple instruction, we call a more specific function. The non simple instruction include integer division and modulus. However there can also be labels and meta instructions. If the operator is meta, then we look at the instructions first argument. For every different meta argument, we call a

different function.

The simple instructions are handles in a function called **_simple_instruction**. This function takes the intermediate instruction and first looks at the operation. We have a map from operations to the string representation in the emit file, so we can easily convert the operation to a string. We then add the argument (Arg objects) to the line (if any are given). We use a function called **_do_arg** to convert the arguments to strings. The first argument appended to the line with a singular space between the operation and itself. Any remaining arguments are appended with a comma and a space between them and the previous argument. Lastly we call the _ins function with the newly generated instruction line, and the comment from the intermediate instruction.

If the operation was a label, we fun a function called **_label**. This function is very simple and just calls _lbl with the argument formatted with _do_arg.

For the division and the modulus, we call the functions **_div** and **_mod** respectively. For both of them, we fist move the second argument into rax, insert the "cqo" (to double the memory in rax to a double quadword) and do the division operation with the first argument. Then, for division, we move rax into the second argument, and rdx for modulo.

9.2 Format arguments

We have a method called **_do_arg**. This method takes an argument (Arg), and converts it into the string representation of that argument, to be used in the instruction. It first looks at the target. If the target is any of the specific registers, we will set the string as the name of that register. If the target is an intermediate register, we will take the next argument from the Arg object, and use the name of that intermediate register as the string. For any intermediate integer, we will set the string to that integer (with a $ in front). And for any intermediate booleans, we use $0 for false, and $1 for true. If the target is a a label, we just use the label as the string.

Lastly we look at the mode. If the addressing mode is direct, then we do nothing to the string. However, if the addressing mode is indirect relative to a given offset, we encapsulate the string in parentheses (to make it indirect) and add the offset (also given as argument to the mode in this case) multiplied by -8 (one quadword, indexing backwards means going forwards on the stack towards newer elements) in front. This makes the call indirect with the given offset. If the addressing mode is indirect relative to a register, we format the

string as such: $(\text{reg}, \%rcx, 8)$. We know that this is only used for arrays, and that we always move the index into rcx before hand. Note that the offset is not negative. This is because memory allocated on a heap has a pointer to the smallest index of that blob, so we need to only index in the positive direction. This is also true for class objects. The fix there was to give a negative offset for the indirect relative to an offset.

When we have taken care of both the target and the mode, we return the string to the caller.

9.3 Meta instructions

There are a few different meta instructions. For each, we call a different function. Every meta instruction name is the same as the function name, except the function name is in all lower caps.

The **main_callee_save** and **main_callee_restore** functions are only ever invoked at the start and end of the main function. The save allocates stack space for the extra stack spilled intermediate registers we need, and saves a pointer to it in rbx. Then it saves the registers from r8 to r15. The restore restores r15 to r8 (reverse from the save) and deallocates the allocated stack space for the stack spilled intermediate registers. We dont really need to save any registers here, but is mainly done to make it easier to potentially link the program to another assembly file when assembling. However this is not intended.

The **caller_save** and **caller_restore** saves all registers from r8 to r15 and the stack spilled intermediate registers, and restores them again after a function call.

The **callee_prologue** saves the current base pointer to the stack and makes the current stack pointer the new base pointer. And the **callee_epilogue** restores the stack pointer with the base pointer, then restores the base pointer with the one save on the stack, and lastly calls "ret" to return from the function.

call_printf takes the intermediate instruction as an argument. We have three different formats to print dependent of the print type. The print type should be included as an element of the intermediate instruction. If the print type is an integer, we move the format for printing an integer (defined in the program prologue), into rdi and the printable object into rsi. We then execute the printf method to print the integer. If the print type was

a boolean, then instead of moving the integer format into rdi, we insert a check on the printable object to determine if it is true or false. Dependent on the outcome, we either insert the true print format into rdi, or the false format (also defined in the program prologue).

The function **allocate_stack_space** allocated space on the stack for local variables. It takes a singular argument, found as the second argument (not Arg) in the instruction. This argument is an integer representing the number of local variables that should be allocated. We only insert one line where we move the stack pointer to allow for the variables. Note that there is no deallocator for stack space. We don't need one, since we always restore the stack pointer to the base pointer after exiting a function.

allocate_heap_space first saves rdx, rsi, rdi, r8, r9, r10 and r15 since we are going to need these registers for a syscall. We move rcx into rsi. rsi is the register that holds the size of the memory to allocate, and rcx will hold the size before calling this meta instruction. We also move rcx into r15 for later use. We bit shift rsi to the right by 3, thus multiplying it by 8 (one quadword). This is the size of the memory we want to allocate. We set rdx to 0, move 9 into rax (syscall for mmap) and a few arguments into r8, r9, rdx and r10, before making the syscall. Now, the memory should be allocated, and rax should hold a pointer to it. If rax is -22, we assume that something has gone wrong, and that no memory was allocated. Therefore, we compare rax to -22 and jump to our error message for bad memory allocation. Lastly, we use r15 access every index in the newly allocated memory and move 0 into every space. Lastly we restore all the registers that we saved at the start. rax should now hold a pointer to the memory.

9.4 Program prologue

The program prologue is include in the top of the generated assembly code. The function **program_prologue(self)** utilises three custom methods: **_raw()**, **_lbl()**, and **_ins()**, which are used to output Assembly directives, labels, and instructions respectively.

In this function, we generate several assembly directives and data items.

mmapEr, **form**, **formTrue** and **formFalse** are labels that mark specific points in the program. **mmapEr** is the label for an error message, this string is output when a memory map operation fails. **form** is a label for a string format, that is used to printing an integer in C **formTrue** and **formFalse** are used to be able to print the strings 'true' and 'false'. The rest is used

for marking the beginning of the code section and the entry point of the program.

9.5 Program epilogue

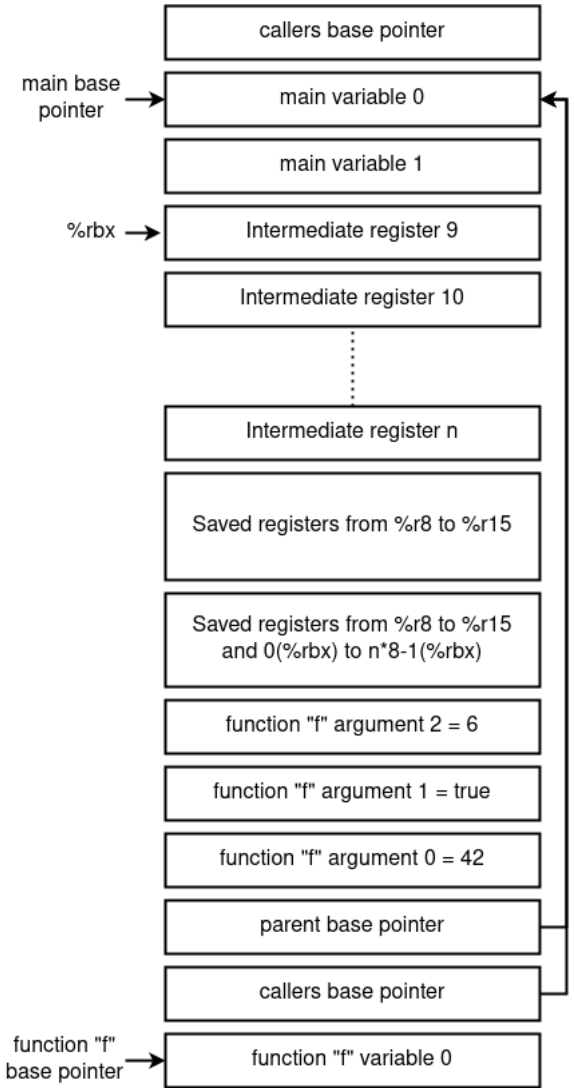
The program epilogue is designed to generate an epilogue for an assembly routine. The first is the **mmap_error** this is a label for an error handler. If something goes wrong with a memory map operation elsewhere in the code, execution might jump to this point. **leaq mmapEr(%rip), %rdi** loads the address of the **mmapEr** string (relative to the Instruction Pointer Register - %rip) into the destination index register (%rdi). This is preparing to print the error message. we use the **xor %rax, %rax** to set value of the register to zero. It is used to indicate the number of floating point parameters passed to a function in the x86-64 calling convention, and printf doesn't expect any in this case. Next we move 60 into **%rax** which is the system call number for exit in x86-64 Linux. This is preparing to exit the program. followed by setting **%rdi** to zero which is done to pass the first argument to a system call. Here, it's setting the exit status to 0 and lastly we call **syscall** to trigger the system call which in this case exits the program. This function essentially creates the epilogue of an assembly program, handling a potential error and ensuring a proper exit from the program.

9.6 Stack example

We have here created an example of what the stack would look like after a simple function call. Before doing this, we have created a simple leif script that has two variables in the main body, and a singular function that takes three parameters and has a variable itself:

```
1  var int i, j;
2  function int f(int i, bool b, int j) {
3      var int result;
4
5      # Stack shown from this point
6
7      if (b) {
8          result = i;
9      } else {
10         result = j;
11     }
12 }
13 i = 42;
14 j = 6;
15 print( f(i,true,j) );
```

In this script, we call the function inside a print statement. We want to show how the stack is represented when we reach the point of the comment.



Note that the stack grows downwards and that the oldest elements are on top. We fist save the current base pointer on the stack when invoking the script, so that we can restore it later. Then we allocate space for the main variables, and then for the intermediate stack spilled registers (There would not be any for the above script, but we have still shown them in this example), and lastly we save the registers from r8 to r15 so that we may restore them again before terminating. Then, in the statement lists for the main body, we call f with three arguments. Here we first save all of the intermediate registers, including the stack spilled ones, if any. These arguments are pushed in reverse order, followed by the parents (in this case it is main) base pointer. Inside the function, we push the callers (the current) base pointer before allocating the space for the local variables for the function.

The given figure represents how the stack would look after all of these values has been pushed.

When the function is exited, we will deallocate the local variables, restore the base pointer, and pop the parent base pointer and all of the arguments from the stack. Then we restore all the intermediate registers, including the stack spilled ones. Now, we are back in the main scope after the function has been called. Right before the program is terminated, we will restore the registers, r8 to r15, deallocate the space for the stack spilled registers, pop the main variables and restore the callers base pointer.

10 Use

Our compiler was designed with the intent that it should work in a Linux environment. You would need to be able to run python 3 to use our compiler. When compiling a .leif file with our compiler, the resulting compiled file would be an x86-64 assembly (.asm) file that the use can assembly any way they see fit.

We have a few compiler flags that can be used with our compiler.

The fist is the "-i" flag. This flag specifies the input file. This is necessary to specify the input file that the user would like to compile.

The "-o" flag is the output flag, and is used to specify where the compiled code should go. The "-o" flag must be followed by a file name. If the flag is not given, the output will be written to the terminal.

The "-s" flag forces the compiler to print the original source code instead on the complied code. It will parse the code and generate the AST, but will then print the AST instead of compiling it any further. This will not run any of the visitors, so the legality of the source code will not be checked any further that what the lexer parse is able to. The printed code might not be completely equal to the original, however the functionality of the code would be the same.

The "-v" flag would make the compiler print out its version number, which would be "unfinished alpha".

The "-h" flag will make it print a helping tutorial out, showing what flags are available, and how to use them.

Lastly, the "-c" flag allows the user to specify the desired register allocation

method. The flag must be followed by either a 0 or a 1. If 0 is chosen, the compiler will use the liveliness analysis and the greedy colouring algorithm to allocate and distribute the registers. If 1 is chosen, the lazy method will be used, which gives every intermediate register a unique memory location. Not using this flag would give the same result as writing "-c 0".

To compile a simple file all the way down to an executable, we recommend calling:

```
python compiler.py -c 0 -i test.leif -o test.asm
as test.asm -o test.o
gcc test.o -o test
```

This will compile the leif file down to an executable.

11 Future

11.1 Peephole Optimising

Peephole optimisation is a technique used in compiler design for improving the efficiency and performance of generated code. This method involves looking at a small window or "peephole" of instructions in a program to detect and eliminate redundant or unnecessary instructions, typically the size of a few instructions or a basic block.

It goes through each line of the code and tries to determine whether there is a pattern that can be optimised. If such a pattern is found, the optimisation is carried out based on an implemented pattern.

The algorithm keeps searching for pattern in the code until no more changes can be made, once it has completed a pass over the code, it check whether something has changed, if it has it runs the algorithm again until no changes has been made. Pseudo code for running such an algorithm could look something like this.

```
repeat
    for each instruction in succesion do
        for each peephole pattern in succesion do
            repeat
                apply the peephole pattern to the code
            until the code did not change
        end
    end
until the code did not change
```

11.1.1 Patterns

Patterns refer to recognisable sequences of instructions that can be replaced with equivalent but more efficient sequences. The ability to recognise these patterns is what makes a peephole optimiser effective.

- **Redundant move** This would look for patterns where a value is loaded from a memory location, moves it into a register, and then directly into another.

  movq a b
  addq b c

  would be replaced by this

  movq a c

  Another redundant move would be from a register into the same register

  movq a a

- **Constant folding** If the value of a constant expression can be calculated at compile time, we can simply replace it by its value. e.g.

  movq 3 a
  addq 5 a

could be replace by

```
movq 8 a
```

- **Strength reduction** Here we look for costly operations that we could replace by cheaper ones. Multiplying by a power of two can be replaced with a left shift operation.

  ```
  imulq 4 a
  ```

  is then replaced by

  ```
  shlq 2 a
  ```

- **Add Zero Instructions** Sometimes in the code a add zero instruction occurs, here the pattern could simply remove this redundant line of code.

11.2 Unintentional behaviour

During the testing of the language, we stumbled upon a few undesired behaviours. In the future we would like to fix these problems, but we did not find time to do so before the deadline.

11.2.1 Class shadowing

Classes does not shadow properly. They shadow in regards to instantiating new class objects, but the attributes are not shadowed. If a class with some attributes is defined in the main scope, and we, in another scope, redefine that same class with different attributes, we can access both classes attributes. Here, the index for the attributes (creates as meta data internally) are the only thing that defines the location for the attributes. This means that attributes can overlap on the index, and that the first attribute in each class definition is the same element. These attributes can have different types, making it possible to read pointers as integers or even to read class pointers as array pointers and vice versa. If two attributes have the same name, the outer most classes attribute will dominate, and the code will use its index and type. Even more bad behaviour comes from the initialisation of new class object. The inner most class definition will shadow the outer. If the two

classes does not have the same size, it can be possible to go out of bounds, although this behaviour does not seem to be problematic when we tested the phenomenon.

This does not seem to be a big problem, although it can lead to heart bleed in the code.

We speculate that the solution would be to find some way of renaming the types in the symbol table, so that every type has a unique name. We would also have to find a system for finding the correct unique type when the type name is used in the rest of the AST.

11.2.2 The null variable

Due to the way we have defined null, originally meant to be a special return type for a function, variables can now be of type null. They can be declared, assigned with other null variables or with null functions. They can be passed to functions as arguments, if that functions specifically takes a null type parameter. Ultimately, the null variable cannot be used for anything, but is just an inherent quirk of the language. Patching this out could be as easy as to make check in the type checker that states that variables and parameters cannot be null type.

11.3 Other possible features

Some other features we would have liked to have added, would be to add a size method to arrays. We could do this by letting the symbol ".size", when called on arrays, legal. We would have to allocate one more quadword for arrays when using the mmap syscall, and initialise the first index to hold the size on the array. We would add one to every index, when trying to index into the array and let ".size" function as the index 0. The size would always be interpreted as an integer.

We would also have liked to implement a string type class, similar to the way arrays work. We could do this by implementing the simple "char" type, so that we may use arrays of type char instead of a string. This would allow us to reuse the functionality of an array. However, a quadword is much larger than needed to store a char, so the printing method for a char array would have to look quite special in the emitter. There are of course other ways of implementing strings, but since chars don't fit well inside quadwords, and

since strings can be of any length, this might prove to be very different from everything else we have implemented.

We would also find it interesting to try and implement classes as real classes with methods and such, instead of just structures as we do now. We would have to think a lot more about the design of these classes, since we can't just store then as arrays anymore, depending on the scope of the task. We might be able to implement simple classes that can hold both attributes and functions, but not nested classes. We could implement the functions mostly like we already do, however, we would have know which class object the function was called on, so we know where to find its attributes. We do believe that this could be handled by modifying the AST at some point using another visitor, to look like the ones we already use. This solution might not be a good solution, so if we had a lot of time to work it out, we would find a better way.

There are also other things that would be interesting to implement, like garbage collection, operator overloading and co-routines, however, we believe that implementing these would force us to rework a lot, if not most, of our current implementation, and add much more to it. We therefore deem that these are the least likely things that we would like to implement in the future, given that we had a lot more time on our hands.

12 Evaluation

In our project description, we stated that our overall goal was to define a "minimum language" and a compiler to implement it. This minimum language should be able to use variables and be able to use the types of integers, booleans, arrays and structures (classes). We wanted to be able to use the normal operations on these, like arithmetic operation on integers, logical operations for booleans and comparisons for both. For arrays we wanted to be able to index into them, and we should be able to access attributes in structures. We should be able to define our own structures, and for both arrays and structures, we needed a way to allocate the needed memory.

We also needed some control structures like "if" and "while" statements, as well as be able to define and call functions with different parameters, and be able to make both nested functions and nested scopes.

We have implemented, and tested, all of this functionality, although we have

found a few unintentional behaviours in some of our functionalities (See the Unintentional behaviour under Future). We can see, through our tests, that the language works as intended (mostly).

In our project description, we also stated some extra features that we potentially would like to implement, if we had time to do so. We mentioned implementing string types, expanding the structures to real classes (able to hold methods inside and such), operator overloading, coroutines and garbage collection. We have not implemented any of these features, since we did not have the time, and came to the conclusion that the implementation of these would be very large in comparison to the other features we have.

Other than that, we also mentioned a few code optimisations. Peephole optimisation, register allocation and liveliness analysis, and loop-optimisation. We did create a visitor to take care of the liveliness analysis and register allocation, and have proven via tests that this optimises the code a noticeable amount. We have also look a bit into peephole optimisation, although we did not find the time to actually implement.

In our compiler, we are relying a lot on the AST and the visitor pattern to traverse said AST. In order to hand meta data and other information to other visitors, or other AST nodes, often add new attributes to the AST nodes, even though we did not define all of them in the AST node classes themselves. For simplicity sake, we could go through the different visitors and see what attributes we add to the different nodes, and perhaps add some of the more important ones to the AST. This would allow us and any potential reader of the code to grasp the data more easily and clearly.

On another note, we could have made the register allocation in another way, where we might split the code generator up into two or more visitors, and add the register allocation methods in between the code generator. This would allow us to run the liveliness analysis and register allocation in a more realistic way, and could also help us if we would have implemented the peephole optimiser.

13 Conclusion

In our project description, we stated that out goal was to create a compiler for a so called "minimum language", that we also defined our self. We did manage to fully implement our minimum language, although with a few bugs mentioned under the unintentional behaviour segment. This does not affect

the functionality of the language too much, and we are still able to compile and execute almost any program written in leif.

We also listed some extra functionalities that we could have added, have we had the time to do so. We managed to add one of these, namely the register allocation and liveliness analysis. Classically, this would be a part of the intermediate code generation, but we have put it right before that. We have also proved that the liveliness analysis and graph colouring algorithm speeds up the compiled code by quite a bit.

All in all, we are content with our progress, and the work that we have done. We have reached our goal of implementing a minimum language, and have even added an extra feature.

14 Bibliography

References

[1] David M. Beazley - PLY (Python Lex-Yacc), https://www.dabeaz.com/ply/ply.html. Last accessed 31 May 2023

[2] Kim Skal Larsen - A Simple Compiler in a Learning Environment (SCIL), https://imada.sdu.dk/u/kslarsen/dm565/scil.php. Last accessed 31 May 2023