**Amethyst**

A Custom Language & Compiler to LLVM IR

David Britton

Honors Project  
Department of Computer Science  
Fall 2024

**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 & 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 & 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, equally referred to as lexing or tokenization. In this stage, the source text is broken into a sequence of *tokens*, which can be though 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 a the wrong data type (Appel & 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 utility lies in the common back-end interface provided by its intermediate representation, LLVM IR. 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 & 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 a static single assignment (SSA) model 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. URL?

**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.

Variables 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 with if, elsif, and else blocks. An if block begins with keyword if, followed by a condition and keyword then. The body of each block begins on a new line and executes if the condition is true. Optional elsif blocks, each with their own condition and keyword then, follow the if body to check additional conditions if the previous conditions are false. If none of the conditions are true, an optional else block can be used to execute default instructions. All conditional blocks are terminated by 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.*

Where type names appear, they may have asterisks ( \* ) appended to them to indicate that they are pointer types. Multiple asterisks mean that the type is a pointer to a pointer. For example, an int\*\* is a pointer to a pointer to an int. There is no distinction between pointers and arrays. Because of this, an Amethyst user should always track of the length of an allocation referenced by a pointer. All dereferencing is performed with the subscript operator, indicated by square brackets ( [] ). 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 SOMEWHERE.

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 the following 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 other programming languages. *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.*

The syntax of method definitions is nearly identical to that of function definitions, the key difference being that the name is preceded by the member sigil. The member sigil is used for method calls within the type definition. For calls outside of the type definition, the dot operator is used.

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 that are 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.

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.

Type is lhs, parameter is rhs

Overloads, but can’t redefine an operator with the same parameter type.

**Assignment of User-Defined Types**

Struct definitions and assignments are shallow copies (literally invocations of the LLVM intrinsic memcpy)

**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). 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 may be concatenated to other expressions. 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 Y shows the tokens created from the (semantically nonsense) text, “id1 +\_all<<=a”:

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

To 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.

A screenshot of a computer

Description automatically generated Figure X provides the pseudocode for a lexer that handles the rules described above:

*Figure X. 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 X* how embedding these non-terminal symbols produces multiple valid while loops, despite their differing overall terminal (token) sequences:

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

**Abstract Syntax Trees & Operator Precedence**

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).

As Consider the expression 2 + 3 \* 4. If this expression is to behave with operator precedence expected

<add\_expr> ::= <mult\_expr>

| <add\_expr> + <mult\_expr>

<mult\_expr> ::= integer

| <mult\_expr> \* integer

*A diagram of a diagram

Description automatically generated*

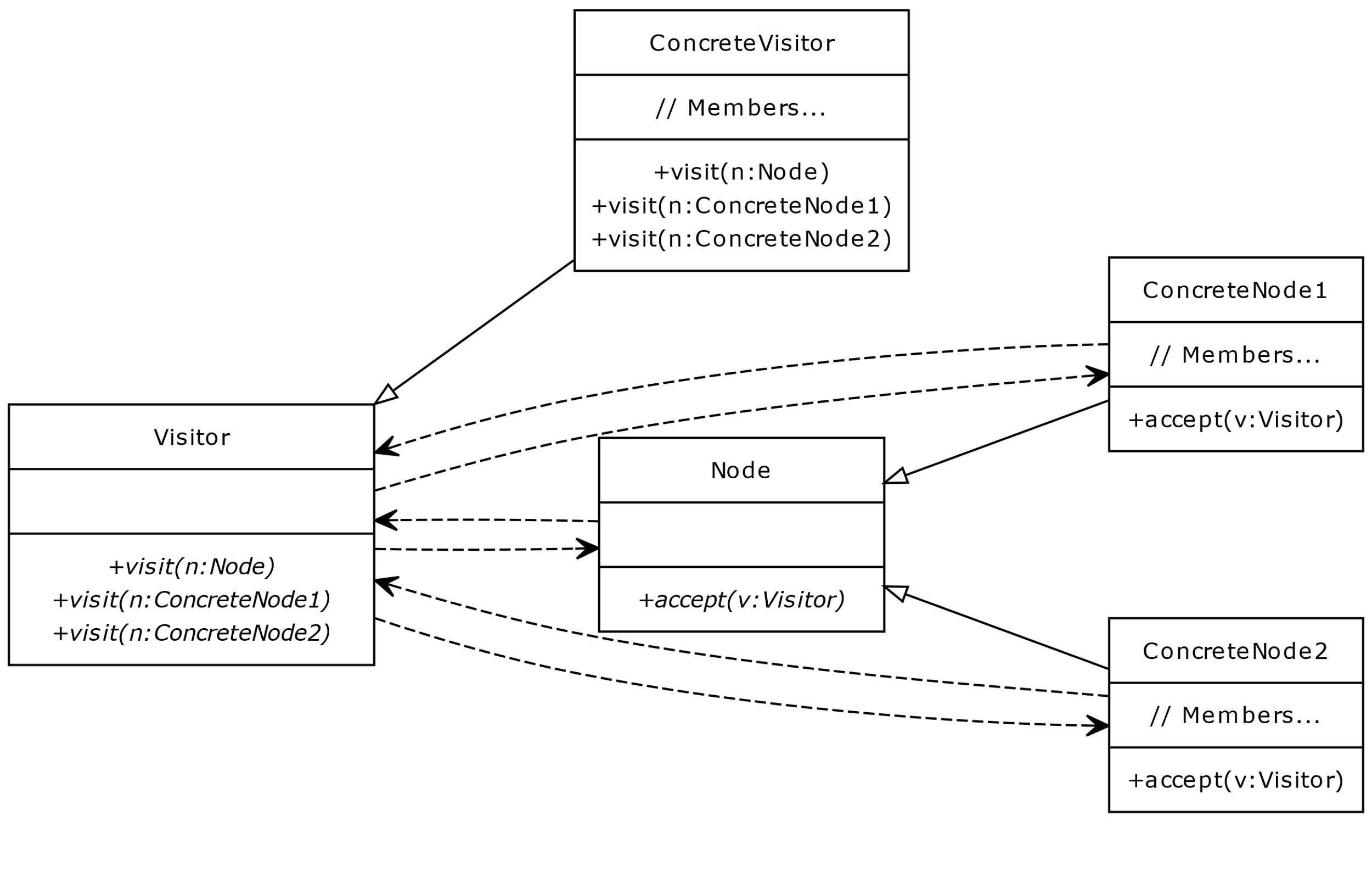
**Recursive-Descent Parsing**

In the Amethyst parser, each parsing method generally corresponds to one grammatical rule.  A parsing method walks over the tokens, beginning at the point in the token sequence where it is invoked.  If the grammar specifies a terminal symbol in the current position, the parsing method confirms the existence of the token and either discards or stores its value.  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 main parsing method to construct its own AST node.

A diagram of a diagram

Description automatically generated with medium confidenceThe 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 (Cite). For clarity, the usage of the term ‘recursive’ here does not mean that the same method is always calling itself directly, but rather that parsing involves a sequence of method calls that repeatedly invoke each other.

*Figure X.*

**d. Visitor Pattern***Figure X. UML diagram for a general case of the Visitor design pattern. Note that dependency arrows to and from ConcreteVisitor have been omitted for clarity. Made with yuml.me.*

**Semantic Analysis**

Symbol tables to help with type checking, name resolution, etc.

Name mangling

Amethyst’s grammar makes determining what’s a type name, a variable name, a function name, etc unambiguous, and that is reflected in the AST. There is a “classic” symbol table maintained for variables in function scope, but otherwise, we can just maintain maps for functions and types. Types contain more maps themselves, for member, methods, ops, constrctrs.

**f. Code generation**

**Translating High-Level concepts into LLVM IR**

LHS and RHS evalutated,

FIGURE: AMETHYST TO LLVM IR colorcoded