**The Amethyst Compiler**

A Custom Language & Compiler to LLVM IR

David Britton

Honors Project  
Department of Computer Science  
Fall 2024

**Table of Contents**

**Abstract**4

I. Background5

Programs & Abstraction5

Compiler Overview6

The Compiler Pipeline6

II. LLVM8

The LLVM Project8

LLVM IR9

Amethyst and LLVM11

III. The Amethst Language12

Programs12

Primitive Data Types, Variables, & Literals12

Comments13

Functions13

Global Variables14

Flow Control14

Pointers & Subscript Access15

Stack & Heap Expressions16

Primitive Operators & Assignments16

User-Defined Types17

Members17

Methods18

Constructor Definitions & *New* Expressions18

Operator Definitions19

Assignment of User-Defined Types20

IV. Concepts & Application in Amethyst21

Regular Expressions21

Tokens & Lexing23

Grammars25

Abstract Syntax Trees26

Recursive-Descent Parsing27

The Visitor Design Pattern29

Semantic Analysis31

Code Generation33

Bibliography35

Appendix36

Implementation of a Dynamic Array of Integers36

Implementation of a Matrix of Integers37

**Abstract**

Amethyst is a custom programming language, whose compiler translates Amethyst source code into textual LLVM IR, which can, in turn, be compiled by LLVM back-ends to produce executable binaries. This paper explores compiler concepts and implementation details for the Amethyst compiler, describes the LLVM architecture, and provides an overview of some Amethyst language features.

**I. Background**

**Programs & Abstraction**

A screenshot of a computer program

Description automatically generatedA computer program is a series of instructions that outline tasks for a computer to perform. Programs are written in *programming languages*, which have rigid vocabulary and syntax that correlate unambiguously to intended behavior (Cooper and Torczon 2).That is, the same sequence of instructions with the same input must produce the same result.

These languages are often classified as *high-level* or *low-level*, referring to their degree of *abstraction* from machine code. A machine code instruction is the smallest unit of computation that a CPU can be instructed to perform, such as adding two integers or storing a value into memory. A high-level, or highly abstract language is one that is far removed from machine code. One instruction in a high-level language may map to several hundred machine instructions. On the other side of the spectrum are assembly languages, the lowest-level human-readable programming languages. Assembly instructions often correspond one-to-one in a mapping to machine instructions.

*Figure 1. Snippets of the same function in multiple languages. Ordered from most to least abstract, the languages are Python, C, LLVM IR, and x86-64 Assembly. The LLVM IR and x86-64 Assembly were generated by compiling the example C function with* Clang*.*

A white rectangle with black text

Description automatically generated**Compiler Overview** A *compiler* is a piece of software that translates source code from one programming language into a target language (Cooper and Torczon 2).Typically, compilers translate source code from a high-level language to a lower-level language. The term *transpiler* might be used to refer to a *source-to-source* translator - that is, one that translates a program from language to another at a similar level of abstraction. In the particular case of the Amethyst compiler, the source language is Amethyst and the target language is LLVM IR.

*Figure 2. Compiler input and output.*

**The Compiler Pipeline**

The compilation process is often broken down into two constituent parts, a front-end and a back-end. The front-end of a compiler typically performs analysis on the source code in three stages. The first stage is lexical analysis, often referred to as lexing or tokenization. In this stage, the source text is broken into a sequence of *tokens*, which can be thought of as the words and punctuation of the language. The next stage is syntactic analysis, or parsing, which verifies that the sequence of tokens conforms to the grammar of language. In Amethyst and other compilers, this stage may also produce an *intermediate representation* (IR), which facilitates the rest of the compilation process. The final front-end stage is semantic analysis, which confirms that the content of the program is meaningful. For example, a semantic analyzer might verify that all variables are declared before they are used, or that no variables are assigned a value of the wrong data type (Appel and Ginsberg 16). After the front-end stages are complete, the IR may optionally be optimized to improve the performance of the target output. Code generation produces the target output. These two processes generally constitute the back-end of the compiler.

A white rectangular sign with black text

Description automatically generated*Figure 3. A typical compiler pipeline.*These stages and Amethyst’s implementation of them are discussed further in section IV.

**II. LLVM**

*A diagram of a diagram

Description automatically generated***The LLVM Project** LLVM is a compiler infrastructure, a collection of compiler back-ends, libraries, and tools, conceived by Chris Lattner and Vikram Adve in 2000 at the University of Illinois. Although LLVM was originally an acronym for “Low-Level Virtual Machine”, this is no longer case, and LLVM is simply the name of the project. LLVM’s most useful tool is its intermediate representation, LLVM IR, which serves a common interface for compiler back-ends. Because of this common interface, any compiler front-end that produces LLVM IR can make use of an LLVM back-end for any target.

*Figure 4. A demonstration of LLVM’s modularity.*

As such, LLVM-based languages should be platform-independent. Even the so-called “middle-end” IR analyzers and optimizers are modular and can be substituted into a compiler pipeline as appropriate.

LLVM sub-projects include the core libraries that provide optimization and code generation for compiler back-ends; Clang, a well-known C, C++, and Objective-C compiler; and the LLDB debugger. Several large programming languages have LLVM-based compilers, notably Swift and Rust.

**LLVM IR**

LLVM IR has equivalent textual, binary, and in-memory representations (Lattner and Adve). Its in-memory representation is used internally, for example, when using LLVM core library functions to optimize IR output. Running clang <filename> -S -emit-llvm on a C source file outputs textual LLVM IR to a .ll file. Replacing -S with -c outputs the equivalent binary LLVM IR to a .bc file.

A screenshot of a computer code

Description automatically generatedLLVM IR is itself a programming language. It has no variables, instead using static single assignment (SSA) with an infinite number of virtual registers. This means that every register is guaranteed to be initialized by a single expression and depends only on the values of registers assigned before its definition (Young).

*Figure 5.* *Equivalent function definitions in C (left) and unoptimized LLVM IR (right). LLVM IR comments, preceded by semicolons, show the relationship between the two. Note that the value of variable* i *is stored in multiple registers (%6 and %7) over the course of the program. Any use of a variable’s value in C maps to a load of its value into a new register in LLVM IR.*

A number and arrows on a white background

Description automatically generatedLLVM IR provides an internal type system with primitive, pointer, and struct types. These structs do not function like structs in higher-level languages, but are pointers with knowledge of the struct’s size and the byte offsets of its elements. Arrays function similarly. The getelementptr instruction, as its name implies, retrieves the address an element of a struct or an array, given an integer index. The elements of an LLVM IR struct are indexed sequentially regardless of their size in memory.

*Figure 6. Despite differing sizes in memory, the indices of the elements of the two structs are both sequential.*

The alloca instruction allocates space on the stack, and the returned memory address can be used later in the program to store and load values with the store and load instructions respectively. Arithmetic operations on primitives are performed with instructions such as add for integer addition, frem for floating point modulus, and icmp for integer comparisons. Control flow is facilitated by jump labels. LLVM IR provides an abstraction for function definitions and calls. This abstraction is limited and does not include function overloading. Special reference parameters are required to return large structs. Functions such as memcpy(), malloc(), and free() – along with other familiar C standard library functions – can be called, provided that they are appropriately declared in the LLVM IR. A comprehensive list of language features is found in the LLVM Language Reference Manual.

**Amethyst and LLVM**

The Amethyst compiler does not directly use any LLVM libraries or tools. It instead generates textual LLVM IR as its target output. This approach is non-standard, but in Lattner’s own words, “Since LLVM IR has a first-class textual form, it is both possible and reasonable to build a front end that outputs LLVM IR as text, then use Unix pipes to send it through the optimizer sequence and code generator of your choice” (Lattner). For clarity, however, LLVM IR is not the intermediate representation of the Amethyst compiler, but the target output. Amethyst’s own IR is an in-memory abstract syntax tree (discussed further in section IV). The LLVM IR output may be fed to a back-end of the user’s choice, but no back-ends are explicitly part of the Amethyst pipeline.

**III. The Amethyst Language**

**Programs**

All programs in Amethyst are a series of the following declaration types: function definitions, type definitions, and global variable declarations.

A black and white text

Description automatically generated**Primitive Data Types, Variables, & Literals**

*Figure 6. A simple Amethyst program containing variable definitions.*

There are four primitive data types in Amethyst: int, float, bool, and char. The types int and float are translated to the 64-bit representations i64 and double in LLVM IR, so a 64-bit system is required to run a compiled Amethyst program. There is no implicit type conversion or type inferencing.

Variable definitions are of the form *name* : *type = expression*. int and float literals are represented, as expected, with digits, and may be negative if immediately preceded by a negative ( - ) sign. A number with no decimal point will be treated as an integer literal in all cases, so to represent a float literal with the value 12, the programmer must be explicit and express the value as 12.0.

bool literals are represented with the keywords true and false. char literals are enclosed by single quotes ( ' ). The escape sequences \\, \n, \t, and \0 are valid and operate as in C. The single quotes must contain only one character unless they contain an escape sequence.

**Comments**

Comments begin with the hash symbol ( # ). Any text following a hash will be ignored by the compiler, until the next newline. *Figures 7 and 8 contain comments.*

A screenshot of a computer program

Description automatically generated**Functions**

*Figure 7. An example Amethyst program demonstrating variable function definitions, non-void returns, and function calls.*

A function definition begins with keyword def. It is followed by a name, a comma-separated parameter list enclosed by parentheses, and – for non-void functions – a colon and a return type. The body of a function must begin on a new line, and each statement in a function body is separated by a line break. Finally, the function scope is closed with keyword end on its own line. It should be noted here that in Amethyst, the only syntactically significant whitespace is a newline break.

The main() function serves as the entry point of the program. A function call is simply the function’s name followed by a comma-separated, parenthesis-enclosed argument list, as shown in the expression of the return statement return power(3,4). In a return statement, the type of the evaluated expression must match the return type of the function in which it appears. In void functions, keyword return is not followed by an expression.

A white background with black text

Description automatically generated**Global Variables**

*Figure 8. A demonstration of global variables and control structures. The program’s exit code is 202.*

Global variables, whose names are preceded by the global sigil ( $ ), are declared in global scope. Amethyst does not support compile-time expression evaluation, so they cannot be assigned values at declaration. Global variables can be otherwise used as normal within function scope.

**Flow Control**

Amethyst supports while loops, but not any other type of loop. Keywords while and do enclose the loop condition. Much like function bodies, the loop bodies begin on new line and are terminated by keyword end on its own line.

Three statements exist to direct flow control within the innermost while loop, each composed of a single keyword. They are: continue, break, and redo. Continue statements jump to immediately before the evaluation of the loop condition. Break statements jump to the end of the loop body. Redo statements jump to the beginning of the loop body, so the condition will not be re-evaluated.

Conditional branching is performed with if blocks, elsif blocks, and else blocks. An if block begins with keyword if, followed by a condition and keyword then. The body of a block begins on a new line and executes if the condition is true. Elsif blocks, with similar syntax but beginning with keyword elsif, optionally follow an if block. The body of an elsif block will execute if its condition is true and all previous conditions are false. If none of the conditions are true, an optional else block can be used to execute default instructions. The entire structure, regardless of the presence of elsif blocks or an else block, always ends with a standalone keyword end.

While loop, if, and elsif conditions must result in bool values. *See Figure 8 for a demonstration of flow control structures.*

**A white background with black text

Description automatically generatedPointers & Subscript Access**

*Figure 9. A program that allocates a 4x4 int matrix on the stack, demonstrating pointers, subscript access, and stack expressions.*

Pointer types are indicated by one or more asterisks ( \* ) following a type name, where multiple asterisks indicate pointers to pointers. For example, an int\*\* is a pointer to a pointer to an int. Arrays are referenced by pointers with no other syntactic distinction. All dereferencing is performed with the subscript operator, indicated by square brackets ( [] ), and the expression enclosed in the square brackets must result in an int.

**Stack & Heap Expressions**

A black and white text

Description automatically generatedThere are two novel language features in Amethyst: stack and heap expressions. Stack expressions begin with keyword stack, followed by an integer enclosed in square brackets and a type name. Stack expressions allocate the space needed to store the specified number of instances of the type, returning a pointer to the allocated memory. *Refer to Figure 9 for a demonstration of pointers and stack expressions.*

*Figure 10. A simple program containing a heap expression and an unheap statement.*

Heap expressions are similar, but begin with keyword heap, and the expression enclosed in the square brackets is not limited to int literals. As the name suggests, they allocate space on the heap. Heap expressions are, in fact, just an abstraction on malloc(), so any memory allocated should be freed with an unheap statement. Unheap statements, which under the hood are a call to free(), are simply composed of keyword unheap and a pointer.

**Primitive Operators & Assignments**

Each primitive type has a set of valid operators, each with differing sets of acceptable lefthand and righthand types. An exhaustive list of accepted operator-type combinations would be impractical (and frankly unhelpful) to include here due to their sheer number. To demonstrate just one, booleans are limited to equality comparisons and logical operators. So, for a given bool named flag, the expression flag and true is semantically valid, but flag + false is prohibited.

Operator precedence in Amethyst is adapted from ANSI C Grammar. In the expression  
a == 1 and b == true, each equality expression evaluates before the logical ‘and’ expression. Exhaustive operator precedence rules would be similarly impractical to list here but can be inferred from the grammar in the Amethyst github repository.

Assignment in Amethyst is performed with the equals sign ( = ). There is no type inferencing, so the resultant types of the lefthand and righthand expressions must match. Assignment is a statement rather than an expression, so assignments cannot be chained. For example, a statement like a = b = c is invalid.

**User-Defined Types**

Type definitions are composed of member declarations, method definitions, constructor definitions, and operator definitions. Type definitions begin with keyword type and the type’s name. The scope of the definition begins on a new line and ends with keyword end on its own line. *The appendix contains implementations of a few common data structures which make use of features that may not be shown in this section’s figures.*

A group of black text

Description automatically generated**Members**

*Figure 11. A definition of a type with two members.*

Members (member variables) are declared using the member sigil ( @ ). Like global variables, they cannot be assigned a value at declaration, so constructors (discussed later) are required to initialize them. The member sigil ( @ ) is used in type definitions to refer to members of the current instance of the type (like this-> in C++). *See the return statement in Figure 12.*

The dot operator ( . ) is used to access the members of an instance of a type outside of its definition, as would be familiar from C++. *See the return statement of main() in Figure 13.*

**A close up of a computer code

Description automatically generatedMethods**

*Figure 12. A type definition with a simple method that returns a member’s value.*

Method definitions have syntax similar to function definitions, the difference being that the method name is preceded by the member sigil ( @ ). Within a type definition, the member sigil is used to call methods of the current instance (like this-> in C++). Outside a type definition, the dot operator is used to call a method of an instance, as would be familiar from C++.

A computer code with black text

Description automatically generated**Constructor Definitions & *New* Expressions**

*Figure 13. An example program, including a constructor definition, a* new *expression, and both internal and external usage of a member variable. The program’s exit code is 4.*

Constructors are specialized methods used to initialize the members of an instance of a type. Constructor definitions begin with keyword new, followed by a parameter list. No return type follows the parameter list, and return statements in constructor scope are not followed by expressions.

Constructors are invoked by *new* expressions, which return a stack-allocated, initialized instance of the type. *New* expressions begin with keyword new, followed by the type name and an argument list. They cannot be used to initialize primitives.

A computer screen shot of a computer code

Description automatically generated**Operator Definitions**

*Figure 14. A type definition with a constructor and two definitions of the same operator. The constructor and the operator overloads are used in main(). The program’s exit code is 12.*

Operator definitions allow operators to be used on instances of user-defined types. They begin with keyword op, followed by the operator whose behavior is to be defined, a single parameter, and a return type. Each operator-parameter combination can only be defined once within a type definition. The defined procedure will be executed when the operator is used with the lefthand expression resulting in the defined type and the righthand expression resulting in the parameter type. Note in Figure 14 that op+ is defined twice, but with different parameters.

A screenshot of a computer code

Description automatically generated**Assignment of User-Defined Types**

*Figure 15. A program demonstrating shallow copy during assignment of variables user-defined types. The program’s exit code is 123.*

Assignment of user-defined types is, under the hood, an invocation of the LLVM intrinsic function, memcpy(). That is, in an assignment of a user-defined type, the value from the righthand expression is directly written to the memory location of the lefthand expression. All assignments are, then, shallow copies. If a deep copy is needed, a function or method must be written to perform it.

**IV. Concepts & Application in Amethyst**

**Regular Expressions**

A useful formalism for lexical analysis is that of *regular expressions*, which are a mechanism for defining sets of strings (composed of symbols) that match particular patterns. The set of symbols on which a regular expression operates is called an *alphabet* (Appel & Ginsburg 18). For Amethyst, the alphabet is the set of ASCII characters, and regular expressions are used to describe the form of keywords, identifiers (variable names, function names, etc.), operators, and punctuation. The examples for the following term definitions use the ASCII character set as their alphabet.

* *Symbols* are the basic unit of a regular expression and are the elements that constitute an alphabet. The regular expression (a) describes the set {“a”}.
* *Concatenation,* indicated by consecutive regular expressions, results in a set of strings created by appending each of the strings defined by the latter expression to each of the strings described by the former. (ab) defines the set {“ab”}.
* *Alternation*, marked by a pipe ( | ), describes a union of two sets of strings. The regular expression (a|b) defines the set {“a”, “b”}. A series of alternations can be expressed by enclosing a series of consecutive sets in square brackets, so [adg] is equivalent to (a|d|g) and defines the set {“a”, “d”, “g”}. Concatenation takes precedence over alternation, so (a|cd) defines the set {“a”, “cd”}. Combining every feature so far, (a[bc]|de)(2|3) defines the set {“ab2”, “ac2”, “de2”, “ab3”, “ac3”, “de3”}.
* *Ranges*, represented by a dash ( - ) between two symbols, is a shorthand for alternations of ordered symbols, like letters and digits in the ASCII character set. [a-d] is equivalent to [abcd] and (a|b|c|d). Ranges are treated as any other expression. For example, [\_a-cA-C] defines the set {“\_”, “a”, “b”, “c”, “A”, “B”, “C”}.
* *Repetition*, indicated by an asterisk ( \* ) after a regular expression, represents the union of the empty string with the set produced by repeated self-concatenations of that regular expression. For instance, (a\*) defines the set {“”, “a”, “aa”, “aaa”, …}. Repetition takes precedence over concatenation, so (ab\*) defines {“a”, “ab”, abb”, …}. A particularly useful construct is the combination of ranges and repetitions. ([a-c]\*) defines {“”, “a”, “b”, “c”, “aa”, “ab”, … “cc”, … “aaa”, “aab”, … “ccb”, “ccc”, “aaaa”, …}.
* *Epsilon* ( ϵ ) is a shorthand for the empty string. The regular expression (a|ϵ) defines {“a”, “”}.

*Note that there is no universally accepted standard for regular expression syntax. The syntax used above is adapted from* Appel & Ginsburg’s *Modern Compiler Implementation in C*.

Identifiers in Amethyst follow the regular expression ([\_a-zA-Z][\_a-zA-Z0-9]\*). This pattern, in plain English, reads: a letter or underscore followed by any number of letters, digits, and underscores. Thus, “instanceCount2” and “matrix\_size” both qualify as valid variable names, but “1stArray” does not. Much of Amethyst can defined more simply, like the addition operator, which is defined by the regular expression (+).

**Tokens & Lexing**

Recall from the first section that tokens are the ‘words’ of the language. They are broken into syntactic categories like keywords, operators, identifiers, etc. Lexing, the process of creating tokens from the source text, involves recognizing patterns in the text; hence, the usage of regular expression to describe the language’s lexical set (vocabulary).

In the Amethyst lexer, each category of token is recognized using a corresponding regular expression. When the lexer recognizes a pattern, it creates a token consisting of three elements: a type, indicating the the syntactic role of the token; a value, or the actual text of the source code that matches the regular expression; and the line number where the token was created.

To demonstrate the transformation of source text into tokens, consider a simple language with the following subset of Amethyst’s lexical rules:

1. Identifiers, with token type ID, are defined by ([\_a-zA-Z][\_a-zA-Z0-9]\*).
2. The plus operator, with token type PLUS, is defined by (+).
3. The less-than operator, with token type LT, is defined by (<).
4. The less-than-or-equal-to operator, with token type LTE, is defined by (<=).
5. All whitespace is ignored.

A diagram of a computer code

Description automatically generatedFigure 16 shows the tokens created from the (semantically nonsense) text, “id1 +\_all<<=a”:

*Figure 16. Characters of the source text mapped to tokens following the given lexical rules.*

A screenshot of a computer

Description automatically generatedTo recognize sequences of characters in the source text that conform to a regular expression in the language’s lexical set, the Amethyst lexer iterates over the text in a while loop. The body of the loop checks the current character against the initial character given by each regular expression. If the characters are the same, the lexer attempts to match the following characters in the source text to that regular expression. After a full regular expression is recognized, the lexer produces the appropriate token.Figure 17 provides the pseudocode for a lexer that handles the rules described above:

*Figure 17. Pseudocode for a simple lexer that recognizes a subset of Amethyst’s vocabulary.*

*It should be mentioned that discrete finite automata (DFAs) are another common*

*formalism used to describe lexers but they are not strictly necessary to describe Amethyst’s lexer.*

**Grammars** A *grammar* is a set of rules that describe how to form statements in a formal language. Amethyst is described by a particular kind of grammar called a *context-free grammar* (CFG). Other kinds of grammars exist, but they are not relevant to this project. In a CFG, a rule may be composed of *terminal* and *non-terminal* symbols. A terminal symbol is the basic unit of a grammar and might, for example, be a keyword, operator, or punctuation. They, in the case of Amethyst, correspond to individual tokens. A non-terminal symbol is a string of terminal symbols and other non-terminal symbols. The recursiveness of this definition is intentional. Consider the example of a simplified rule for while loops in Amethyst:

<while loop> ::= "while" <expr> "do" TERM <body> "end"

A screenshot of a computer keyboard

Description automatically generated The keywords while, do, and end, as well as TERM (a line terminator), are terminal symbols. The components <expr> and <body> are non-terminal symbols with their own grammatical rules. Note in Figure 18 how embedding these non-terminal symbols produces multiple valid while loops, despite their differing overall terminal (token) sequences:

*Figure 18. Two differing – yet valid – while loop token sequences.*

**Abstract Syntax Trees**

Once the tokens have been produced, the parser performs syntactic analysis, validating that a sequence of tokens conforms to the syntax specified by the rules of the grammar and, while doing so, constructs an abstract syntax tree (AST).

*A diagram of a diagram

Description automatically generated*Consider the expression 2 + 3 \* 4. If this expression is to behave with standard operator precedence, 3 \* 4 should evaluate to 12, which in turn is added to 2, resulting in 14. This hierarchy can be easily represented by a tree, with each operator as an internal node and its sub-expressions as children. In this example, the leaf nodes are the integers 2, 3, and 4.

*Figure 19. A simple AST.*

A diagram of a diagram

Description automatically generated This kind of hierarchical structure can be extended beyond simple expressions. Consider a while loop, which is composed of a condition expression and body statements. Each one of these components can be represented as a node with its own children. Figure 20 illustrates how a simple while loop may be represented as an AST.

*Figure 20. Conversion between while loop source code and its AST representation.*

**Recursive-Descent Parsing**

The Amethyst parser has parsing methods that – generally – each correspond to one grammatical rule. A parsing method walks over the tokens, beginning at the parser’s current position in the token sequence. If the grammar specifies a terminal symbol in the current position, the parsing method confirms that the current token is appropriate and either discards or stores its value. If the current token is not appropriate, a parser error is thrown, including the line number stored by the bad token, along with the expected token type. If the grammar specifies a non-terminal in the current position, the corresponding parsing method is invoked, and the invoked method performs the steps above. The AST node created and returned by this call is recorded and used at the end of the initial parsing method to construct its own AST node.

The entire program is parsed through a series of progressively deeper method calls, beginning and ending within the scope of the first one called, aptly named parseProgram(). This strategy, where parsing methods are called corresponding to the appearance of non-terminals in the grammar, is called *recursive-descent* parsing (Cooper and Torczon 109). For clarity, the usage of the term ‘recursive’ here does not mean that the same method is always calling itself directly.

A diagram of a diagram

Description automatically generated with medium confidence*Figure 21. A representation of recursive-descent parsing on a simple while loop. A token and the statement that consumes/discards it have the same color. The non-colored tokens are parsed by invoked methods.*

**The Visitor Design Pattern**

A diagram of a software program

Description automatically generated with medium confidence The Amethyst compiler uses the visitor design pattern to perform the rest of the stages of compilation. In broad terms, the visitor design pattern separates nodes from their operations and packages related operations into their own classes, called visitors (Gamma et al. 331). For Amethyst, this means that the AST nodes do not contain their own methods for semantic analysis and code generation. Instead, the classes SemanticAnalyzerVisitor and CodeGeneratorVisitor perform their respective operations with methods that take the AST nodes as parameters.

*Figure 22. A UML diagram showing the relationships between two of Amethyst’s concrete visitors and two concrete nodes. There are many, many more node types. Dependencies inherited from the base classes are omitted for clarity.*

The structure of the Visitor pattern in Amethyst is as follows: each AST node inherits from a base abstract node class which contains a virtual accept(:AbstractVisitor) method. Each visitor inherits from a base abstract visitor class that contains a visit(:ConcreteNode) method for every concrete node. Every concrete node class overrides the accept() method from the abstract node, and in its implementation, it calls the visitor’s visit() method, passing itself as a parameter. The accept() method takes a pointer to an abstract visitor as a parameter, so a v-table lookup determines the correct visit() method to dispatch. Inside the visit() method, the visitor performs the specific operation required for that particular node type. If the node has children that also need to be visited, the visitor calls each child node’s accept() method, passing a pointer to itself as a parameter.

The benefit of the visitor pattern is that it makes adding functionality very simple. Instead of changing each node when a new operation is needed, a new visitor can be created. This avoids the risk of side-effects when adding new functionality. In addition to visitors that perform semantic analysis and code generation, the Amethyst compiler has a visitor that outputs a representation of the AST in the DOT format (a graph description language), allowing the AST to be displayed visually. Other visitors, like a linter or code statistics collector could be added, changing neither the other visitors nor the nodes themselves.

The pattern does have costs, however. In gaining the ability to easily add functionality, adding new node types becomes more difficult. Because each visitor depends on each node, a new visit() method must be created in each visitor when a new node type is added. The visitor pattern also introduces cyclical dependencies, since the nodes depend on the abstract visitor, and the visitors depend on the nodes (visible in Figure 22). The pattern also introduces boilerplate code. Each node overrides the virtual accept() method, using the exact same lines of code. The cyclical dependency mentioned above requires a forward declaration of every node type before the definition of the abstract visitor. The pattern may also have an impact on performance, since a v-table lookup is required to determine the appropriate invocation every time a visit() method is called.

**Semantic Analysis**

Amethyst’s semantic analyzer traverses the AST and ensures at each node that the language’s features are used correctly. For each node type, ensuring correctness means something different. At a variable declaration node, for example, the semantic analyzer ensures both that the variable’s type is either a primitive or has been defined beforehand and that the variable’s name is unique in its scope. For a break statement node, the semantic analyzer ensures that the statement appears in a loop scope.

The semantic analyzer uses a variety of mechanisms to carry out its operations. One such mechanism is the *symbol table*, which ensures that variable declarations and references respect scoping rules. It is a stack of *Scope* objects that grows and shrinks as the semantic analyzer enters and exits scopes on the AST. These *Scope* objects are composed of a type (global scope, function scope, if block, etc.) and a map of variable names to data about the corresponding variables. To demonstrate usage of the scope stack, consider what must happen to ensure that a variable is used properly. When the semantic analyzer encounters a Variable node during its traversal, it checks each level of the scope stack to see if the variable has been declared beforehand. If not, an error is thrown. The symbol table is also used when verifying control flow statements. To ensure that break, continue, and redo only appear in loops, the semantic analyzer checks the type of each scope level. If none of the scope types are loops, an error is thrown.

Another mechanism is the expression type stack. When an expression of any kind (variables, addition, function calls) is visited by the semantic analyzer, the final step is to push the expression’s type onto the expression type stack. An expression with sub-expressions will use the sub-expressions’ types on the stack for its own type checking. Consider the Amethyst expression foo(1, 1.2) or false for a given function foo with return type bool, a first parameter int, and a second parameter float. Figure 23 demonstrates how the expression type A diagram of a diagram

Description automatically generatedstack grows and shrinks as the semantic analyzer visits each node in a post-order traversal.

*Figure 23. Snapshots of the expression type stack as the semantic analyzer visits AST nodes.*

The semantic analyzer occasionally alters the AST nodes during visitation. For example, when visiting nodes for function calls, method calls, new expressions, and uses of operators on user-defined types, the semantic analyzer will add a mangled version of the procedure’s name to the node. This facilitates code generation but introduces coupling between the semantic analyzer and code generator that is – strictly speaking – not necessary.

Given the 47 distinct AST node types, detailing every check the semantic analyzer performs would be impractical for this paper. Each source code file is documented with descriptions of checks relevant to each node type.

**Code Generation**

In the Amethyst compiler, the code generator converts the AST into textual LLVM IR.

As has been the case, there are too many node types to practically describe the translation of each. The following will be a broad overview of code generation for a few language features.

The generator handles expressions by assigning the values of sub-expressions to the IR’s virtual registers, then using those registers in the outer expression. The registers are stored on a stack, that grows and shrinks as an expression’s nodes are traversed. Operators (+, <<, %, or, etc.), are translated directly into LLVM operations (add, shl, frem, or, respectively), though the LLVM operation depends on the datatypes of the lefthand and righthand sub-expressions.

A call to a function generates an LLVM function call, whose syntax is very similar to function calls in other programming languages. Amethyst functions that return user-defined types – structs in LLVM IR – use sret parameters, meaning that the caller allocates memory for the returned struct and passes a pointer to the memory into the function. Similarly, for a method call, a usage of a defined operator, or a constructor invocation on a user-defined type, the generator emits an extra parameter or argument, passing a pointer to the struct that it operates on.

When accessing a member of a user-defined type (an element of a struct in LLVM IR), the generator emits a getelementptr (GEP) instruction as an assignment of the offset memory address to a new register. Arrays are similarly accessed. Assignment to struct or array elements involves storing values at these memory locations.

Control flow constructs, such as conditional blocks and while loops, use the branch (br) instruction and branch labels. For a while loop, for example, the code generator emits a labeled block that evaluates the condition expression. It then emits a branch instruction based on the register containing the boolean result of the condition expression. It emits a labeled block for the body of the loop, the final statement of which is a branch back to the condition label. Finally, the code continues in a block labeled as an exit. Break, continue, and redo statements in loops map to branch A diagram of a diagram

Description automatically generated with medium confidenceinstructions, targeting their respective labels.

*Figure 24. Translation of an AST to LLVM IR produced by the Amethyst compiler. Each node is the same color as the statement(s) produced when the code generator visits it.*

**Bibliography**

Appel, Andrew W. and Ginsburg. Maia. *Modern Compiler Implementation in C*. Princeton University Press, 1998.

Cooper, Keith D. and Torczon, Linda. *Engineering a Compiler 2nd Edition*. Morgan Kaufmann Publishers, 2012.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Lattner, Chris. “LLVM,” in *The Architecture of Open Source Applications (Volume 1).* https://aosabook.org/en/v1/llvm.html.

Lattner, Chris and Adve, Vikram. "LLVM: a compilation framework for lifelong program analysis & transformation." *International Symposium on Code Generation and Optimization*, 2004. CGO 2004., San Jose, CA, USA, 2004, pp. 75-86, doi: 10.1109/CGO.2004.1281665.

Young, Miguel. "A Gentle Introduction to LLVM IR.” 8/1/2023. https://mcyoung.xyz/2023/08/01/llvm-ir/.

**Appendix**

I. Implementation of a Dynamic Array of Integers

type DynamicIntArray

@data:int\*

@length:int

@capacity:int

new()

@capacity = 8

@data = heap [@capacity] int

@length = 0

return

end

def @append(value:int)

# allocate new space when reaching end of capacity

if @length + 1 == @capacity then

# allocate new capacity

@capacity = @capacity \* 2

tempData:int\* = @data

@data = heap [@capacity] int

# copy old into new

i:int = 0

while i < @length do

@data[i] = tempData[i]

i = i + 1

end

unheap tempData

end

@data[@length] = value

@length = @length + 1

return

end

# destructor

def @destroy()

unheap @data

return

end

end

II. Implementation of a Matrix of Integers

# definition of matrix type

type Matrix

# member declarations

@data:int\*\*

@size:int

# constructor, allocates size x size int matrix on heap

new(size:int)

@size = size

@data = @allocateNxN(@size)

return

end

# method, get a deep copy of this matrix

def @deepCopy():Matrix

result:Matrix = new Matrix(@size)

i:int = 0

while i != @size do

j:int = 0

while j != @size do

result.data[i][j] = @data[i][j]

j = j + 1

end

i = i + 1

end

return result

end

# method to allocate an nxn on heap, returns a ptr to the matrix int\*\*

def @allocateNxN(n:int):int\*\*

rows:int\*\* = heap [n] int\*

i:int = 0

while i != n do

rows[i] = heap [n] int

i = i + 1

end

return rows

end

# continued on next page...

# operator to multiply matrices (standard matrix multiplication)

# returns a new heap allocated matrix

op\*(rhs:Matrix):Matrix

# if sizes don't match, do no allocation & return a matrix of size 0

if @size != rhs.size then

return new Matrix(0)

end

# sizes match, perform matrix multiplication

result:Matrix = new Matrix(@size)

i:int = 0

while i != @size do

j:int = 0

while j != @size do

k:int = 0

while k != @size do

result.data[i][j] = result.data[i][j] + @data[i][k] \* rhs.data[k][j]

k = k + 1

end

j = j + 1

end

i = i + 1

end

return result

end

end