

## Final Project for the *Practical Course: Systems Programming in C++* Summer Term 2020

Michael Freitag, Moritz Sichert ({freitagm,sichert}@in.tum.de)  
<https://db.in.tum.de/teaching/ss20/c++praktikum>

Create a fork of the Git repository at <https://gitlab.db.in.tum.de/cpplab20/final>. The repository contains a project scaffold on which you can base your solution. **The final project is due on 16.08.2020 at 23:59. Do not forget to git push your solution before the deadline expires.**

Once the deadline for this sheet expires, a signed tag will be created in your fork automatically. Your solutions will be graded based on the state of your repository at this tag. **Do not attempt to modify or remove this tag, as we cannot grade your solution otherwise.**

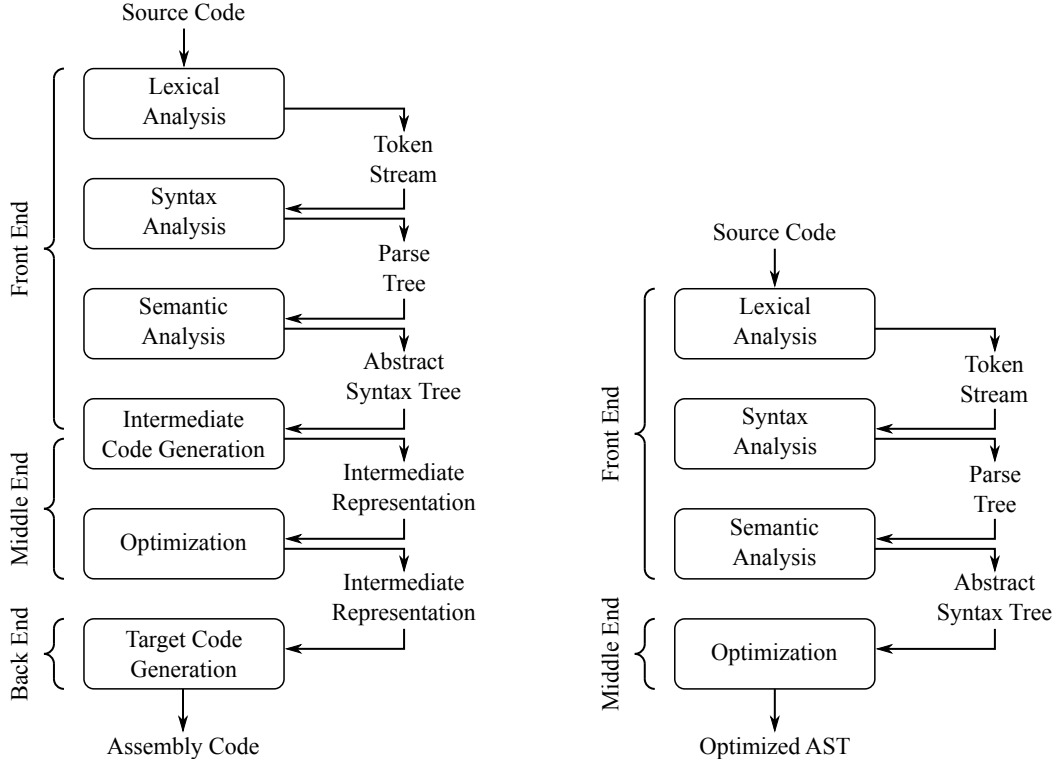
### Outline

For your final project, you will implement a (simplified) just-in-time compiler for a simple programming language. In particular, you will implement a C++ library which provides the following high-level functionality:

- A user of the library can register a number of functions by passing their source code to the library.
- The library exposes an interface that allows its users to call these registered functions.
- Once a function is called for the first time, the library compiles the source code string into an optimized abstract syntax tree that can be evaluated to obtain the result of a function call.

Your implementation approach should roughly follow the roadmap prescribed by the milestones listed below. Each milestone describes in detail which goals you need to achieve in order to complete the milestone, and contains a detailed description of the algorithms that are required to achieve said goal. Apart from the requirements listed in each milestone, your implementation should fulfill the following general criteria, which are presented in no particular order of importance:

- The program scaffold does not contain any predefined classes. You should come up with a suitable architecture for your implementation that promotes clean and extensible code.
- The program scaffold contains only minimal `CMakeLists.txt` files. You will need to adjust the `CMakeLists.txt` files according to your requirements.
- As your implementation will contain quite a few classes, you need to organize your source code in a suitable manner (e.g. put logically separate parts of your code in separate directories and namespaces).
- Your code should contain some minimal documentation.
- You should write some test cases that verify the correctness of your implementation. You should at least verify that the major algorithms you implement are working as expected. The program scaffold already takes care of locating the `GTest` library, and our GitLab will attempt to execute your test cases when you push your code.
- Your code should adhere to the best practices presented in the lecture. Furthermore, you should take care to use suitable data structures and avoid unnecessary overhead.



(a) Main components of an industry-grade compiler such as g++. (b) Main components of the simplified compiler implemented in this project.

Figure 1: Comparison of an industry-grade compiler (a) and the simplified compiler implemented in this project (b).

The remainder of this section provides a high-level overview of the process of compiling a function, which will comprise the majority of work in this project. The compilation process is typically organized into a number of very clearly defined stages in an industry-grade compiler (cf. Figure 1a), which gradually transform a program from source code to optimized assembly code. For the purposes of this project, we will omit some of these stages (cf. Figure 1b), and output an optimized abstract syntax tree instead of optimized assembly code.

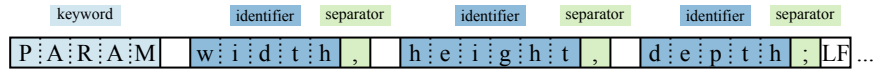
## Lexical Analysis

The lexical analysis stage stands at the beginning of the compilation process. It interprets the source code as a stream of characters, and attempts to combine the individual characters into valid tokens of the programming language which have a defined meaning. Whitespace characters are typically ignored during lexical analysis and all subsequent stages. During this stage, only some very simple errors such as illegal characters in the input can be detected. The output of the lexical analysis stage is a stream of tokens which is passed on to the next stage.

Consider, for example, the following character stream which could be the start of a legal program in the programming language you are going to implement.

P	A	R	A	M		w	i	d	t	h	,		h	e	i	g	h	t	,		d	e	p	t	h	;	L	F	...
---	---	---	---	---	--	---	---	---	---	---	---	--	---	---	---	---	---	---	---	--	---	---	---	---	---	---	---	---	-----

The lexical analysis tokenizes the character stream and assigns a defined meaning to each token such as “keyword”, “identifier”, or “separator”.

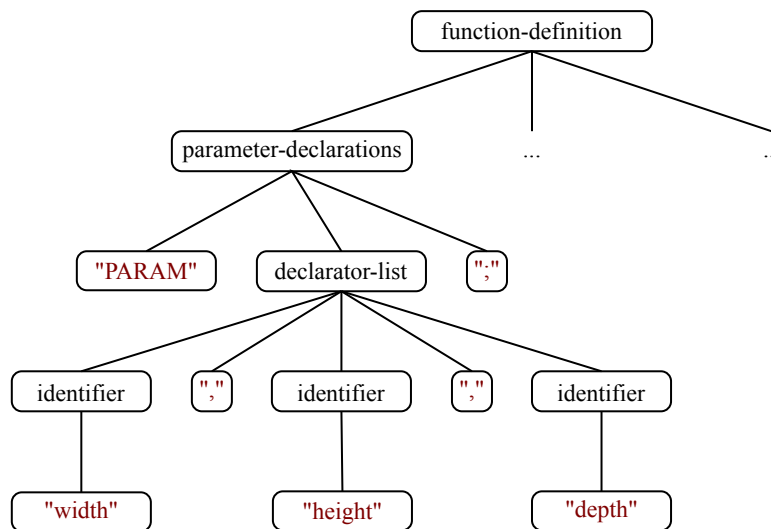


Note that the whitespace characters in the input character stream (shown in white) are not part of any token.

## Syntax Analysis

The syntax analysis receives the token stream from the lexical analysis stage and attempts to parse the syntactic structure of the source code based on a set of formal rules. These rules are typically specified in the form of a formal grammar (see below). Many syntactic errors, such as forgotten semicolons, closing parentheses etc. can be detected in this stage. If parsing succeeds, this stage will produce a parse tree with a structure that closely matches the structure of the formal grammar. While the parse tree already resembles a finished abstract syntax tree, it still contains much syntactic information that is not required in an abstract syntax tree, such as references to separator tokens. Furthermore, a syntactically correct parse tree may still contain semantic errors that have to be detected in the next stage.

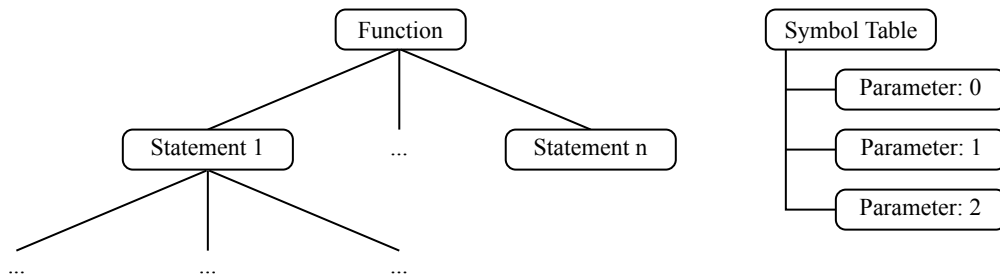
Continuing the above example, the syntax analysis stage will attempt to recursively apply the rules specified by the formal grammar. The token stream shown above could then result in the following fragment of a parse tree.



## Semantic Analysis

This stage analyzes the semantic structure of a parse tree, and produces an abstract syntax tree in the process. A syntactically correct program may still contain many semantic errors such as references to undeclared variables, assignments to constant variables, etc. The semantic analysis stage traverses the parse tree and collects global information (e.g. a symbol table) which enables detection of such semantic errors. During traversal of the parse tree an abstract syntax tree is typically built simultaneously. The abstract syntax tree contains a semantically correct representation of the program that abstracts away many of the syntactic details that are not required during the final stages of compilation.

Continuing our example, a possible abstract syntax tree and symbol table could look as follows.



Note that the abstract syntax tree has a considerably flatter structure than the parse tree, and omits various information present in the parse tree.

## Optimization

The final stage in our simplified compiler optimizes the abstract syntax tree produced by the semantic analysis stage. Optimization is achieved through transformations of the abstract syntax tree, such as constant folding in expressions or dead code elimination. Some of these optimizations require some global information that needs to be computed from the initial abstract syntax tree. For example, dead code analysis requires some sort of dependency information.

## The Programming Language

The programming language that should be supported by your library is based on the educational programming language PL/0 with some simplifications. The major characteristics of this language are:

- Each program defines exactly one function that takes a fixed number of integer parameters and returns a single integer value.
- The language supports exactly one data type, namely 64-bit signed integers.
- All parameters and variables that are used have to be declared at the beginning of a program.
- There are no control flow statements or function calls.
- The body of a program consists of a sequence of variable assignment statements, followed by a single return statement. Statements can contain simple arithmetic expressions.

Informally, a program consists of the following components.

1. Parameter declarations (optional): `PARAM  $P_0, \dots, P_N$ ;`
2. Variable declarations (optional): `VAR  $V_0, \dots, V_N$ ;`
3. Constant declarations (optional): `CONST  $C_0 = c_0, \dots, C_N = c_N$ ;`
4. Body begin: `BEGIN`
5. Program body:  `$statement_0$ ; ... ;  $statement_N$`
6. Body end: `END`
7. Program terminator: `.`

A statement can either be an assignment statement or a return statement.

- Assignment statement: `variable := arithmetic-expression`
- Return statement: `RETURN arithmetic-expression`

Finally, arithmetic expressions can contain the usual unary operators (+ and -), binary operators (+, -, \*, and /), as well as parentheses. They can refer to parameters, variables, constants and integer literals.

### Example

For example, the following program computes the weight of a block of concrete in kilograms given its width, height, and depth in meters.

```
PARAM width, height, depth;
VAR volume;
CONST density = 2400;

BEGIN
    volume := width * height * depth;
    RETURN density * volume
END.
```

### Formal Definition

Formally, the programming language is defined by the grammar shown in Figure 2. The grammar is given in extended Backus-Naur form (EBNF), and precisely defines the syntactic structure of a valid program.

```

function-definition = [ parameter-declarations ]
                    [ variable-declarations ]
                    [ constant-declarations ]
                    compound-statement
                    ". ".

parameter-declarations = "PARAM" declarator-list ";".
variable-declarations = "VAR" declarator-list ";".
constant-declarations = "CONST" init-declarator-list ";".

declarator-list = identifier { ",", identifier }.
init-declarator-list = init-declarator { ",", init-declarator }.
init-declarator = identifier "=" literal.

compound-statement = "BEGIN" statement-list "END".
statement-list = statement { ";", statement }.
statement = assignment-expression
          | "RETURN" additive-expression.

assignment-expression = identifier "!=" additive-expression.
additive-expression = multiplicative-expression
                    [ ( "+" | "-" ) additive-expression ].

multiplicative-expression = unary-expression
                          [ ( "*" | "/" ) multiplicative-expression ].

unary-expression = [ "+" | "-" ] primary-expression.
primary-expression = identifier
                  | literal
                  | "(" additive-expression ")".

```

Figure 2: The EBNF grammar of the programming language supported by the simple compiler. *identifier* can be any combination of lower- and upper-case letters that is not a keyword, and *literal* can contain any combination of digits.

It contains a number of *terminal symbols* which define the basic tokens which can legally be used to write a program. All strings in quotation marks in Figure 2 are terminal symbols, including the keywords PARAM, VAR, CONST, BEGIN, END and RETURN. Furthermore, we consider identifiers and literals to be terminal symbols. An identifier can contain any combination of lower- and upper-case letters that is not a keyword, and a literal can contain any combination of digits. If you are already familiar with EBNF, you can safely skip over the remainder of this section.

In addition to the terminal symbols, the grammar contains *non-terminal* production rules which specify how the terminal symbols can legally be combined. These production rules take the form of assignments, where a specific production rule is assigned to a name. EBNF uses a certain syntax for its production rules, with the following relevant components.

- **Concatenation** (indicated by whitespace-separated terminals or nonterminals). Allows several production rules to be applied in the specified order.
- **Termination** (indicated by a dot .). Terminates the definition of a production rule.

- **Alternation** (indicated by a pipe |). Allows exactly one of several production rules to be applied.
- **Optional** (indicated by square braces [...]). The enclosed production rule may be applied zero or one times.
- **Repetition** (indicated by curly braces {...}). The enclosed production rule may be applied zero or more times.
- **Grouping** (indicated by round braces (...)). Makes grouping of production rules explicit.

### Example

Consider, for example, the following production rule named `assignment-expression`.

```
assignment-expression = identifier "!=" additive-expression.
```

This rule expresses that an assignment expression consists of an `identifier` terminal symbol, followed by the token `!=`, followed by an `additive-expression` nonterminal. The `additive-expression` is in turn defined as follows.

```
additive-expression = multiplicative-expression
                      [ ( "+" | "-" ) additive-expression ].
```

That is, an additive expression must contain a `multiplicative-expression` nonterminal as the left hand side, and may optionally contain a right-hand side which may be one of the tokens `+` or `-` followed by another `additive-expression` nonterminal. Continuing recursively in this fashion, the full syntax of a valid assignment expression is specified.


## Milestone 1

(5 points)

Before we can start working on the actual compiler, we need to implement some infrastructure for source code management and error printing. Compilation errors can occur in any one of the lexical analysis, syntax analysis, and semantic analysis stages. If such an error occurs, we would like to be able to print some useful information that aids the user in debugging the error. Thus, we need to maintain a mapping to the original source code even in later stages that do not operate on the original source code anymore (e.g. in the semantic analysis stage which operates on parse trees).


In order to avoid duplicating the required logic, this functionality should be centralized so that it can be used by the various compilation stages. Instead of storing actual source code fragments in the later stages, e.g. in the parse tree, this allows us to store lightweight references into the original source code. Besides reducing code duplication, this approach also avoids duplicating source code fragments throughout the compilation process, which helps avoid inconsistencies.

The infrastructure should provide the functionality to resolve such lightweight references into line numbers and positions in the actual source code. Furthermore, it should be able to print context for a location in the source code, so that printing error messages like the following is possible.



```
6:19: error: expected ')'  
  c := -a * (b + d;  
              ^  
6:13: note: to match this '('  
  c := -a * (b + d;  
              ^
```

Furthermore, we would also be able to print context for a range of source code, for example in an error message as follows.



```
7:5: error: expected ':='  
  a RETURN b  
    ^~~~~~
```

Note that your implementation of context and error printing does not have to follow these examples exactly, as long as it conveys the salient information to the user.

## Algorithms

No specialized algorithms are required in this milestone.

## Requirements

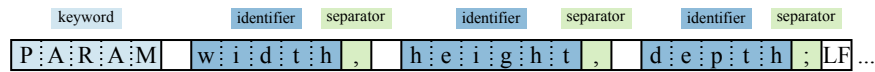
- (a) Define and implement a suitable type for centralized source code management as outlined above.
- (b) Define and implement suitable types that serve as a reference to a location or a range in the original source code.
- (c) Implement functionality to resolve a location reference to a line number and character position within that line.
- (d) Implement functionality to print context for a location or range reference as outlined above.



## Milestone 2

(15 points)

This milestone covers the lexical analysis stage of the compilation process which is also called the *lexer*. As outlined above, the purpose of the lexer is to transform the raw source code into a stream of valid tokens of the programming language, as shown in the following figure.



Any terminal symbol of the grammar constitutes a valid token. In order to facilitate syntax analysis in the next stage, it is beneficial to gather as much information as possible in this stage. For example, terminal symbols should be grouped into several broad categories, such as identifiers, literals, operators etc (indicated by different colors in the figure). Within these categories, tokens can sometimes be differentiated in even more detail, e.g., an operator token can be a plus operator, a minus operator etc. Of course, each token also needs to store information about its content (i.e. the actual characters contained in the token).

Once you have defined suitable types to represent tokens, the actual transformation of the source code to tokens can be done in a single pass over the source code. This can be achieved with the algorithm described below.

### Algorithms

The lexer should scan the source code character-by-character. In order to produce the next token, you can employ the following algorithm:

1. Advance the current position in the source code until a non-whitespace character is found.
2. Examine the character at the current position to determine the type of the next token. The terminal symbols of our programming language are chosen in such a way that the first character of a token unambiguously determines its type (i.e. no backtracking or lookahead is required).
3. Greedily add as many of the following characters as possible to the token, until
  - adding another character would result in an invalid token, or
  - a whitespace character is found, or
  - the end of the source code is reached.

For example, the string `1234` should be lexed as a single literal token, instead of several smaller literal tokens. However, the string `1234a` should be lexed as a literal token followed by an identifier token, since adding `a` to the literal `1234` would result in an invalid token. The string `12 34` should be lexed as two literal tokens as the first token is terminated by a whitespace character after the character `2`.

4. Advance the current position in the source code to the next character that has not been lexed yet.

If the lexer encounters a character that cannot be part of any legal token (e.g. the question mark character `?`), or if it encounters an unexpected character (e.g. in a multi-character operator), it should report an error and the compilation process should be terminated.

### Requirements

- (a) Define and implement suitable types that identify the different types of tokens (see above).
  - Tokens should use references to the original source code (see previous milestone) instead of directly storing their characters.

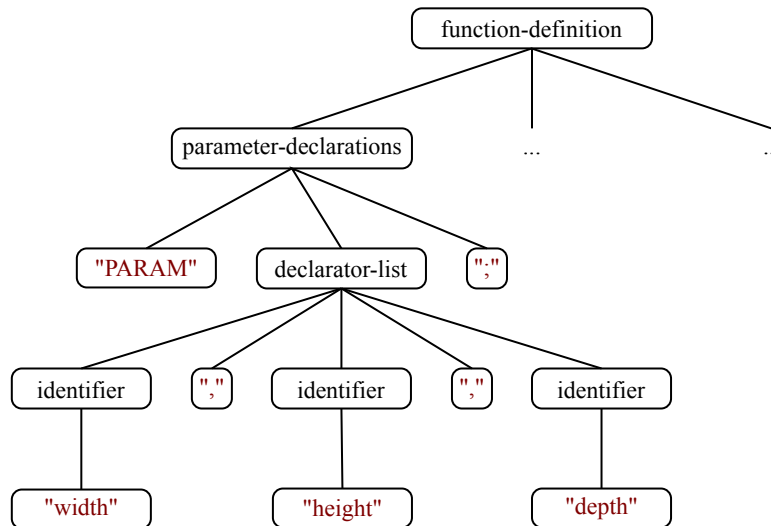
- Note that at some point during the compilation process, the actual integer value of literal tokens needs to be computed. This could either be done during lexing, or during semantic analysis, both of which are reasonable options.
- (b) Define and implement a lexer that scans the original source code and produces a stream of tokens.
- The lexer should not include any whitespace characters in its output (cf. also step 1 of the above algorithm). We consider spaces, tabs and newline characters to be whitespace.
  - Note that whitespace characters still serve an important purpose as a separator for tokens in the input as described in step 3 of the algorithm.
  - The lexer should use the infrastructure implemented in the previous milestone to notify the user about any errors.
  - Note that our programming language can be parsed without backtracking or lookahead, so the subsequent stages need to look at only one token at a time. That is, you can implement the lexer in a way that produces only the next token on demand (e.g. in a `next` method), instead of lexing the entire input at once.

You should write tests for your lexer that verify that it produces the expected output. Ensuring that the lexer works as expected will make implementing the next milestone much easier.

### Milestone 3

(25 points)

In this milestone, you will transform the token stream produced by the lexical analysis stage into a parse tree, which constitutes the syntax analysis stage of the compilation process. A fragment of a possible parse tree is shown in the following figure.



For this we first need to define and implement the parse tree data structure. The parse tree should consist of exactly one node type for each non-terminal or terminal symbol type in the formal grammar of the programming language.

Nodes that represent terminal symbols of the grammar do not have any children, and simply contain information about the terminal symbol they represent. A sensible way of representing terminal symbols could be to differentiate between identifiers, literals, and generic tokens which comprise the remaining terminal symbols (i.e. operators, keywords and separators).

While it would be both possible and valid to do so, we do not need to distinguish between operator, keyword, and separator nodes in the parse tree. Before creating such a node, the parsing algorithm described below will check that the corresponding token has the correct type. Thus, we can deduce the type of these token nodes from their position in the parse tree. For example, the parse tree node for an assignment statement will have one generic token node among its children which represents the assignment operator. If the parsing algorithm succeeded, we can know for sure that this generic token node must contain an assignment operator.

Nodes that represent non-terminal symbols of the grammar have children nodes that correspond to the expansion of the production rule associated with the non-terminal symbol. For example, the parse tree node for an assignment statement will have three children, corresponding to the **identifier**, **"::="**, and **additive-expression** components of the respective production rule in the grammar.

If a production rule contains alternation (i.e. the pipe operator **|**), the corresponding node needs to store additional information about which alternative it represents. For example, the node type for the **primary-expression** non-terminal can either be an identifier, a literal, or a parenthesized expression. In such cases, a sensible solution could be to store all children in a vector, and have an additional data member in the node class. In case of the **primary-expression** non-terminal, this member can then be used to store whether the corresponding node represents an identifier, a literal, or a parenthesized expression.

Finally, nodes representing non-terminal symbols also need to account for optional or repeating production rules through similar measures.

## Algorithms

With the parse tree definition in place, we can turn our attention to the actual process of parsing, i.e. transforming a token stream into a parse tree. Fortunately, the structure of our grammar allows for a straightforward parsing algorithm, since it is possible to make all decisions that are required during parsing simply by looking at the next token (formally, it is an LL(1) grammar).

In particular, you will implement a *recursive-descent parser*. A recursive-descent parser has exactly one function for each non-terminal symbol of the grammar which attempts to construct the corresponding parse tree node from the token stream. These functions all operate according to the following general principle:

1. If the production rule contains alternation on the top level, look at the next token to determine which of the alternatives should be used. For example, in case of the **statement** non-terminal symbol, we parse a return statement if the next token is the **RETURN** keyword, and an assignment statement otherwise.
2. Parse the production rule component-by-component, applying the following steps for each component:
  - If the component is a terminal symbol, check that the next token matches the expected token. If a mismatch occurs, an error should be reported and compilation should be terminated. Otherwise, create a parse tree node representing the terminal symbol.
  - If the component is a non-terminal symbol, create a parse tree node representing the non-terminal symbol by recursively calling the parsing function for the respective non-terminal symbol.
3. If parsing of all components succeeded in the previous step, build and return a parse tree node with the child nodes created in the previous step.

Some care has to be taken for production rules which contain optional components. For example, parsing the **function-definition** non-terminal symbol will always start by attempting to parse a **parameter-declarations** non-terminal symbol. However, since this component of the production rule is optional, parsing of the **parameter-declarations** non-terminal symbol may fail without immediately aborting compilation.

## Requirements

- (a) Define and implement node types for the parse tree.
  - There should be one node type for each terminal and non-terminal symbol in the formal grammar.
  - No sophisticated semantic inheritance hierarchy is required for the parse tree nodes. It is sufficient to have a simple inheritance hierarchy that focuses on reducing code duplication.
  - Each parse tree node should maintain a reference to the original source code which encompasses all characters that correspond to the node. This is required for accurate error reporting in the subsequent stages.
  - It should be efficient to determine the type of a parse tree node without relying on `dynamic_cast` which may be useful in the next milestone.
- (b) Define and implement a recursive-descent parser for our programming language.
  - The parser should publicly expose a function that can parse the **function-definition** non-terminal symbol which is the top-level symbol in our grammar.

- Internally, the parser should implement the recursive-descent algorithm outlined above to achieve this.
  - The parser can either take the token stream as a parameter of some sort, or manage the creation of a lexer itself.
- (c) Define and implement a parse tree visitor base class. It should have a pure virtual `visit` member function for each non-abstract node type. All nodes should have a virtual `accept` member function that takes a parse tree visitor and calls its `visit` function with a reference to itself. Implement a visitor that prints the parse tree in the DOT format known from the programming assignments.

You should write tests for your parser that verify that it produces the expected output. The DOT printing functionality can be extremely useful for visually debugging errors. It can also be beneficial to test that the parser correctly detects syntax errors in ill-formed programs. Ensuring that the parser works as expected will make implementing the next milestone much easier.

## Milestone 4

(20 points)

In this milestone you will implement the Abstract Syntax Tree (AST) that is a better description of the *semantics* of the code as opposed to its structure in the parse tree. The process of turning a parse tree into an AST is called *semantic analysis* which you will also implement.

Just like the parse tree, the AST can be represented by objects that represent a node in the tree. Each object should also know about its children to make tree-traversal possible. Unlike the parse tree, the structure of the AST should not be tightly coupled to the grammar, however. In the parse tree you mostly implemented one node type for each symbol in the grammar. For the AST each node class should instead represent a semantic concept. The parse tree, for example, has an additive expression node that can have a multiplicative expression node as its child. Logically it is not necessary to separate between additive and multiplicative operations. An appropriate AST class could represent all four binary arithmetic operations in one node class. For this milestone you have to come up with a class hierarchy that represents the semantic structure of the program.

The semantic analysis is also responsible to find and report semantic errors in the program, like using an undeclared identifier. Those errors cannot be prevented by only enforcing the grammar which is why the parser will not find them. For this you will need to implement a *symbol table* which keeps track of all existing identifiers and their properties, e.g., whether they are constants.

## Algorithms

For the semantic analysis you need to traverse the existing parse tree and generate an AST for it. This can be done by writing functions that take a node from the parse tree, recursively call each other with the child nodes, and return the generated AST nodes. There could be a function called `analyzeStatement()` that takes a reference to an object that represents a statement from the parse tree. It should return an AST node generated from this statement. To analyze the parse tree statement, this function could then use a function called `analyzeExpression()` to get an AST node that represents the expression on the right-hand side of the assignment. There should be several more, similar functions that potentially call each other and eventually generate a complete AST out of the parse tree.

Just like for the parse tree, it is also desirable to be able to generate a visual representation of the AST. This again can be implemented nicely with the visitor pattern.

## Requirements

- (a) Define and implement node types for the AST:
  - There should be classes that represent *statements*, *expressions*, and *functions*.
  - There should be subclasses for different kinds of statements and expressions.
  - It should be efficient to determine the type of an AST tree node without relying on `dynamic_cast` which may be useful in the next milestone.
- (b) Define and implement a class that represents the symbol table. It should keep track of all identifiers that are declared in the program.
  - The symbol table should keep the source code location of each declaration to enable better error messages.
  - It should be efficient to find the declaration given the name of an identifier.
- (c) Implement the semantic analysis. It should take the root node of a parse tree and eventually return the root node of the corresponding AST. If any semantic errors are found, they should be printed with an error message that points to the error in the source code. Your implementation should be able to find at least the following errors:

- The same identifier being declared twice
  - Using an undeclared identifier
  - Assigning a value to a constant
  - Using an uninitialized variable (constants and parameters are always considered to be initialized)
  - Missing return statement
- (d) Implement an AST visitor base class that has a pure virtual `visit` member function for each non-abstract AST node type. All nodes should have an `accept` member function that takes a visitor and calls its `visit` function with a reference to itself. Implement a visitor that prints the AST in the DOT format known from the programming assignments.

You should write tests for your semantic analysis that verify that it produces the expected output. The DOT printing functionality can be extremely useful for visually debugging errors. It can also be beneficial to test that the semantic analysis correctly detects semantic errors in ill-formed programs. Ensuring that the semantic analysis works as expected will make implementing the next milestone much easier.

## Milestone 5

(20 points)

A compiler usually takes an AST and converts it into an internal, low-level language called *intermediate representation* or *IR* which can then be executed by a virtual machine. You will *not* have to implement an IR. To still be able to execute a program, you will have to extend your AST so that it can execute the program it represents, instead.

To execute a program, your implementation should step through all statements and execute them in order. It also has to keep track of all variables and their current value. When the execution reaches a return statement, the return value must be computed. Then, the execution should be stopped. The execution should also stop and print an error message when the program attempts to divide by zero.

For this milestone you will also have to implement optimizations that transform the AST. Optimizations are usually separated into logically distinct *passes*. Each optimization pass implements one kind of optimization. Then, multiple passes are combined and executed in sequence to generate an optimized program. Two of the most common optimizations are *dead code elimination* and *constant propagation*.

Dead code elimination tries to remove all code that will never be executed. In our programming language this is exactly all code that comes after the first return statement. This can be implemented by walking over the AST and stopping when the first return statement is found. All following statements can then be removed.

Constant propagation is an optimization that tries to pre-calculate as much as possible to make the execution of the program faster. It transforms the program by pre-calculating the results of all expressions that can be computed without having to execute the program and then replacing those expressions with constants. For this, the optimization must go over the AST and find out which expressions can be pre-calculated. It also has to keep track of when variables are overwritten by a new constant or non-constant expression to accurately determine how expressions can be pre-computed.

## Algorithms

To implement optimization passes it may be useful (but not necessary) to use the visitor pattern.

For the constant propagation you can use the following algorithm:

1. Initialize a table that has an entry for every variable that contains its currently known constant value if it is constant.
2. Initialize an empty mapping that maps expressions to their potentially constant value.
3. Iterate over all statements in their program order. For each statement:
  - Traverse the expressions used in the statement in post-order. Check if the expression can be evaluated. For expressions that refer to a variable, check the table of variables. For expressions with other expressions as children, check the mapping of expressions. If the expression can be evaluated, evaluate it and update the mapping of expressions.
  - If the statement is an assignment statement, update the table for the variable that is assigned to accordingly.
4. Iterate over all assignment and return statements. Replace all expressions that are known to be constant by their constant value by using the expression mapping.



## Requirements

- (a) Define and implement a class used for the evaluation of AST nodes. This class should store all variables used in the program. After executing a program, it should be possible to extract the return value from this class. It should also be possible to determine whether an error occurred during execution (i.e. division by zero).
- (b) Modify your AST node classes so that they can use the class defined in requirement (a). They should use it to execute the program as described above by passing it recursively to their child nodes.
- (c) Define and implement an abstract class that represents an optimization pass. It should have at least one member function that takes an AST and performs an optimization on it.
- (d) Define and implement an optimization pass that implements dead code elimination as described above.
- (e) Define and implement an optimization pass that implements constant propagation as described above. Note that you do not need to implement constant propagation that would require reordering of operands, e.g.,  $a + (1 + 2)$  should be optimized to  $a + 3$  but  $(a + 1) + 2$  can be left unoptimized (assuming  $a$  is not constant, of course).

You should write tests for the evaluation and optimization of the AST. The tests should include a few programs that are evaluated and tested for the correct return value. It is also useful to write programs that lead to an error. You should also test the optimization passes by writing a few programs for every pass and check if the pass transformed the program as expected.

## Milestone 6

(15 points)

For the last milestone you will write code that ties all of the previous milestones together. In the end, a developer that uses your compiler framework should only have to use few classes and functions to turn a string that contains the program source into a function that can be called. You can take the following code snippet as a motivation for your implementation:

```
Pljit jit; // Create an object that manages just-in-time
          // compilation (jit) of functions
// Register a function with its source with the jit object
auto func = jit.registerFunction("PARAM a, b BEGIN [...] END.");
// Call the PL/0 function with the arguments 123 for a and 456 for b
auto result = func(123, 456);
```

In essence, you should implement a single class that allows for easy just-in-time (JIT) compilation of functions in our programming language. It should be possible to register functions with their source code and then call them with arguments. To really be considered JIT-compiled, your implementation should not start parsing and analyzing the function source before it is called for the first time. Once a function is registered, it should also be possible to call it simultaneously from multiple threads.

This can be implemented by using *handles* to registered functions. When a function is registered, your JIT class should only return such a handle to the function. The resources that are required by the function (e.g. the AST nodes) should be kept by the JIT class. The handle only acts as a reference to those resources. This also has the advantage that a handle is cheaper to copy (e.g. into different threads) compared to copying the resources needed by a function.

When a function is called for the first time through a function handle, it should be compiled and optimized. Because the handle can be copied to multiple threads before the function was compiled, the implementation must make sure that it handles multiple threads calling and potentially compiling the function correctly. Ideally, once the function was compiled by any thread and later other threads call this function, it should not be compiled again.

## Algorithms

To implement the thread-safe compilation of functions you can use mutexes or a more efficient lock-free strategy that relies only on atomic operations.

To make it possible to call a function handle like in the code snippet, you can use a variadic template for the `operator()` that can take an arbitrary number of arguments. However, you could also simply pass a vector containing the parameters to the function handle.

## Requirements

- (a) Define and implement a JIT class that handles registering functions with their source code as described above. Make sure that the header file that is needed to use this class requires as few transitive includes as possible.
- (b) Define and implement a class that represents a function handle and is returned by the JIT class when registering a function. As described above, a function should only be compiled when it is called for the first time. Make sure that different handles to the same function can be used concurrently.

Write tests that use your JIT class to register and call a few functions. You should also test that the function handles can be passed to multiple threads and then called without causing issues.