

Developing an Imperative Programming Language Based on Pattern Matching and Value Semantics

by

Zhandarbek Saginov

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

NAZARBAYEV UNIVERSITY

May 2026

© Nazarbayev University 2026. All rights reserved.

Author
Department of Computer Science
May 2026

Certified by
Hans (Jean) de Nivelle
Associate Professor
Thesis Supervisor

Accepted by
Adnan Yazici
Chair of Computer Science, School of Engineering and Digital Sciences

Developing an Imperative Programming Language Based on Pattern Matching and Value Semantics

by

Zhandarbek Saginov

Submitted to the Department of Computer Science
on May 2026, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Continuous studies in compiler constructions and the development of automation tools for compiler modules enabled the integration of more abstract and high-level concepts into modern programming languages. The efficient paradigms and improvements for writing clean and extensible code not only simplified the development of the compilers but also the way programming languages support new statement constructs. Therefore, the focus has shifted from developing an efficient general-purpose compiler to small compilers with more specialized rules and areas, promoting safety, readability, expressiveness, and reasoning. Existing works on mutable value semantics (MVS) have mainly focused on theoretical foundations and practical applicability analysis. While, pattern matching, originally introduced in functional languages such as Haskell and OCaml, is now increasingly integrated into imperative programming languages. Valeriya (derived from ‘Value Semantics’) is a new statically typed imperative language, which using the TreeGen parse-tree automation tool, and current studies in MVS, supports local reasoning, semantic efficiency with promoted safety, and pattern matching on an abstract data type construct, called property. The main contribution of this thesis is the design and implementation of the Valeriya programming language, including its compiler infrastructure, the introduction of the property type construct for algebraic data types, and the integration of mutable value semantics with structural pattern matching in an imperative programming language setting. The language aims to eliminate uncontrolled mutation and simplify coding logic, calculus, rule-based systems, scripting, compiler-interpreter development using property type constructs and pattern matching on them. Using modern compiler tools and practices, Valeriya facilitates the use of strictly mutable value semantics on complex algebraic data types, providing foundational work for promoting MVS in more complicated programming language paradigms.

Thesis Supervisor: Hans (Jean) de Nivelle
Title: Associate Professor

Contents

1	Introduction	13
1.1	Motivation	14
1.2	Language Objectives	15
1.3	Thesis Contents	15
2	Related work	17
2.1	General Compiler Principles	17
2.1.1	Compilation Steps	19
2.1.2	Compile-time and Error Handling	24
2.2	Programming Languages	25
2.2.1	Primitive Data Types and Algebraic Data Types	25
2.2.2	Variables	26
2.2.3	Reference Semantics	26
2.2.4	Value Semantics	27
2.2.5	Mutable Value Semantics	28
2.2.6	Pattern Matching	32
2.3	TreeGen and Tree Virtual Machine	34
3	Valeriya Programming Language Design	39
3.1	Primitive Data Types	39
3.2	Algebraic Data Types	41
3.2.1	Struct Types	41
3.2.2	Enum Types	41

3.2.3	Property Type Definitions	41
3.2.4	Recursive Declaration	42
3.2.5	Disallowed field calls on field calls	43
3.3	Handling Arrays	44
3.4	Deterministic Behavior Using Value Semantics	45
3.4.1	Mutable Value Semantics	45
3.4.2	Copy-on-Write	46
3.4.3	Inout Parameters	47
3.5	Matching and Type Deduction	48
3.5.1	Type Deduction of Options	49
3.5.2	Property Type — Use Cases	50
4	Implementation	55
4.1	Compiler Implementation Details	55
4.2	Lexer	56
4.3	Parser	58
4.3.1	Algorithms For Parsing	60
4.3.2	Operator Precedence Parsing	61
4.4	Semantic Analysis	62
4.5	Rules for Mutable Value Semantics	65
4.6	Code Generator	68
5	Conclusion	71
5.1	Overview of Valeriya	71
5.2	Future Works	72
5.2.1	Generic ADTs	72
5.2.2	Multi-dimensional Arrays	72
5.2.3	Multithreading and Concurrency	72
5.2.4	Matching Limitations, Structure Bindings	73
5.3	Conclusion	73

Appendix	77
A.1 Language Grammar (Extended Backus-Naur Form Definition)	77
A.2 Token Labels	79
A.3 (.rec) File Required by TreeGen For AST Definitions	81

List of Figures

2-1	General Compiler Architecture	24
2-2	a) reference semantics: variables share the same memory; b) value semantics: variables store different data;	27
4-1	Valeriya Compiler's Design	55
4-2	Abstract Syntax Tree (AST) of Statements of Valeriya Programming Language	59
4-3	Example: generated TreeGen parse tree in Valeriya.	63

List of Tables

4.1	Operator precedence levels used by the Valeriya parser. Higher value means higher precedence.	62
-----	---	----

Chapter 1

Introduction

Contemporary compiler construction is a heavily researched topic with many automation tools, abstraction codes, and theoretically verified approaches [1][3][9][12][18][20]. The current decision-making process for programmers creating a programming language is not about thinking about low-level instructions, but thinking about the idea of the level of abstraction, flow control, semantics, and providing possibilities in building one effectively. The compiler principles suggest two important stages, analysis and synthesis [18]. During the analysis phase, we want to formally permit the language to have a defined set of ideas and practicabilities, while synthesis should take care of translation to the target language [10].

This work presents a new imperative programming language based on pattern matching and value semantics called Valeriya. By synthesizing the theoretical works in compiler construction [20][12][14], and the use of a parse-tree generator tool, Tree-Gen [4][3], Valeriya effectively supports the concept of Mutable Value Semantics (MVS) [14][15], and a new abstraction type construct called *property*. Valeriya exemplifies how the use of modern tools and approaches can help to develop a compiler following Niklaus Wirth’s guiding principles in Chapter 2.

1.1 Motivation

The first motivation arises from the visible vulnerabilities in Python and C^{++} programming languages. Both languages support reference semantics, which violates equational reasoning and promotes unsafe code. Undoubtedly, reference semantics is the core of Python and Java, while C^{++} supports both value and reference semantics, often leaving the choice to the programmer; providing explicit control of the program's states by the programmers. However, in Valeriya, we will try to promote the concept of mutable value semantics (MVS), a semantics of a language that implicitly includes the power of reference semantics while preserving user-view with pure value semantics [14][15]. If MVS is proven to be efficient for the same reference semantics code, it can become a basis for future programming languages that try to promote safe and expressive code.

The second motivation is introducing an algebraic data type (ADT) powered with safe pattern-matching logic, called *property*. In functional programming, the term ADT commonly refers to recursive sum and product types [6][16]. In this thesis, the term ADT is used in a broader sense to denote user-defined composite data types, including structs, enums, and property types. The term property was chosen to distinguish Valeriya's tagged union construct from conventional ADTs and to emphasize that each option represents a distinct structural property or state of a value that can be safely decomposed through pattern matching. This idea was inspired by the latest C^{++} 26 proposal on pattern matching in a language standard [13]. The existing C^{++} `std::visit` function on `std::variant` types is effectively what Valeriya is trying to implement, but in a more readable pattern. There are a huge number of problems with the `std::visit` function: verbosity, heavy template logic, and strict return type homogeneity. `std::visit` suffers from a significant combinatorial explosion of dispatch tables when multiple variants are visited simultaneously, forcing the compiler to instantiate exponentially increasing control paths [5]. Furthermore, the approach lacks native integration with the language's control flow; it is impossible to execute a `break`, `continue`, or an early return from the surrounding function scope within a visitor's

lambda. Valeriya, on the other hand, tries to create a similar ADT to `std::variant` and apply safe and efficient matching than `std::visit`.

1.2 Language Objectives

This work introduces a new imperative statically-typed language based on property type constructs and matching called Valeriya. Valeriya implements mutable value semantics (MVS) to control mutation of its variables. The main objectives for the Valeriya programming language:

- **Support Formal Reasoning.** Mutable value semantics (MVS) in Valeriya effectively promotes equational reasoning and logical soundness of the programs by treating variables as distinct values, disallowing untracked value mutation and state changes [2][15]. Equational reasoning refers to the ability to replace expressions with equivalent values without changing program behavior.
- **Introduce a way of defining Algebraic Data Type.** Property is a new high-level recursive abstraction sum type construct in the Valeriya that simplifies expressing logic, calculus, rule-based systems, and scripting commands.
- **Promote pattern matching.** Introduce structural pattern matching, a decomposition tool in Valeriya that helps to match property type constructs' states and options to manipulate the fields of the property types safely.

1.3 Thesis Contents

The thesis structure is given as follows:

- **Chapter 1.** Introduction to the language, motivations, objectives, and use areas.
- **Chapter 2.** The existing related works in the field of compiler constructions and programming languages. How other programming languages implement

pattern matching and Dave Abraham’s view on mutable value semantics (MVS). How the TreeGen tool can be used to accelerate the implementation of the compiler.

- **Chapter 3.** High level of view on Valeriya’s possibilities. Theoretical definitions of mutable value semantics (MVS) and practical cases of pattern matching in the language.
- **Chapter 4.** Compiler implementation details, algorithms, and translation code.
- **Chapter 5.** Conclusion and future works.
- **Appendix.** Extended Backus-Naur Form (EBNF), Lexer tokens, TreeGen definition file.

Chapter 2

Related work

2.1 General Compiler Principles

Compiler construction has been a prominent field in computer science for half a century, and thus it is one of the most extensively researched topics. The first ever compiler for a programming language was developed in 1957 by the IBM team, and it was called FORTRAN. The ever-increasing performance and more human-readable coding are all intertwined with the continuous development in the compiler and interpreter assembly. A very prominent figure in compiler design is Niklaus Wirth, whose proposed compiler-building principles created the foundation for modular and structural design; his layered functional design provided many opportunities to make a compiler an extensible system [12]. Beginning from an efficient and monolithic GCC compiler founded in 1980 by Richard Stallman, up to the point where compilers are treated as a backend for any form of front-end languages, according to Chris Lattner's LLVM infrastructure (2002) [9].

In line with [18], a good programming language and its compiler must follow the criteria: well-defined syntactic and semantic description, determinism of program structure, fast translation, reliability, machine independence, generality, extensibility, provability, and consistency with commonly used notations. Also, we will constantly refer to the guiding principles for programming language and compiler design by Wirth:

- Using straightforward techniques.
- Generating good code, not optimizing poor code.
- Language should not encourage bad programming styles by repairing them.
- Generating predictable code.
- The simplified language is better than the complicated compiler.
- Striving for short compilation time

We will revisit the derived design principles above when building our own compiler. The principles and practices of design suggest the following goals in a compiler.

- **Develop Compilers as Small as Possible:** A compiler should remain as compact and efficient as possible without compromising functionality. Minimalism in design reduces complexity, improves performance, and makes maintenance more manageable.
- **Ensure Descriptive and Recoverable Error Handling:** Error messages should be informative, precise, and context-aware, enabling programmers to identify and correct issues efficiently. The compiler should also be designed to recover gracefully from errors rather than terminating abruptly.
- **Maintain a Clear and Direct Translation Path:** The translation process from source code to target machine instructions must follow a transparent and systematic sequence. Each phase should have clearly defined roles and responsibilities.
- **Prioritize Readability and Modularity of Implementation:** The internal structure of the compiler should be organized into coherent modules that separate concerns logically.

2.1.1 Compilation Steps

Programming languages are tools used to construct formal descriptions of finite computations [20]. A programming language provides essentially three components for describing such computations: data types, objects, and values with operations defined upon them; rules that establish chronological relationships among specified operations; and rules fixing the static structure of a program. These components together constitute the level of abstraction on which we can formulate algorithms in the language. Developing a programming language is tied to constructing a compiler or interpreter that processes a stream of characters representing the language components. Compilers or interpreters preprocess, analyze, catch errors, optionally optimize, and generate a binary to be executed on the computer hardware.

A compiler is a program that sequentially passes the source code through defined compilation steps. Usually, it is denoted as a sequence of transformations from one representation to another. The formula represents a sequence of transformations from source code to target code through intermediate representations:

$$(SL, L_1), (L_1, L_2), \dots, (L_n, TL), \quad (2.1)$$

where SL is source language, and TL is target language; each L_i is called an intermediate representation (IR) [20]. Some compilers employ multi-stage passes for a more optimized output executable.

Intermediate Representation

Internal representation is given as structures inside the compiler that are passed through the compiler modules to transfer information about tokens, in-language statements, type constructs, syntax tables, and others. Intermediate representation (IR) is an independent source that is often used for optimization and intermediate translation. For example, LLVM generates LLVM IR, TypeScript is translated to JavaScript source, and the GCC compiler produces Gimple IR.

Lexical Analysis.

The tool performing the lexical analysis is often called a lexer, tokenizer, or scanner; we will be referring to as lexer from now on. Lexer is the elemental step for preparing the list of tokens and eliminating syntax errors. A token is usually a pair or a tuple (λ, a) , where λ is the label and a is its attribute. Some tokenizers include additional information, such as position, source file, or even their dependence on other tokens for error localization and handling [11]. Token's attribute can be a keyword; in that case, we only provide a λ as a token.

Lexers scan the entire source code once and pass the stream of tokens to the subsequent layers. The input to the lexical analysis is a stream of characters. In addition to lexical analysis's main task of transforming the stream of characters to the stream of tokens, it can also focus on stripping out the whitespaces and comments (i.e., preprocessing), line and column counting (i.e., information on compilation), and error handling (i.e., catching lexical exceptions) [18]. For example, the lexer of the C compiler's preprocessor also handles macro definitions as one of its functionalities.

Lexers must discern tokens by their labels, based on the regular expressions. The finite automaton generates the token list based on the regular expressions defined by the language definitions. Here are some fundamental regular expressions commonly used in programming languages:

```
digit  → 0|1|2|...|9
letter → a|b|...|z|A|B|...|Z
symbol → +|-|^|!|&|_|...

identifier → [letter|_]+ [letter|digit|_]*
integer    → 0 | [digit^0] [digit]*
com_line   → [//]+ [digit|letter|symbol]*
```

The sequence of regular expressions that are prioritized before others is called a dictionary. Lexers must define a set of reserved words or keywords that come before the regular expression of identifiers. Identifiers and keywords must not overlap, so

that we can then correctly build the statements of the language. The reserved words are passed as tokens without attributes but with corresponding labels.

In compiler constructions, we create a nondeterministic finite automaton (NFA) with a set of transitions from one state. Unlike deterministic finite automata (DFAs), NFAs introduce epsilon transitions and can have multiple states followed after one state and the next token. NFA is a tuple of [1][20]:

$$M = (Q, \Sigma, \delta, S, F) \quad (2.2)$$

where Q is a finite set of states, Σ is the input alphabet, $S \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, δ is the translation function from one state to a set of states.

$$\delta : Q \times (\Sigma \cup \lambda) \rightarrow 2^Q \quad (2.3)$$

where 2^Q is the set of all subsets of Q .

NFAs are formal definitions of regular expressions' translation paths. Compilers do not work with NFAs directly, since NFAs require simultaneous exploration of multiple execution paths, which is inefficient and impractical for real-time lexical analysis. To construct an efficient lexical analysis, the NFA is made deterministic by applying a subset construction that converts sets of NFA states into states of an equivalent DFA. DFA's translation function is deterministic and has only one controlled translation path:

$$\delta : (\Sigma \cup \lambda) \rightarrow Q \quad (2.4)$$

Since the final DFA usually has redundant translation steps, compilers employ algorithms to minimize the DFA. The final optimized DFA is what compilers use to generate a stream of ready-to-parse tokens:

$$T = \{t = (a, \lambda, I) | a \in A_\lambda, \lambda \in \Lambda, I - \text{location information}\} \quad (2.5)$$

where A_λ is a set of attributes with valid token label, Λ is a set of all token labels.

Syntactic Analysis.

This stage is accountable for linking tokens to the structural grammar rules of the language. This stage can combine the parser and semantic analyzer as a single unit. A parser builds a structural representation (often a parse tree or syntax tree) of the source code by using techniques such as top-down parsing or bottom-up parsing. Abstract Syntax Trees (ASTs) are commonly used as a more compact and semantically meaningful representation than raw parse trees, capturing the hierarchical structure of the program without unnecessary syntactic details. Similarly to the lexical analysis, there are two distinct goals of syntax analysis: check if the input program is syntactically correct, and produce or trace the structured AST [16]. Most of the errors discovered in the current two analysis steps are known as compile-time errors [18].

Once a valid syntax tree is generated, it becomes the foundation for subsequent phases, including semantic analysis, intermediate code generation, and optimization.

Parsers are categorized according to the direction in which they build parse trees [16][1]:

- **Top-Down Parser.** They build the tree in preorder: from the root to the leaves of the tree. The input tokens are scanned from left to right, and the parser finds the next sentential form in the leftmost derivation [16]. Most parsers consume a single token of lookahead to make a prediction of the language statements; $LL(1)$ is a common type of top-down parser. If the statement fails, the top-down parser rewinds its tokens to search for the next similar statements.
- **Bottom-Up Parser.** They build the tree from the leaves to the root. Unlike the top-down parser, the bottom-up parser produces the rightmost derivation in reverse [16]. Bottom-up parsers usually wait for terminal symbols, such as the semicolon (;), to reduce the visited tokens to the structural rules of the language. $LR(1)$ is a common type of bottom-up parser.

Code Generator.

The task of the object generator is to create a relocatable machine code using the intermediate representation produced in the previous steps. Compilers can also generate intermediate object files that can be aggregated to the final executable or binary. Ultimately, the code generation stage bridges the abstract semantics of the language with the concrete operational semantics of the target architecture.

Any compilation can be broken down into two major tasks [18]: analysis discovers the structure and primitives of the source program, determining its meaning; synthesis creates a target program equivalent to the source program. In general, analysis and synthesis are two distinct stages accomplishing their own defined tasks: synthesis modules do not spend time verifying the input IR — they assume that analysis is already finished, and operate as the IR is clean and ready to be used for binary generation. For instance Low Level Virtual Machine (LLVM) separates frontend analysis stages, such as parsing and semantic analysis, from middle-end and backend optimization/transformation passes. In this context, LLVM analysis passes primarily gather information used by later optimization and synthesis stages [10][9]. Valeriya follows a similar separation, where lexical analysis, parsing, and semantic verification are completed before translation into C intermediate representation.

Although some simple compilers are sufficient with only one stage [18], the number of stages can vary depending on the language complexity, design decisions, optimization requirements, and language specifications. It is feasible to have multiple stages combined as a one pass, so that each small pass is done interlacingly as one module. Nevertheless, LLVM, a high-level compiler infrastructure used for powering and compiling any language, provides an LLVM front-end that uses multi-stage optimization and focuses on modular design and reusability [9]. LLVM has many stages, and it is proven to be an elegant modern implementation of compilers.

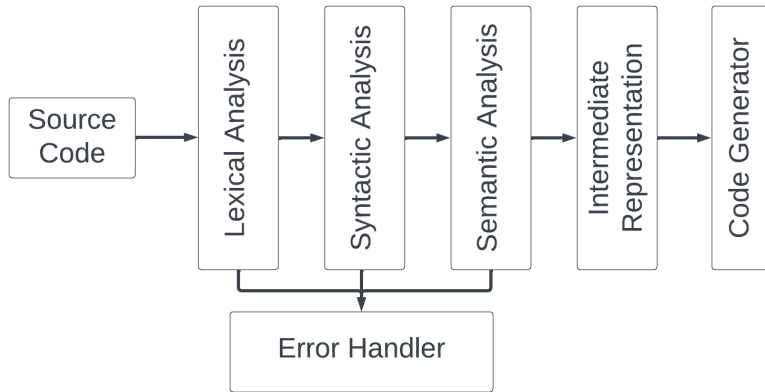


Figure 2-1: General Compiler Architecture

2.1.2 Compile-time and Error Handling

The time the source code is translated to the target code is called compilation time. During compilation, we can encounter numerous errors declared as compilation errors due to their occurrence in the compilation phase. Possible error causes are: internal compiler error, its environment, hardware and operating system issues, programming design errors, missing dependency errors, etc. [20]. According to [18], every stage should be linked to the block of the error handler. The compilation phase is responsible for gathering all possible errors and giving definitive descriptions of what went wrong. The target language is not generated if a compilation error happens.

There are three types of errors that occur during the analysis phases: invalid compiler input or command-line usage, syntax errors, and semantic errors. Invalid compiler input or command-line usage is about providing bad arguments to the compiler options, such as passing a source file with an unexpected extension. Syntactic errors are characterized as errors in the structure of the language, such as missing identifiers, operators, or keywords. Semantic errors are determined by errors in agreement, such as type incompatibility, or usage of undeclared variables [18].

The internal representation and code generation must not throw any errors. By the time the IR reaches the synthesis phases, it has to be verified to be free of errors in the analysis phases. However, after the final executable is ready, we can still encounter runtime errors. Although the runtime error is usually a flaw in the logic of a program,

and not a compiler fault, compiler designers must carefully test if the compiler itself adequately translates the source to the final IR. A runtime error is an error occurring during the program's execution. Runtime errors arise because of unhandled thrown exceptions, or direct or implicit terminate function calls [5].

2.2 Programming Languages

2.2.1 Primitive Data Types and Algebraic Data Types

Languages must define primitive data types — structures in memory with specific representation, operations defined on them, and handling according to the programs' needs. Also, programming languages usually define sequences of types called arrays, which are contiguous blocks of types that support indexing and in-place rewriting. Primitive data types, though language-dependent, usually include **integer**, **double**, **character**, **boolean**, **unsigned integer**, **unsigned double**, and **their arrays**.

Most programming languages, such as C^{++} , allow defining custom data types and structures using classes, structs, unions, enumerated types, and tuples [5]. Custom types use a combination of primitive or other custom types to define the behavior of some complex structures. For example, a well-known `std::string` is a class that stores an array of chars within itself, providing safe interface functions and overloaded operators to manipulate the raw char pointer. Those custom types are also known as algebraic data types (ADTs). There are two primary types of ADTs: sum types and product types [7] [8].

- A **sum type** is a type that has an "OR" relationship between its fields: only one field is active and valid at a time. Examples: enumerated types, union types, variants.
- A **product type** is a type that has an "AND" relationship between its fields: every field is accessible and can be changed independently. Examples: struct types, classes, tuples, records.

- A **combined type** is either a sum type or a product type that stores other ADTs as its fields. They can also store the instances of themselves, making it a recursive dependence type. Examples: **property type definitions** (this work).
- A **recursive ADT** is either a sum type or a product type that can store an instance of its own type as a field. Recursive ADTs are never implemented in pure value semantics, since their size cannot be statically determined due to recursive structure. Therefore, they use pointers to heap memory to dynamically allocate as much memory as needed in the runtime.

2.2.2 Variables

Programming languages are based on commands that programmers give and change the states and variables based on those commands. A variable is an instance of the primitive or algebraic data types. Programming languages use those components to simplify the conceptual map in building algorithms.

Imperative programming languages must define a kind of semantics to actually change the underlying data in their variables. For example, C^{++} supports value, reference, pointer, and move semantics; all of which are ways to transfer data between variables [5]. This work relies on value semantics. Each variable holds its own copy of the memory, which opposes reference semantics that allow sharing data stored in the same address in memory. As an example, Python has implicit reference semantics, which means that the assignment of a variable to another implicitly makes the assigned variable refer to the same memory location; thus, changing one of them leads to a change for both.

2.2.3 Reference Semantics

The reference semantics for variables means that different variables may refer to the same object with the advantage of avoiding copies of the data. However, this introduces unexpected dependencies between variables, making it hard to reason about

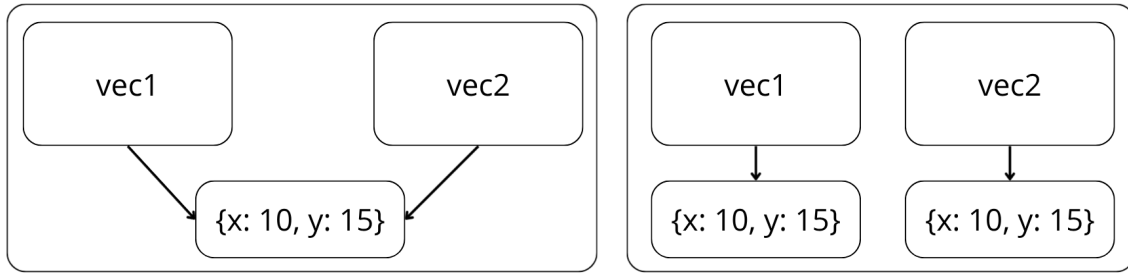


Figure 2-2: a) reference semantics: variables share the same memory; b) value semantics: variables store different data;

algorithms. When different variables access the same memory, they are technically different names to the shared resource, see Fig 2-2 a). The primary advantage of reference semantics over value semantics is performance. Because variables do not deep copy large ADTs or arrays, it is extremely efficient to use reference semantics for read-only access or in-place controlled mutating [14]. However, it becomes difficult to formally reason about the program that may change the memory invisible to other aliased variables. Local reasoning fails with reference semantics because we have to consider cells referring to the same memory. Now, we are required to reason about the system as a whole, not as independent small parts.

2.2.4 Value Semantics

Value semantics enables each variable to store its unique copy of some data, see Fig. 2-2 b). So, changes or mutations are always clear and predictable. All types have value semantics enabled [14]. Value semantics is easy to reason about without thinking beyond the concept that variables are containers for data. The compiler's task is to enable optimization using reference and move semantics without exposing the nature of value semantics.

Value semantics is the default setting for primitive data types. Copying primitive data types is simple and does not create overhead. For arrays, strings, and complex ADTs, it is better to use reference semantics where appropriate, minimizing the deep copy of a large number of fields.

2.2.5 Mutable Value Semantics

We do not want to allow reference semantics ambiguity in our language, but we still want to use reference semantics to increase the efficiency of the code. As we discussed above, reference semantics is great for read-only access or controlled mutation, only if the compiler can prove their uniqueness and controlled access [14].

```
int sum_arr(const int* arr, int size)
{
    int sum = 0;
    for (int i = 0; i < size; i++)
    {
        sum += (arr[i]);
    }
    return sum;
}
```

In the C-code above, variable *arr* is a const pointer, meaning it has read-only access. We do not copy a large block of memory from *arr*, but use it only to read variables. The behavior is controlled, and it is easy to formally verify that the post condition is that the *arr* is unchanged. This is our first desired use case of the reference semantics.

```
void rand_init_arr(int* arr, int size)
{
    for (int i = 0; i < size; i++)
    {
        arr[i] = rand() % 100;
    }
}
```

In the C-code above, variable *arr* is a pointer that we expect to modify. The function declaration tells us that our array is modified inside the function. We will

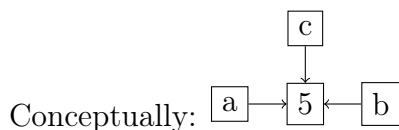
call such variables **inout** variables, also called output parameters. This is our second desired use case of reference semantics. We can prove that the resource is not shared anywhere except the function, so we can formally reason about the program. Moreover, we gain a performance boost since we do not copy the array. Inout variables are viewed as copy-on-call and copy-on-return, though internally they are treated as references [14]. Because references never escape into user-visible constructs, all values remain independent. Therefore, reference semantics should be treated as a tool for optimization, while preserving the value semantics of local and predictable states.

The central semantics of this work is mutable value semantics (MVS). MVS is a variation of value semantics that internally utilizes reference semantics for optimization. According to Dave Abrahams et al., reference semantics promotes unsafe code in trying to optimize it. We will constantly refer to Dave Abrahams' work related to mutable value semantics [14][15]. Value semantics should be a basis for the language. For users, everything is conceptually viewed as values; from the compiler's side, it should be handling these three scenarios for mutability and efficiency. The following examples are given in the Valeriya programming language (this work, see the next Chapter) and with primitive integers to visualize the process of each approach.

Read-only access and copy-on-write

Variables can refer to the same memory as long as they do not change the memory. References are shared until a mutation occurs, at which point a real copy is made for the variable, alternating the data.

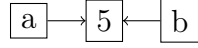
```
int a = 5;
int b = a;
int c = b;
```

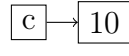


When `b` and `c` are assigned to `a`, they are allowed to refer to the value of `a`. But as soon as one tries to change it, it will create its unique copy without allowing mutation.

We achieve this using reference counting on our lvalues. Lvalue is a value that has non-temporary address.

```
c = a + b;
```



Conceptually: 

Inout parameters

The output parameter is modeled in a way that it is viewed as copy-on-call (argument passing) and copy-on-return (value returning), while internally it uses the reference to the data to modify it.

Consider the example:

```

fn offset(inout int x, int delta) -> void {
    x = x + delta;
}

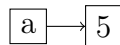
fn offset2(inout int x, int delta) -> void {
    offset(x, delta);
    offset(x, delta);
}

int a = 5;
offset2(a, a);
  
```

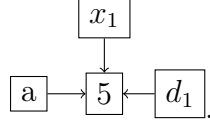
Based on this example, we show the process of inout handling in MVS language.

Let $x_1, x_2, x_3, d_1, d_2, d_3$ be a local variables of each call to the *offset* and *offset2* functions. x_1, d_1 belong to *offset2*. x_2, d_2 belong to *offset* (first call). x_3, d_3 belong to *offset* (second call).

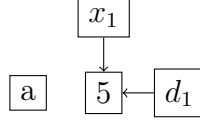
Step 1. Variable initialization in *main* function:



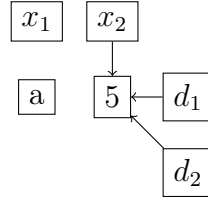
Step 2. Call *offset2*(a, a):



Since a knows x_1 is inout, it releases its pointer and waits for x_1 :

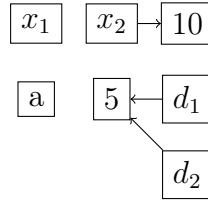


Step 3. Call *offset* inside *offset2*:

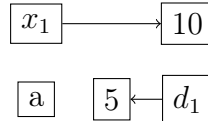


Now x_1 knowing that x_2 is inout, releases its pointer and waits.

Step 4. *offset* function now computes $x_2 += d_2$. Which means that x_2 that was referring to 5 now must create its own copy and store its value:

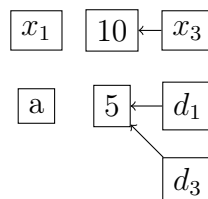


Step 5. After first *offset* function, we now return x_2 reference to x_1 . d_1 no longer exists after function ends:



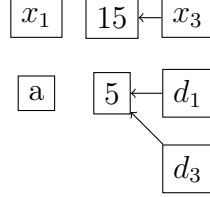
a is still waiting for inout x_1 .

Step 6. Call second *offset* function:



x_1 knows x_3 is inout, and waits for it.

Step 7. After $x_3 += d_3$, we do not need copy on write, since x_3 is the only pointer, so it alternates the data in place:



Step 8. Release all variable that went out of scope of the *offset2* function, and get value from inout variable $x1$ to a : $a \rightarrow 15$

Move Semantics

In the Valeriya programming language, we are treating ADTs, arrays, and strings internally as C pointers to enable MVS logic above. Therefore, the return from the function in Valeriya can be viewed as moving the value from the function's scope to the actual assigned variable. While internally it is a copy of a pointer, conceptually we can view it as a move semantics similar to how Return Value Optimization (RVO) elides copying in C^{++} [5].

2.2.6 Pattern Matching

With an increasing need for more human-readable languages, structural pattern matching has become more common in programming languages [19]. Pattern matching originates from functional programming languages such as Haskell and OCaml, where algebraic data types and structural decomposition are core language features. Although originally founded in functional languages, the matching feature is now being integrated into modern imperative languages [6][2]. It is known as a generalization of **switch/case** and **if/elif/else** blocks to simplify matching with ADTs. The pattern matching equivalent to these will be easier to read, understand, and maintain [19]. Pattern matching consists of two operations: matching and running the underlying subexpression. There are two semantic ways of expressing matching:

First-Order Matching

Matching follows the first-match logic; when the first matched expression is executed, the rest of the code is skipped. Therefore, the more specific case should come before the general case.

Imperative programming languages use this matching logic and include control-flow dependency. Reordering the code leads to a change in behavior. Many use this first-order logic's advantage for limiting the ranges of the next cases using general cases before specific cases. While this provides flexibility and explicit control over the flow, it also introduces the possibility of subtle bugs when clauses are reordered.

Order-Independent Matching

There is a different matching logic called order-independent matching [2]. When matching is performed non-deterministically, we must ensure that only single clause matching is performed, so we ignore case ordering. The authors [19] suggest this type of matching for declarative languages for the following reasons:

- **Clause Order Sensitivity.** Matching is complete and unambiguous, with no two patterns matching in different case clauses. Complete matching should also add a default pattern (or wildcard pattern) that matches when other clauses fail [19]. This eliminates subtle bugs where reordering clauses changes the program's meaning.
- **Equational Reasoning.** Predictable semantics simplifies equational reasoning with one-to-one mapping from matched clause to execution.
- **Ease of Development.** Ordering rules are ignored by programmers while they are only focusing on the patterns and actions.
- **Algebraic Transformations.** Independence of patterns from the program flow lets us manipulate patterns with no significant disturbance to the interpretation of the other patterns.

Historically, pattern matching has been associated with strings and regular expressions as a basis for implementation [17]. Contemporary implementation of pattern matching goes beyond simple value matching as in `switch/case` blocks; according to [13], matching is implemented to handle type matching, polymorphic classes, structure bindings, and their nested variations. Python also extends matching to deduce types and nested structures. **match** keyword is reserved for this case.

The limitations of pattern matching are based on the visible properties of matched values: for simple types, tuples, or arrays; and for the abstract data types, it is not that easy [19]. Simple direct matching fails when it comes to data structures, custom classes, union types, or polymorphic class types. Nevertheless, it depends on compiler specifications and type checking, so it is possible to allow matching in the unusual cases.

This work focuses on matching on **property** type constructs from the Valeriya programming language. Property type is a sum type of its options, where options are the product types storing the information about the option. The property is:

$$opt_1 \mid opt_2 \mid \dots \mid opt_n \tag{2.6}$$

Each option is called a summand of the property type [7]. Matching makes it easy to view valid options, and using type deduction, access fields seamlessly. In programming languages, this deduction to simple variables from the ADT using pattern matching is called structure binding [5][13].

2.3 TreeGen and Tree Virtual Machine

TreeGen is a tool used for generating C^{++} classes for representing abstract syntax tree (AST) nodes as a recursive data type. TreeGen is especially useful for creating language-specific syntax structures to parse them based on the selector types, or options [3]. A highly relevant feature for this work is automatic memory management using the resource acquisition is initialization (RAII) idiom. TreeGen enables efficient

in-place rewriting of tree structures when they are not shared; the RAI and the copy-on-write (COW) principles guarantee the internal behavior of the underlying fields.

Depending on its option, a tree variable can have different fields. This implies that fields do not always exist. There are four types of fields [3]:

- **Prefix Fields** are stored in the stack, and are independent of the current option. They can be used to store the source information of the tree.
- **Local Fields** are stored in the stack, but are dependent on the current option. They can be used to store information about the current statement or expression, such as variable names or type information.
- **Scalar Fields** are stored in the heap, and are dependent on the current option. They can be used to store recursive data; an if statement stores another statement or expression.
- **Repeated Fields** are stored in the heap, and are dependent on the current option. They act as a vector and can store contiguous blocks of statements or expressions.

When developing a programming language in C^{++} , the classes generated by TreeGen can form recursive expressions and statements of the language. The TreeGen also provides efficient code by not using heavy polymorphic classes but raw tagged union types dependent on the selector types [3][4].

A polymorphism-based approach creates runtime complexity both in time and memory. Deriving from the base **Statement** class would load every polymorphic class with a special pointer called virtual table pointer (vtable), which is used to dynamically point to the proper implementation method of the derived class [5]. Additionally, the generated AST would be a list of pointers to the base **Statement** class, and each instance check using dynamic cast is runtime costly.

The TreeGen implementation is based on the Tree Virtual Machine (TVM) fields. The TVM fields are the building blocks that use RAI and COW idioms. TreeGen creates one big class **Statement**, which, based on its selector type, stores only the

relevant fields. With no runtime overhead and meticulous memory management, we achieve a compile-time performance boost.

Input format

We must provide the simple instructions in a specified file extension (.rec) for the target TreeGen classes to be generated. Referring to the Extended Backus-Naur form (EBNF) in the Appendix section we create two main recursive classes: **Expression**, **Statement**.

To create an option in the class for TreeGen, we specify this structure:

```
%define Expression (EmptyExpr)

%option OPTION_NAME { OPTION_SELECTOR } =>
[ REPEATED_FIELD : TYPE ],
SCALAR_FIELD_1 : TYPE, ...
# LOCAL_FIELD_1 : TYPE, ...
```

This is an example code used to add **fields** to the **option** in the **definition**. The fields are one of: scalar field, repeated field, and local field.

Only one field can be a repeated field in a single option. Therefore, if our statement (option) needs more than one repeated field, it should recursively refer to another option of the main class that stores the additional repeated field [3] or use `std::tuple` as a type for the repeated field.

Scalar fields are used for recursively nesting statements. Because scalar fields are pointers, we can achieve storing statements in other statements.

Local fields store specific information about the current option: variable name, function name, type information, attributes of the identifiers, and so on.

The TreeGen tool, developed by H. de Nivelle [3], is used for Parsing and Semantic Analysis of this language. The statements and expressions of the language are generated by the parser from a list of tokens passed by a Lexer. The following received Parse tree in the form of a TreeGen class is then sent to the Semantic Analysis layer

and soon to the CGenerator class to generate C intermediate representation.

Chapter 3

Valeriya Programming Language Design

3.1 Primitive Data Types

Primitive-like built-in types in Valeriya include: **int**, **uint**, **double**, **char**, **bool**, **string**. The types directly match the C's primitive types, except for the string. **int**, **uint** are 32-bit integer and unsigned integer types; **double** is a 64-bit numeric type; **char**, **bool** are 8-bit character and boolean types; **string** is not a direct map to C primitive types, and is a pointer array of characters.

The primitive types in Valeriya, except strings, store distinct values and do not use copy-on-write for their variables, since copying primitive types is cheap and efficient as is. Strings internally are arrays of varying sizes; therefore, reference semantics are internally utilized for them.

Syntactic Examples:

```
int a = -5;
double pi = 3.14
char c = 'x';
string s = "Hello";
uint b = 5;
```

```
bool bl = true;
```

Operations on Primitive Types

For numeric types, the operations are the same as in the C language: addition, subtraction, multiplication, division, and the modulo operator. Valeriya also adds the exponent operator given as `**` for the numeric types, similar to Python's exponent operator. Numeric types can be compared using: `<=`, `>=`, `==`, `!=`, `>`, `<`. The result of the comparison is a Boolean type.

Boolean types in Valeriya support logical binary operators: `&&`, `||`, `==`; and logical unary negation: `!`.

Strings support concatenation, deletion, indexed access, and a special reserved function *len*.

- String concatenation allows merging two strings or a string with a char to a new string. $string + string \rightarrow string$, $char + string \rightarrow string$, $string + char \rightarrow string$.
- String deletion allows deleting several characters either from the beginning or the end. $string - int \rightarrow string$, $int - string \rightarrow string$. If the right operand is an integer, then we delete that number of characters from the back of the string, and from the beginning otherwise.
- String indexing is similar to array indexing in all programming languages; we define an integer expression inside square brackets to get a character at the given index. $string[int] \rightarrow char$.
- Length function returns the actual number of characters in the string. $len(string) \rightarrow int$.

3.2 Algebraic Data Types

3.2.1 Struct Types

Valeriya, just like C, allows creating structs, an aggregate of variables of one type. Structs are product types of their fields given as:

$$S = \prod_{i=1}^n (l_i : T_i) = \{l_1 : T_1, l_2 : T_2, \dots, l_n : T_n\} \quad (3.1)$$

Where $l_i \in \mathcal{L}$ is a label of the field i in the set of all labels $\mathcal{L}$, and $T_i \in \mathcal{T}$ is a type of the field i in the set of all types $\mathcal{T}$. A field of ADT is a type binding $l_i : T_i$.

The size of the struct type is given as a sum of all of its fields:

$$\text{sizeof}(S) = \sum_{i=1}^n (\text{sizeof}(T_i)) \quad (3.2)$$

3.2.2 Enum Types

Enumerated types are sum types that store a set of unit types:

$$E = \sum_{i=1}^n (tg_i) = tg_1 \mid tg_2 \mid \dots \mid tg_n \quad (3.3)$$

Although enumerated types are sum types (ADTs), they internally do not use pointers and are passed by value in Valeriya. The memory footprint of an enum is determined by the number of variants it contains:

$$\text{sizeof}(E) = \begin{cases} n \leq 2^8, & \text{sizeof}(E) = 1 \text{ byte} \\ 2^8 < n \leq 2^{16}, & \text{sizeof}(E) = 2 \text{ bytes} \end{cases} \quad (3.4)$$

3.2.3 Property Type Definitions

Valeriya programming language implements an abstract type construct called property. The property definition is a specific variation of sum types or union types. A sum type (tagged union) is an algebraic data type (ADT) that stores n variables as

variants. Unlike product type (structs), which allocates memory for each of its n variables, sum type allocates just enough memory to store its biggest type. For the tagged unions, the memory buffer is reinterpreted as the active variant's type based on the value of the tag [8].

A property type is given as an "OR" relationship of the underlying product types called options, $O = S$. Option is a struct type stored inside the property type. Property types are tagged union types, so they allow users to map specified option names to access their option fields. Then a property P is:

$$P = \sum_{i=1}^n (tg_i, O_i) = (tg_1, O_1) \mid (tg_2, O_2) \mid \dots \mid (tg_n, O_n) \quad (3.5)$$

Where $O_i \in P$ is a component option of the property type, and $tg_i \in P$ is a component tag that distinguishes the option from others.

The size of the property type is given as the sum of the size of the tag and the size of the biggest option. Since tags are enum types in the language, their sizes are equal:

$$\text{sizeof}(P) = \text{sizeof}(tg) + \max\{\text{sizeof}(O_1), \text{sizeof}(O_2), \dots, \text{sizeof}(O_n)\} \quad (3.6)$$

3.2.4 Recursive Declaration

The property structure allows recursive declarations inside option definitions. Since properties are also types, $P \in \mathcal{T}$, we allow properties to be declared inside another property. Struct types are also allowed to be recursive, $S \in \mathcal{T}$.

A nested design is useful if we want to create sub-properties of the general property. For example, the option **and** in the property **Formula** can store fields *Formula l*, *Formula r* to perform logical *and* in the form of a tree:

```
property Formula
{
    boolval: { bool v; }
    not      : { Formula v; }
```

```

    and      : { Formula l; Formula r; }
    or       : { Formula l; Formula r; }
    implies: { Formula l; Formula r; }
}

```

Here **Formula** is a type in $\mathcal{T}$. **{boolval, not, and, or, implies}** are the option tags, while the expressions following the colon symbols are options, $\{O_{boolval}, O_{not}, O_{and}, O_{or}, O_{implies}\}$.

The same way, struct types allow recursive fields such as:

```

struct ListNode
{
    ListNode node;
    int val;
}

```

Here **ListNode** is a type in $\mathcal{T}$. **{node:ListNode, val:int}** are fields, $\{l_1 : T_1, l_2 : T_2\}$.

Valeriya hides the pointers for the recursive **Formula** and **ListNode** types because it conserves value semantics. Recursive expansion of values creates ADTs with infinite sizes. Therefore, Valeriya interprets the recursive calls as C pointers to the heap memory to expand the ADTs at runtime. Each pointer in Valeriya is 8 bytes, so at compile time, we know the sizes for our ADTs. Nevertheless, Valeriya variables should be viewed as value containers because Valeriya takes care of ownership and memory safety internally.

3.2.5 Disallowed field calls on field calls

Valeriya bans field calls on field calls for its ADTs. These restrictions were introduced to simplify semantic analysis and preserve predictable mutation behavior in MVS. In particular, deeply chained field access would require more sophisticated alias and pointer ownership tracking mechanisms.

3.3 Handling Arrays

Valeriya supports single-dimensional arrays of any type in $\mathcal{T}$ only. Multidimensional arrays often create semantic confusion and weaken expressiveness, requiring complex pointer arithmetic implementation. This contradicts the motivation of the language of creating an MVS environment with a safe type system and clear mathematical reasoning. Therefore, multidimensional arrays are forbidden for Valeriya. Instead, we can take advantage of the ADTs in Valeriya, promoting more readable and formally sound design:

```
struct Array1D
{
    int n[];
}
struct Array2D
{
    Array1D n[];
}
```

There are two types of arrays in Valeriya: static and dynamic. The names suggest that static arrays are stored in the stack with the sizes known at compile time, while dynamic arrays are stored in the heap and can shrink and extend in size.

- Static arrays are created by specifying their size in square brackets for the variable. For example, `int n[5]` - an integer array of size 5.
- Dynamic arrays are created just by typing empty square brackets. For example, `int n[]` - a dynamic integer array.

Operations on Arrays

Indexing is supported for both arrays. Elements of initialized array variables can be accessed with square brackets and an index inside the brackets. While semantically

any unsigned integer is accepted for indexing, they do not check if out-of-bound errors occur; hence, memory safety should be enforced by programmers.

For dynamic arrays, Valeriya introduces simple operations for manipulation, similar to string types:

- Concatenation: $array + array \rightarrow array$.
- Removal: $array - int \rightarrow array$, $int - array \rightarrow array$.
- Indexing: $array[int] \rightarrow type$.
- Length: $len(array) \rightarrow int$.

3.4 Deterministic Behavior Using Value Semantics

An imperative language must allow programmers to manipulate the program's state using variables. A variable in Valeriya can be an instance of primitive types, struct types, enumerated types, and property types. Valeriya promotes deterministic and predictable behavior of its states by controlled mutable value semantics (MVS). This control is managed in the semantic analyzer layer of the Valeriya compiler.

3.4.1 Mutable Value Semantics

Mutable value semantics (MVS) is a type of semantics in programming languages, where values uphold their independent states while internally using reference semantics for optimization and runtime memory usage [14]. Pure value semantics is rigid and is unable to handle recursive dependence in complex ADTs. MVS solves this problem by allowing implicit pointer and reference semantics and optimizing the memory efficiency behind obscured independent values. The concepts used in MVS: Copy-on-Write (COW), Inout variable handling (output parameters in functions), and move semantics on initialization and function returns.

Definition 1. A semantic environment σ is a mapping from variables to memory locations, and a memory model μ is a mapping from memory locations to values,

where every value has a reference counter. If c is a statement in the language, then $(\sigma, \mu) \xrightarrow{c} (\sigma', \mu')$: σ', μ' means that the statement c transforms σ, μ into σ', μ' . Valeriya sets condition that any transition c from (σ, μ) to (σ', μ') always follows MVS properties.

3.4.2 Copy-on-Write

Copy-on-Write (COW) is a resource management mechanism that allows implicit sharing in the implementation, while language users view it as copying. MVS includes COW to optimize variable sharing with full or temporary read-only access. In Valeriya, COW is triggered on struct types, property types, and array variables, since copying is computationally expensive for them.

The COW mechanism uses reference counting (rc) for the variables.

Definitions 2. For a semantic environment σ , and a memory model μ .

1. Let $a : T \in \mathcal{L}$ be a variable in the language of type $T \in \mathcal{T}$ with $\sigma(a) = L_1$, where L_1 is memory location; we have: $\mu(L_1) = (v_1, rc_1 = 1)$.

Since $rc_1 = 1$, if we assign with rvalue v_2 : $(\sigma, \mu) \xrightarrow{a \leftarrow v_2} (\sigma, \mu')$,

Then mutation happens in place: $\sigma'(a) = L_1$, and $\mu'(L_1) = (v_2, rc_1 = 1)$.

2. Let $a : T \in \mathcal{L}$ be variable in the language of type $T \in \mathcal{T}$ with $\sigma(a) = L_1$, where L_1 is memory location, with $\mu(L_1) = (v_1, rc_1)$. If $b : T \in \mathcal{L}$ is assigned with an lvalue, a : $(\sigma, \mu) \xrightarrow{b \leftarrow a} (\sigma', \mu')$, then we increment L_1 's reference counter: $\sigma'(a) = \sigma'(b) = L_1, \mu'(L_1) = (v_1, rc_1 \geq 2)$; we also decrement the reference counter for a 's previous memory location. Now we have two variables with read-access to L_1 .

3. Let $a, b : T \in \mathcal{L}$ be variables in the language of type $T \in \mathcal{T}$ with $\sigma(a) = \sigma(b) = L_1$, where L_1 is memory location; we have: $\mu(L_1) = (v_1, rc_1 = 2)$.

If we transition the semantic state by assigning to an rvalue: $(\sigma, \mu) \xrightarrow{a \leftarrow v_2} (\sigma', \mu')$

Then $\sigma'(a) = L_2$ since $rc_1 > 1$, with $\mu'(L_2) = (v_2, rc_2 = 1)$, and $\sigma'(b) = L_1$ and $\mu'(L_1) = (v_1, rc_1 = 1)$. Now we have distinct values for different variables.

4. If $rc_i = 0$, then deallocate L_i .

An lvalue is a value with a constant address to which you can assign values. Rvalue is a temporary value with no address.

From the definitions, if an lvalue is assigned to a variable, the variable now refers to the memory of that lvalue, decreasing its previous memory location's reference counter. If an rvalue is assigned to a variable and the reference counter is greater than 1, it now allocates its own memory, decreasing the previous memory location's reference counter; mutation happens in place. When a reference counter for a memory location is zero, the memory location is freed.

If the reference counter for the memory L is greater than 1, and a tries to change its value, it will create its copy despite being the initial pointer to the memory. This is expected behavior, since we do not want uncontrolled mutation happening to other variables referring to L .

Valeriya has trivial and non-trivial types. For trivial types with sizes known at compile time, the operation consists of a bitwise copy of the right operand. For non-trivial arrays, strings, and recursive ADTs, each element is copied individually, allocating space in the heap, and not breaking value independence [14]. This approach reduces unnecessary copies because copied values might never be mutated and could be shared.

3.4.3 Inout Parameters

A common practice in C/C++ programming languages is the use of output parameters. Usually, C developers use the return value of a function to provide error messages and use output parameters to retrieve the results from a function. The approach boosts the performance and code maintenance; it allows having multiple output parameter pointers instead of creating a specialized struct for output pointers. Instead of dealing with reference bindings, Valeriya reserves a special keyword called **inout** that follows copy-in and copy-out semantics, ensuring MVS remains intact. Valeriya promotes **inout** variable uniqueness and safety, preserving the efficiency similar to

the C output parameters. A conceptual example for input handling is given in Section 2.2.6.

Definitions 3. For a semantic environment σ , a memory model μ , a function $f : (\mathcal{L}_f : \mathcal{T}) \rightarrow \mathcal{T} \mid \mathbf{void}$ with a σ_f, μ_f .

1. Let $a : T \in \mathcal{L}$, and $x^{inout} : T \in \mathcal{L}_f$ with $\sigma(a) = L_1$. If we pass a to f with $(\sigma, \mu) \xrightarrow{\text{pass } a \text{ to } f} (\sigma', \mu')$, we get $\sigma_f(a) = \sigma_f(x^{inout}) = L_1$ with $\mu_f(L_1) = (v_a, rc = 1)$, and $\mu'(\sigma'(a)) = (f(a), rc = 1)$.
2. Let $a \in \mathcal{L}$, and $x_1^{inout}, x_2^{inout} \in \mathcal{L}_f$ with $\sigma(a) = L_1$. If $\sigma_f(a) = \sigma_f(x_1^{inout}) = \sigma_f(x_2^{inout})$, then this is a runtime error. Inout uniqueness is not satisfied.

Proof of uniqueness for input parameters enables proper mapping from the actual passed value to the awaited memory location. The state and memory of the function (σ_f, μ_f) form an independent state transition that follows other MVS techniques inside the function, see **Definition 1**.

3.5 Matching and Type Deduction

The property is a union structure or a sum-type of its defined options. Hence, one option can be active and safe to access at a time; a property's size is the maximum size of one of its options. A definitive difference between usual matching and property matching is that the property matching allows matching options and automatically casts the matched variable to its option definition and accesses its fields safely ($\langle \text{var_decl} \rangle$ in the $\langle \text{option_def} \rangle$). Programmers can use this feature to match types, which is less direct in many imperative programming languages such as C^{++} and Java. For example, one can use polymorphism and inheritance in OOP languages, but they still need to downcast and check if it's safe to access: calling **instanceof** in Java or using **dynamic_cast** in C^{++} .

Matching follows a first-match semantics, where the first applicable pattern is executed. Since property options are discrete and mutually exclusive, ambiguity does

not arise. Aggregating options using operator *or* (`|`) is possible, though the fields will not be able to accessed.

3.5.1 Type Deduction of Options

Property matching enables automatic deduction of the active option type inside case clauses. The following example demonstrates recursive matching on a Formula property type.

```
property Formula
{
    boolval: { bool v; }
    not      : { Formula v; }
    and      : { Formula l; Formula r; }
    or       : { Formula l; Formula r; }
    implies: { Formula l; Formula r; }
}

fn eval(Formula expr) -> bool {
    match (expr) {
        // ...
        case and ->
            { /* expr.l and expr.r are available here */ }
        // ...
    }
}
```

If one option matches, then the other options will not match according to the first-order logic. This means that the compiler can exit the match statement after executing the case clause body. The match statement also adds a *wildcard pattern* (`case _`) to match any option; however, it must be used to assert some unhandled options, rather than matching to the remaining options. Because the wildcard pattern

does not pay attention to the option it matches against, we cannot downcast or type-
deduce its fields. Therefore, it is considered a mechanism for throwing errors or
sending messages.

In order to make equational reasoning possible, it is recommended not to match
the same option more than once. On top of that, the wildcard pattern should always
come at the end. While the wildcard pattern is allowed to be placed at the beginning
of a match statement by semantic rules of the language for debugging purposes, any
case clause following the wildcard pattern becomes dead code.

Matching is also allowed for enum types. C programming language implements
matching for enums using **switch/case** statement. With the direct translation to C,
Valeriya overloads its match statement for enums, too.

3.5.2 Property Type — Use Cases

Properties uniquely allow the creation of typed options. The main idea came from
C++'s `std::visit` function, which can be used for pattern matching on `std::variant`.
The LLVM compiler infrastructure uses curiously recurring template pattern (CRTP)
with `std::variant` to effectively achieve static polymorphism [10][5]. Similarly, TreeGen
stores options for the active language statement or expression to perform parsing and
semantic analysis [3]. `std::visit` is too rigid, and its structures are always fixed. Va-
leriya removes the heavy templated *C++* code to a more readable design, simplifying
the process of writing an interpreter or compiler because of native tree-like recursive
access.

```
property Statement
{
    VariableInit: { string varname, typename; }
    Condition    : { string condition; Statement satisfied[]; }
    WhileLoop    : { string condition; Statement wl_body[]; }
    // ...
}
```

The recursive declaration adds flexibility and coverage of the expressions. The following example shows how property types unravel calculus differentiation.

```
property Expr
{
    Var    : { string name; }
    Const  : { double value; }
    Add    : { Expr a; Expr b; }
    Sub    : { Expr a; Expr b; }
    Mul    : { Expr a; Expr b; }
    Div    : { Expr a; Expr b; }
    // Cos, Sin, Tan, ...
}

fn diff(Expr e, string var) -> Expr
{
    match (e)
    {
        case Const -> { return Const{0}; }
        case Var -> {
            if (e.name == var) {
                return Const{1};
            }
            else {
                return Const{0};
            }
        }
        case Add -> {
            return Add{diff(e.a, var), diff(e.b, var)};
        }
        case Sub -> {
            return Sub{diff(e.a, var), diff(e.b, var)};
        }
    }
}
```

```

    }
    case Mul -> {
        return Add{
            Mul{diff(a, var), b},
            Mul{a, diff(b, var)}
        };
    }
    case Div -> {
        return Div{
            Sub{
                Mul{diff(a, var), b},
                Mul{a, diff(b, var)}
            },
            Mul{b, b}
        };
    }
}

```

Valeriya can also help to implement different data structures as LinkedList:

```

struct ListNode
{
    ListNode node;
    int val;
}

fn Traverse(ListNode node) -> void
{
    if (node == null) { return; }

    printf("->%d", node.val);
}

```

```
    Traverse(node.node);  
}
```

This example shows how Valeriya’s syntax simplifies writing script commands for game engines:

```
property Command  
{  
    Move: { double dx, dy; }  
    Rotate: { double angle; }  
    Stop: {}  
}  
  
//...  
  
fn RunCommand(Command cmd)  
{  
    match(cmd)  
    {  
        case Move -> { /* ... */ }  
        case Rotate -> { /* ... */ }  
        case Stop -> { /* ... */ }  
    }  
}
```

The rest of the syntax for the language and available tokens are given in the **Appendix** section.

Chapter 4

Implementation

4.1 Compiler Implementation Details

The target compiler for Valeriya¹ will consist of six modules: Lexical Analysis, Parser, Semantic Analysis, Interpreter (C Generator), Code Generator, and Error Handler. We call lexical analysis, parser, and semantic analysis the analysis phases. And we call the internal representation and code generator the synthesis phases. The error handler is directly built into each analysis phase and is responsible for locating and throwing compilation errors. The code generator is an LLVM C compiler called Clang. Valeriya's IR is C code, which is ready to be compiled with the LLVM backend. The modular design of the language is given in the Fig. 4-1.

¹github link: <https://github.com/Zhandr678/Valeriya-Programming-Language>

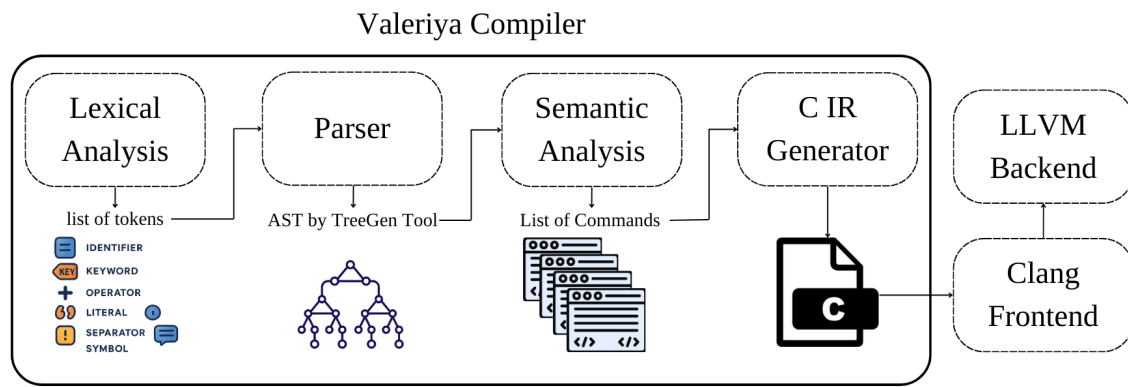


Figure 4-1: Valeriya Compiler's Design

4.2 Lexer

The compilation process begins with scanning the source file and generating tokens. Lexers are the layers in the compiler that structure a list of characters into the explained list of tokens (λ, a, I) . λ is label, a is attribute, and I is location information. We define Λ as a set of labels $\Lambda = [\mathbf{Keyword...}, \mathbf{Identifier...}, \mathbf{Literal...}, \mathbf{Operator...}, \mathbf{Comment...}]$.

Valeriya compiler's lexer assigns labels based on the language-defined regular expressions (regexes). The regular expressions for Valeriya are the same as the C language's. We assign each label to the following categories:

For a regexes:

```
digit  → 0|1|2|...|9
letter → a|b|...|z|A|B|...|Z
symbol → +|-|^|!|&|_|...
space  → [ ]
```

- **Keywords** are reserved words of the language. They are expected to form statements and can not be used as identifiers. If a parser meets a keyword, it immediately tries to parse it into the statement as defined in the language's grammar. The token labels for keywords start with a **KW_** prefix, see Appendix.
- **Identifiers** are custom variable names defined by users. For this language, the names can be given to types, structs, functions, options, and properties. Identifiers are checked against the language's dictionary to not overlap with keywords.

The regex for identifiers:

```
identifier → [letter|_]+ [letter|digit|_]*
```

- **Literals** are primitive rvalues to be stored in variables, such as quoted strings, numbers, true/false literals, and null. The token labels for literals start with a **LIT_** prefix, see Appendix.

1. An integer regex:

$$\text{integer} \rightarrow 0 \mid [\text{digit}^0] [\text{digit}]^*$$

2. A double regex:

$$\text{number} \rightarrow \text{integer} (\backslash.[0-9]^*)? ([e \mid E] [+ -]? [0-9]^+)?$$

3. A quoted string regex:

$$\text{quote} \rightarrow [""] [\text{symbol}^{" \mid \backslash n}]^* ["]$$

4. A char regex:

$$\begin{aligned} \text{char} \rightarrow [&'] ([\text{symbol}^{'})] \mid [\backslash n] \mid [\backslash t] \mid [\backslash 0] \mid \\ &[\backslash r] \mid [\backslash \backslash] \mid [\backslash '] \mid \text{space}) [&'] \end{aligned}$$

- **Operators and Symbols** are 1-2 character long definitions. They are used for preserving language semantics using punctuation and for applying defined operations to their operands. The token labels for operators start with **SYM_** prefix, see Appendix.
- **Comments** are long blocks of text that do not result in a token. They are used to provide information to programmers. Single-line comments start with `//` symbols, and multiple-line comments start with `/*` and end with `*/`.

The regexes for comments:

$$\text{any_char} = \text{digit} \mid \text{letter} \mid \text{symbol} \mid \text{space}$$

```

com_line  → [//]+ [any_char^(\\n)]*
com_block → [/*] [any_char^[*] | [*]]* [*/]

```

- **End-of-File (EOF).** The last token in the tokens list. It is used to tell the parser that there are no more tokens for parsing.

A minimized deterministic finite automaton (DFA) for the regexes above is directly integrated into the lexer module of Valeriya. The lexer was implemented manually in *C++*. The main method for the lexer is *ReadAndClassifyNext*, which analyzes the next sequence of symbols from the file reader's buffer and constructs a token. The method continuously returns the end-of-file token if no more tokens are in the buffer. All of the labels for tokens are given in the Appendix.

Failing to satisfy a regex leads to a lexical exception. In Valeriya, the lexical exceptions include: invalid numerical tokens, unterminated quotes, unterminated block comments, and unknown symbols (examples: '@', '#', '\$').

4.3 Parser

After the list of tokens *T* is generated, we pass them to the parser to get an abstract syntax tree (AST). For Valeriya, the AST is a recursive **Statement** class generated by the TreeGen tool. A **Statement** is a general n-ary tree, in which the program is ordered in the depth-first search (DFS) manner. More precisely, if an AST node is not a leaf node, the control flow enters that node to follow the program's chronological order. For example, from the fig. 4-2, the flow is:

$S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow (\text{Entering the Block } S_3) \rightarrow S_1^{S_3} \rightarrow S_2^{S_3} \rightarrow \dots \rightarrow$
 Exiting the Block $S_3 \rightarrow S_4 \rightarrow (\text{Entering the Block } S_4) \rightarrow \dots$

The rules for the **Statement** are from the extended Backus-Naur form (EBNF) from the Appendix. Additionally, in the Appendix section, there is a source (.rec) file

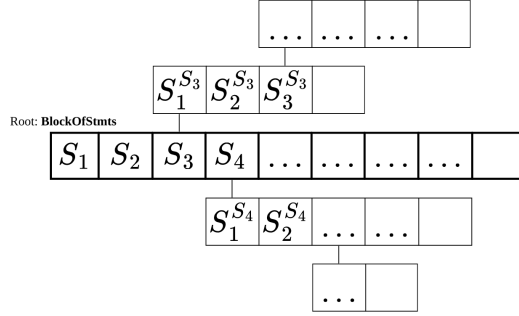


Figure 4-2: Abstract Syntax Tree (AST) of Statements of Valeriya Programming Language

used for automatic generation of the tree in the TreeGen tool. A root of the AST has a selector option *BlockOfStmt* with a repeated field of type **Statement**, see Fig. 4-2.

In Valeriya's AST, we have: root node, internal nodes, and leaf nodes. Every node is a **Statement** class, which is a sum type of all dependent recursive TreeGen options.

- A root node is the beginning of the program. It has the option of *BlockOfStmt* and stores a scalar vector of repeated **Statements**.
- An internal node is a child node of the root or other internal nodes. In Valeriya, the recursive internal nodes include: **BlockOfStmt**, **ConditionStmt**, **WhileLoopStmt**, **ForLoopStmt**, **MatchStmt**, **CaseClauseStmt**, **MakeFunctionStmt**. They must be the internal nodes because we execute the recursive bodies, **BlockOfStmts**.
- A leaf node is a child node that does not store other nodes. The instances of leaf **Statement** nodes are: **VarInitStmt**, **ArrayInitStmt**, **AssignmentStmt**, **MakeFunctionStmt**, **MakeStructStmt**, **MakeEnumStmt**, **MakePropertyStmt**, **ReturnStmt**, **BreakStmt**, **ContinueStmt**, **ExprCallStmt**. Since each node in the AST must be the same type, the **Expression** class is contained in the **Statement** class under **ExprCallStmt** selector option.

Expressions are also given as tree structures and can be divided to internal and leaf nodes.

- Internal **Expression** nodes: **ArrayIndexExpr**, **InitListExpr**, **FieldCallExpr**, **StructInitExpr**, **FnCallExpr**, **UnaryExpr**, **BinaryExpr**. Each internal expression can store more expressions recursively as its fields. For example, **ArrayIndexExpr** stores recursive *at* expression, that should be of type **int** during semantic analysis phase.
- Leaf **Expression** node: **EmptyExpr** (null), **IntLiteralExpr**, **DoubleLiteralExpr**, **BoolLiteralExpr**, **StringLiteralExpr**, **CharLiteralExpr**, **VarNameExpr**. These nodes store a direct value which then can be translated to the IR after a semantic verification.

In the syntactic analysis, internal nodes typically represent non-terminal grammar rules, while leaf nodes represent terminal symbols or literal values. The fields for each **Statement** and **Expression** classes' options are given in the (.rec) file in the Appendix section.

4.3.1 Algorithms For Parsing

A parsing algorithm used in Valeriya is a top-down backtracking parser with recursive descent written manually in C^{++} . At first, it felt easier to implement a top-down LL parser with the given context-free grammar by hand; larger classes of LR grammars are usually constructed using automated tools [1]. But, the implementation from hand introduced many small caveats that required careful attention and exact logic implementation. Nevertheless, it is suggested to use ready verified tools to save more time and eliminate mistakes when developing.

- **Deterministic Selection.** If a look-ahead token $t_i \in T : A \rightarrow \alpha_1$ corresponds to the only grammar (single production rule, α_1 , for a given non-terminal symbol A), then it consumes the tokens until a terminal token of that specific grammar.

For any deviation from the grammar, the parser throws a parser exception with an error message and location information.

- **Non-deterministic Selection.** If a token $t_i \in T : A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots$ matches more than one grammar, then it chooses the grammar in the current order (α_i) and tries to parse it. If parsing fails for the current grammar, the parser rewinds its stack and tries to match the next grammar in the order (α_{i+1}). Failing to parse any grammar rule causes an invalid statement exception.

A parsing rule that deterministically matched to the specific grammar, α , triggers the *ResetLookAhead* function to consume the tokens in the backtracking stack as proper grammar syntaxes. Subsequently, for the given grammar, a parsing rule can recursively descend to the next nested grammar rule. The result of parsing is the TreeGen class AST of **BlockOfStmt** option.

4.3.2 Operator Precedence Parsing

Valeriya utilizes operator precedence parsing for its expression constructs. Operator precedence determines the order in which operators are parsed; thus, their position in the parse tree. In some low-level languages, operations strictly follow left-associativity without allowing parentheses [20], while modern high-level languages follow mathematical operator precedence with left-to-right associativity.

The following table demonstrates the weight order of operators in Valeriya, assuming operations are semantically allowed by the language. The precedence level of 0 - operator is located closer to the root of the parse tree, while the level of 3 - operator is deeper in the tree [1][20]. Operators with equal precedence are parsed as left-associative since the algorithm selects the rightmost operator as the main splitting point during the recursive expression construction. The parser prioritizes the brackets before the actual operators. So, any expression enclosed in parentheses " $(...)$ ", brackets " $[...]$ ", or braces " $\{...\}$ " are parsed prior other operators.

For example, expression:

```
int i = 3 * n[5 + 10 * 2] - (2 ** 4);
```

Precedence	Operators
3	**
2	* / \%
1	+ - & ^
0	== != < > <= >= &&

Table 4.1: Operator precedence levels used by the Valeriya parser. Higher value means higher precedence.

will have the following tree of selector option **BinaryExpr** of the TreeGen class **Expression**, see Fig 4-3.

4.4 Semantic Analysis

Valeriya semantics is given as:

$$S = \{\sigma, \mu, \mathcal{L}, \mathcal{T}, \mathcal{F}\} \quad (4.1)$$

where, σ is a semantic environment used for mapping variables to memory location (see Chapter 3), μ is a memory model for reference counting and value tracking (see Chapter 3), $\mathcal{L}$ is a symbol table that maps variables to types, $\mathcal{T}$ is a type table that stores ADT's type information, and $\mathcal{F}$ is a function table that stores functions' return types and argument lists. Semantic analyzer closely operates with C IR generator, by passing semantically captured tables $\mathcal{L}, \mathcal{T}, \mathcal{F}$ to it. Therefore, although σ, μ are used for mutable value semantics (MVS) logic in the runtime, they are considered components of the semantics of the language.

Valeriya's semantic analyzer takes a generated AST and verifies without making changes to the AST:

1. No use of undeclared variables, types, functions. For $n \notin \mathcal{L} \mid t \notin \mathcal{T} \mid f \notin \mathcal{F}$, throw "does not exist" exception.
2. No repeated declaration of the existing names. For **VarInitStmt**, **ArrayInitStmt**, if $var \in \mathcal{L}$, throw "already declared" exception.

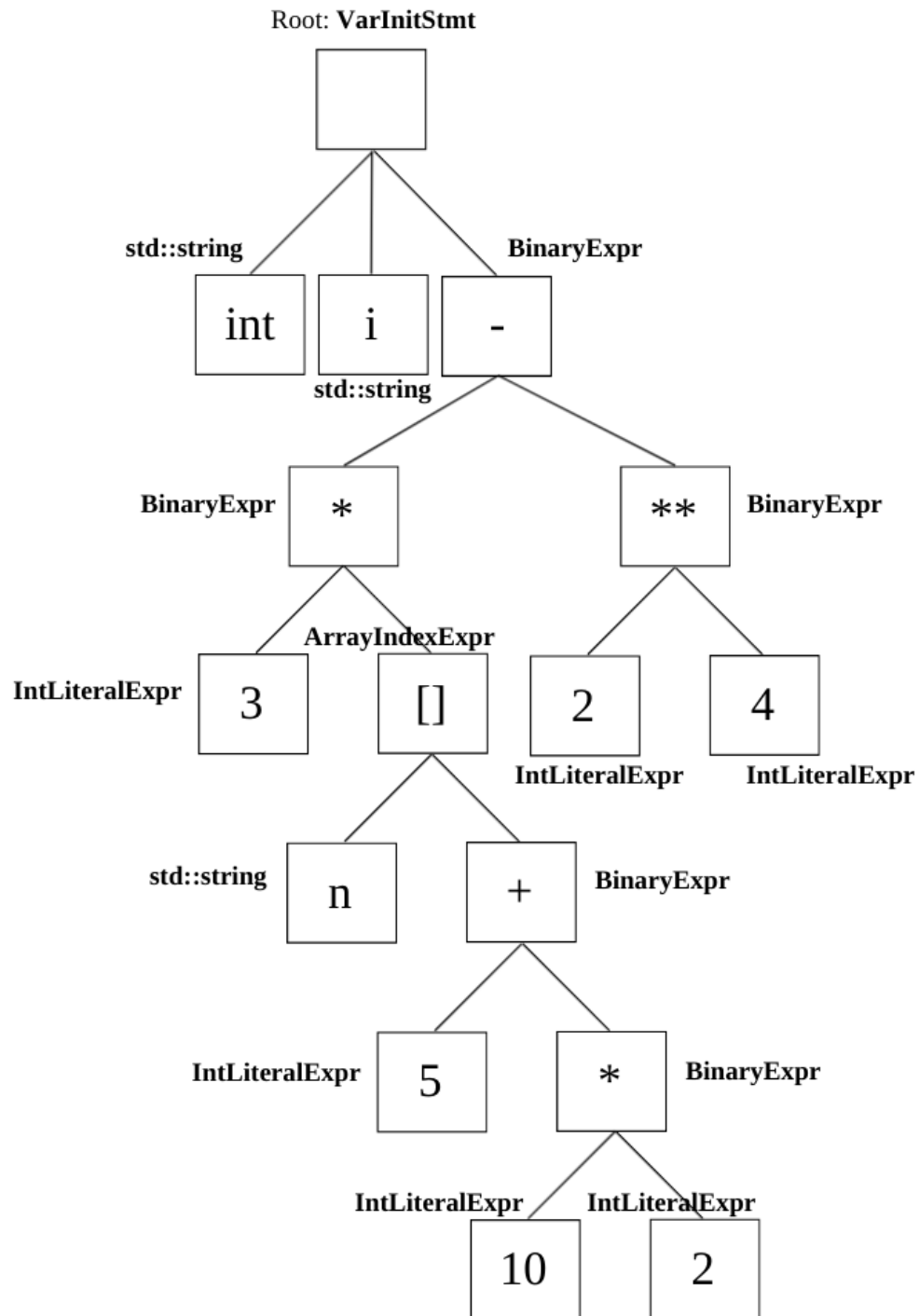


Figure 4-3: Example: generated TreeGen parse tree in Valeriya.

3. No type mismatches during assignment and value passing. In Valeriya, only numeric and boolean types can be implicitly casted and assigned to each other. Additionally, options of property type constructs can also be assigned or passed to a property variable. Otherwise, throw a "cannot deduce type from-to" exception.
4. No invalid operands for given operators. Operands and operators should follow the rules according to the defined operations in Chapter 3. Otherwise, throw an "unexpected operands for operator" exception.
5. Functions returning in all control paths. A function is expected to return in its normal flow, inside if-else, if-elif-else blocks, or in all control paths in the match statement. Otherwise, throw a "function may not return in all paths" exception.
6. Proper type evaluation for expressions. Fields of ADTs must be properly accessed within **FieldCallExpr** expressions. Otherwise, throw a "field does not exist" exception.
7. Proper option deduction in **match** statements. Given the case clause for the matched expression, allow deducing the matched property type to the struct type of its option to access the option's fields.
8. Programming language constraints. Constraints include: forbidden multidimensional arrays, disallowed field calls on field calls, and no post-increment or pre-increment operators.

For each TreeGen **Statement** in the AST, the semantic analyzer checks the 8 points above. If a semantic agreement violation occurs, the compiler throws a semantics exception with a relevant message of the error and the location information.

Semantic analysis for match statements operates on expressions that deduce to a property type or an enum type (**MatchStmt**, **CaseClauseStmt**). For enum types, the match statement operates as C/C++/Java/C#/JavaScript/TypeScript

style **switch/case** blocks. For property types, the temporary expression inside **match** is treated as an instance of the current case clause’s option. So, even though the matched expression deduces to the property type, $P \in \mathcal{T}$, inside the match statement, it is temporarily evaluated to $O \in P \in \mathcal{T}$, allowing programmers to safely decompose complex property types into concrete structure types.

In order to avoid memory corruption issues when manipulating internal compiler pointers, types in Valeriya must be the same before assigning. Type casting is forbidden to ensure a full value semantic experience. However, numerical primitive types are allowed for implicit conversion. Primitive types are not involved in mutable value semantics logic, and they are safe to properly cast in the C language. Therefore, int, double, uint, and boolean types can be assigned to each other, except for double, which cannot be assigned to other types.

4.5 Rules for Mutable Value Semantics

To support mutable value semantics (MVS) during the runtime, Valeriya must generate runtime reference counting for its ADTs to support copy-on-write (COW) and inout function arguments. Heap memory allocation and deletion are hidden from users with automatic memory handling. To achieve this, Valeriya introduces a HashMap of memory addresses to the heap memory information in the generated C code:

```
typedef void (*_MVS_Deleter)(uintptr_t);
typedef struct HeapMemory
{
    size_t ref_count;
    size_t size_in_bytes;
    _MVS_Deleter deleter;
} HeapMemory;
```

The **HashMap** uses **LinkedList** data structure for its buckets. The **MVSHashMap** is the memory model μ for tracking values and references discussed in Chapter 3.

```

typedef struct MVNode
{
    uintptr_t key;
    HeapMemory value;
    struct MVNode* next;
} MVNode;

typedef struct
{
    MVNode** buckets;
    size_t capacity;
    size_t size;
} MVSHashMap;

static MVSHashMap mu;

```

We introduce the minimally sufficient helper MVS interface functions:

- **MVSInit.** Initializes the HashMap and allocates memory for buckets. Called at the start of any program in Valeriya.
- **MVS_Destroy.** Frees the existing HashMap and its buckets. Called at the end of any program in Valeriya.
- **MVS_RefCount(uintptr_t address).** For the given address returns the current reference count. For uninitialized addresses or NULL returns 0.
- **MVS_RegisterNew(uintptr_t address, size_t size, _MVS_Deleter deleter).** Increments the reference count for the given address, and assigns a deleter. Called for every initialized ADT or array in Valeriya.
- **MVS_DetachPointer(uintptr_t address).** Decrements the reference counter for the given address. MVSRegistered deleter is called on the address if address becomes 0.

- **MVS_SameLoc(uintptr_t a, uintptr_t b).** A function to check if the two pointers referring to the same location.

More importantly, when defining user ADTs (structs, properties), Valeriya must generate rules for using those ADTs in the language. The compile time information for the source language provides enough type information for writing a template for each ADT.

For each struct ADT, Valeriya creates the following functions:

- **xx_free_StructName.** This function is passed to the **MVS_RegisterNew** as a deleter. For each dynamically allocated field in the struct, the function calls **MVS_DetachPointer**. Thus, we free the whole struct ADT, and also decrement reference counters recursively for each allocated pointer field.
- **xx_init_StructName.** Structs can have zero or more fields, and depending on the type, they can be allocated on the heap. This function takes all the fields for the struct ADT and assigns them by calling **MVS_RegisterNew** for each heap-allocated field.
- **xx_clone_StructName.** When copy-on-write is triggered for the variable with a reference counter greater than one, the cloning function should be triggered to bitwise copy the fields from the current variable to the cloned variable. Consequently, the cloning function is called for each recursive ADT field for the given struct unless it is NULL.

For each property ADT, Valeriya creates the following functions:

- **xx_PropertyName_option_is_OptionName.** This function is generated per property ADT's option. For each of these functions, the tag is checked to see if the property currently stores the option of the tag. These functions are used in **match** statements to type-deduce in case clauses' bodies. When freeing the property, it also checks which option is currently active and needs to be freed.

- **xx_init_PropertyName_OptionName.** This function is generated per property ADT's option. Each function initializes the current option and sets a tag for the current option.

The remaining three of the property interface functions are the same as struct ADT's interface functions with similar declaration, deallocation, and cloning.

The implementation of runtime MVS allows Valeriya to strictly follow value semantics concepts supporting equational reasoning, while utilizing pointer manipulations for complex structures, recursive ADTs, and arrays. This supports Dave Abraham's conclusion on Mutable Value Semantics in his programming language, Hylo. By relying on automatic memory management, we achieve safety, predictability, and local reasoning in high-level, copy-on-write code, which avoids unnecessary memory traffic with reference counting, and we take advantage of function return parameters, without adding pointer complexity to the language [14][15].

4.6 Code Generator

The generation of the executable is passed to the Clang/LLVM compiler. While Valeriya acts as a transpiler to the C code, which can be further compiled to the binary using any C compiler, the choice for clang compiler is characterized by the clang's multi-stage code optimization and cross-platform support [10][9]; Microsoft Visual C compiler (MSVC) targets only WinOS, while GCC requires additional MinGW instrument tools for Windows.

CMake is used for building the source code for the Valeriya compiler, and produces an executable called "vri". The source code can be built on Linux, Windows, and MacOS without any changes. However, externally, the systems must have a clang compiler and CMake installed.

- CMake: 3.18+
- Linux, Windows Clang: 10.0+ (C++20)
- MacOS Clang: 12.0+

Nonetheless, it is not obligatory to have a clang compiler installed. The C IR is generated right in the same place as the source file, but with *.c* extension. So, users can compile the C IR manually or write an additional program to compile with a compiler of choice.

Chapter 5

Conclusion

5.1 Overview of Valeriya

This work explained the capabilities of Valeriya imperative programming language based on value semantics, supporting pattern matching on a new ADT construct. The programming model of the language excludes reference semantics and pointer arithmetic from users, allowing a full value semantics experience. Essentially, the semantics rules and memory model following the formal definitions of MVS can be applied to any language attempting to fully operate on value semantics. While the formal definitions of MVS suggest upholding controlled value mutation under both single-threaded and multi-threaded programs, a careful analysis of parallel programs should be conducted.

This work concludes that mutable value semantics (MVS) can be strictly implemented both theoretically and practically, as well as apply it to more complex structures and array types, and not just plain-old-data (POD) types.

The use of the TreeGen tool sufficiently decreased the implementation time for the compiler. It adapts the syntax for parsing rules in a basic language and allows seamless integration into the compiler's code. The tool has proven to be a production-level parse tree generator with minimal overhead.

5.2 Future Works

Valeriya did not practically implement the following programming language possibilities under mutable value semantics (MVS) logic. With rigorous theoretical research, the subjects for possible future extensions for the language are the following:

5.2.1 Generic ADTs

It is possible to allow generic ADTs inside the language. Generic ADTs are the usual ADTs that are constructed per instances of their templated types [5]. For example, in C^{++} :

```
template <class T, class V>
struct Pair
{
    T first;
    V second;
};
```

This struct is the normal struct ADT that is expanded to more instances based on the type values for T and V. Therefore, mutable value semantics (MVS) logic is also supported for each of the instance structs from the template.

5.2.2 Multi-dimensional Arrays

With meticulous control on the pointer arithmetic in the C IR generation part, it is possible to extend array types to support more dimensions, while still preserving mutable value semantics. Since this work did not focus on pointer algebra, the possibility of the language was clipped for more focus on mutable value semantics.

5.2.3 Multithreading and Concurrency

An important direction for future work is the extension of Valeriya with multithreading support. Mutable value semantics (MVS) provides a promising foundation for

concurrent programming, as value-based data ownership can reduce the risks of data races and unintended aliasing that commonly occur in reference-based languages.

However, introducing multithreading requires a careful formal analysis of how mutable values are shared and transferred between threads. While mutable value semantics is immune to data races, mechanisms such as ownership transfer, message passing, or controlled synchronization primitives could be introduced to ensure safe mutation across threads [14]. Additionally, the interaction between copy-on-write semantics and concurrent execution would require precise guarantees to avoid inconsistent program states.

5.2.4 Matching Limitations, Structure Bindings

The matching logic in Valeriya is simple, the same as calling if/elif/else blocks. Although Valeriya allows type deduction in match statements (similar to Rust), matching statements in Python allow matching against arrays, dicts, tuples, and strings, so many possibilities for matching were not applied.

5.3 Conclusion

The work demonstrates that mutable value semantics can be implemented in a practical compiler while preserving predictable behavior and enabling efficient execution. Overall, the results of this work show that it is feasible to design an imperative language where mutable value semantics serve as the primary programming model. The approach provides a balance between the clarity of value semantics and the efficiency of reference and pointer-based implementations. The Valeriya language, therefore, is an example that aims to combine deterministic semantics, expressive type systems, and efficient implementation strategies.

Bibliography

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, USA, 2 edition, 2006.
- [2] D. Binder and L. Ermantraut. The algebra of patterns. In *Proc. 39th Eur. Conf. on Object-Oriented Programming (ECOOP)*, pages 2:1–2:28. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, 2025.
- [3] H. de Nivelles. Manual of TreeGen 2023, February 2024. Available: Nazarbayev University, Faculty Development Competitive Research Grant Program (FDCRGP), Grant No. 021220FD1651.
- [4] Hans de Nivelles. Tree virtual machine (tvm). Technical report, Nazarbayev University, December 2022. Faculty Development Competitive Research Grant Program (FDCRGP) number 021220FD1651.
- [5] M. Gregoire. *Professional C++*. Wiley, 4th edition, 2021.
- [6] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016.
- [7] A. Igarashi and H. Nagira. Union types for object-oriented programming. In *Proc. 2006 ACM Symp. on Applied Computing (SAC)*, pages 1435–1441, Dijon, France, April 2006.
- [8] A. Kennedy and C. V. Russo. Generalized algebraic data types and object-oriented programming. In *Proc. 20th Annu. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–36, San Diego, CA, USA, October 2005.
- [9] C. A. Lattner. LLVM: An infrastructure for multi-stage optimization. M.S. thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, IL, USA, 2002.
- [10] LLVM Project. LLVM 22.0.0git Documentation. [Online]. Available: <https://llvm.org/docs/>.
- [11] Dina Muktubayeva and Hans de Nivelles. Using the tokenizing tools of maphoon. Technical report, Unpublished Technical Report, July 2023.

- [12] H. Mössenböck. Compiler construction—the art of niklaus wirth. In *University of Linz*. 2000.
- [13] M. Park. Pattern matching: match expression. [Online], February 2024. ISO/IEC JTC1/SC22/WG21 N2688 = P2688R1.
- [14] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. Native implementation of mutable value semantics. In *Proceedings of the 16th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '21)*, pages 1–4, Online, July 2021. Association for Computing Machinery.
- [15] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. Implementation strategies for mutable value semantics. *Journal of Object Technology*, 21(2), 2022.
- [16] Robert W. Sebesta. *Concepts of Programming Languages 12th*. Pearson, New York, 2019.
- [17] D. A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In *Proc. ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71. ACM, 1991.
- [18] Y. Su and S. Y. Yan. *Principles of Compilers: A New Approach to Compilers Including the Algebraic Method*. Higher Education Press; Springer-Verlag, Beijing; Heidelberg, 2011.
- [19] S. T. Taft. Rigorous pattern matching as a language feature. *Ada Letters*, XLII(2):69–74, December 2022. Presented at HILT'22, Oakland, MI, USA.
- [20] W. M. Waite and G. Goos. *Compiler Construction*. Springer-Verlag, New York, NY, USA, 1984.

Appendix

A.1 Language Grammar (Extended Backus-Naur Form Definition)

$\langle program \rangle ::= \{ \langle stmt \rangle, \dots \}$

$\langle stmt \rangle ::= \{ \langle expr \rangle$
| $\{ \langle stmt \rangle, \dots \}$
| $\langle var_decl \rangle$
| $\langle func_decl \rangle$
| $\langle var_assign \rangle$
| $\langle struct_def \rangle$
| $\langle enum_def \rangle$
| $\langle for_loop \rangle$
| $\langle while_loop \rangle$
| $\langle cond_stmt \rangle$
| $\langle property_decl \rangle$
| $\langle match_stmt \rangle$
| $\langle return_stmt \rangle$
 $\}$

$\langle ar_binop \rangle ::= + \mid - \mid / \mid * \mid \% \mid **$

$\langle bool_binop \rangle ::= != \mid == \mid \& \mid \mid \mid \&\& \mid \mid \mid \mid \mid OP_LESS \mid OP_LEQ \mid OP_GREATER \mid$
 $OP_GEQ \mid OP_XOR$

$\langle type_name \rangle ::= \text{int} \mid \text{uint} \mid \text{double} \mid \text{string} \mid \text{bool} \mid \text{char} \mid \text{void}$

$\langle type_name \rangle ::= \langle struct_name \rangle \mid \langle enum_name \rangle \mid \langle prop_name \rangle$

$\langle var_decl \rangle ::= \langle type_name \rangle \langle var_assign \rangle \mid \langle var_name \rangle;$

$\langle var_decl \rangle ::= \langle type_name \rangle \{ \langle var_assign \rangle \mid \langle var_name \rangle, \dots \};$

$\langle var_assign \rangle ::= \langle var_name \rangle = \langle expr \rangle$

$\langle func_decl \rangle ::= \text{fn } \langle func_name \rangle (\langle param_list \rangle) \rightarrow \langle type_name \rangle \{ \{ \langle stmt \rangle, \dots \} \}$

$\langle param_list \rangle ::= \{ \langle type_name \rangle \langle var_name \rangle, \dots \}$

$\langle func_call \rangle ::= \langle func_name \rangle (\{ \langle var_name \rangle \mid \langle expr \rangle \mid \langle func_call \rangle, \dots \});$

$\langle struct_def \rangle ::= \text{struct } \langle struct_name \rangle \{ \{ \langle var_decl \rangle; \dots \} \}$

$\langle enum_def \rangle ::= \text{enum } \langle enum_name \rangle \{ \{ \langle enum_option_name \rangle \} \}$

$\langle prop_decl \rangle ::= \text{property } \langle prop_name \rangle \{$
 $\quad \{ \langle option_def \rangle \}$
 $\quad \}$

$\langle option_def \rangle ::= \langle option_name \rangle : \{ \{ \langle var_decl \rangle \} \}$

$\langle match_stmt \rangle ::= \text{match } \langle var_name \rangle \{$
 $\quad \{ \langle case_clause \rangle \}$
 $\quad \}$

$\langle case_clause \rangle ::= \{$
 $\quad \text{case } \{ \langle option_name \rangle \mid \dots \} \rightarrow \{ \{ \langle stmt \rangle \} \}$
 $\quad \mid \text{case } _ \rightarrow \{ \{ \langle stmt \rangle \} \}$
 $\quad \}$

$\langle cond_stmt \rangle ::= \text{if } (\langle expr \rangle) \{ \{ \langle stmt \rangle, \dots \} \}$

$\langle cond_stmt \rangle ::= \text{if } (\langle expr \rangle) \{ \{ \langle stmt \rangle, \dots \} \}$
 $\quad \text{else } \{ \langle stmt \rangle, \dots \}$

$$\begin{aligned}
\langle cond_stmt \rangle &::= \text{if } (\langle expr \rangle) \{ \{ \langle stmt \rangle, \dots \} \} \\
&\quad \{ \text{elif } (\langle expr \rangle) \{ \{ \langle stmt \rangle, \dots \} \}, \dots \} \\
&\quad \text{else } \{ \langle stmt \rangle, \dots \} \\
\langle while_loop \rangle &::= \text{while } (\langle expr \rangle) \{ \{ \langle stmt \rangle, \dots \} \} \\
\langle for_loop \rangle &::= \text{for } (\langle var_decl \rangle; \langle expr \rangle; \langle expr \rangle) \{ \{ \langle stmt \rangle, \dots \} \} \\
\langle return_stmt \rangle &::= \text{return}; \\
\langle return_stmt \rangle &::= \text{return } \langle expr \rangle; \\
\langle expr \rangle &::= \{ \langle var_name \rangle \\
&\quad | \langle expr \rangle \}
\end{aligned}$$

A.2 Token Labels

KW_INT	-> int
KW_UINT	-> uint
KW_DOUBLE	-> double
KW_STRING	-> string
KW_BOOL	-> bool
KW_CHAR	-> char
KW_VOID	-> void
KW_IF	-> if
KW_ELIF	-> elif
KW_ELSE	-> else
KW_FOR	-> for
KW_WHILE	-> while
KW_CONTINUE	-> continue
KW_BREAK	-> break
KW_STRUCT	-> struct
KW_PROP	-> property
KW_MATCH	-> match

KW_CASE -> case
KW_RET -> return
KW_FUNC -> fn
KW_INOUT -> inout
KW_WILDCARD -> _

IDENTIFIER -> a

LIT_INTEGER -> Z (Whole Numbers)
LIT_NUMBER -> R (Real Numbers)
LIT_STRING -> " " (Quoted Strings)
LIT_CHAR -> ' ' (Symbol)
LIT_TRUE -> true
LIT_FALSE -> false
LIT_NULL -> null

SYM_LBRACE -> {
SYM_RBRACE -> }
SYM_LPAR -> (
SYM_RPAR ->)
SYM_LBRACKET -> [
SYM_RBRACKET ->]
SYM_SEMICOLON -> ;
SYM_COLON -> :
SYM_COMMA -> ,
SYM_DOT -> .
SYM_ARROW -> ->
SYM_PLUS -> +
SYM_MINUS -> -
SYM_TIMES -> *

```

SYM_DIV      -> /
SYM_MOD      -> %
SYM_NOT      -> !
SYM_OR       -> |
SYM_AND      -> &
SYM_LOR      -> ||
SYM_LAND     -> &&
SYM_ASSIGN   -> =
SYM_EQUAL    -> ==
SYM_NEQ      -> !=
SYM_XOR      -> ^
SYM_LESS     -> <
SYM_GREATER  -> >
SYM_LEQ      -> <=
SYM_GEQ      -> >=
SYM_POW      -> **

COM_LINE     -> //
COM_BLOCK    -> /* (text) */

WHITESPACE   -> \n | \t | \r | [ ]
EOF          -> End-of-File

```

A.3 (.rec) File Required by TreeGen For AST Definitions

```

%dir ../Sources/Statements
%namespace val

```

```

// ----- EXPRESSIONS -----

#define Expression (EmptyExpr)

%option EmptyLiteral { EmptyExpr } =>
%option IntLiteral { IntLiteralExpr } =>
    # value : int
%option DoubleLiteral { DoubleLiteralExpr } =>
    # value : double
%option BoolLiteral { BoolLiteralExpr } =>
    # value : bool
%option StringLiteral { StringLiteralExpr } =>
    # value : std::string
%option CharLiteral { CharLiteralExpr } =>
    # value : char
%option VarName { VarNameExpr } =>
    # name : std::string
%option ArrayIndex { ArrayIndexExpr } =>
    array_expr : Expression,
    at : Expression
%option InitList { InitListExpr } =>
    [ exprs : Expression ]
%option FieldCall { FieldCallExpr } =>
    caller : Expression,
    field : Expression
%option StructInit { StructInitExpr } =>
    [ inits : Expression ],
    # struct_name : std::string
%option FnCall { FnCallExpr } =>
    [ args : Expression ],

```

```

    # fn_name : std::string
%option Unary { UnaryExpr } =>
    expr : Expression,
    # op : std::string
%option Binary { BinaryExpr } =>
    lhs : Expression,
    rhs : Expression,
    # op : std::string

// ----- STATEMENTS -----

%define Statement (EmptyStmt)
%option Empty { EmptyStmt } =>
%option Block { BlockOfStmt } =>
    [ statements : Statement ]
%option ExprCall { ExprCallStmt } =>
    expr : Expression
// ASSERT 2nd Expression is VarName or FieldCall or ArrayIndex
%option Assignment { AssignmentStmt } =>
    expr : Expression,
    dest : Expression
// ASSERT Statement is VarInit (1D array) or ArrayInit (2D, 3D, ... arrays)
// Though added, ArrayInit for 2D, 3D, ... is not parsed
%option ArrayInit { ArrayInitStmt } =>
    init_expr : Expression,
    type_info : Statement,
    alloc_size : Expression,
    # is_dynamic : bool
%option VarInit { VarInitStmt } =>
    init_expr : Expression,

```

```

    # var_name : std::string, type_name : std::string
// ASSERT Statement is VarInit
%option MakeStruct { MakeStructStmt } =>
    [ inits : Statement ],
    # struct_name : std::string
%option MakeEnum { MakeEnumStmt } =>
    [ enum_variants : std::string ],
    # enum_name : std::string
%option FnArgs { FnArgsStmt } =>
    default_expr : Expression,
    [ array_info : std::pair <bool, int> ],
    # type_name : std::string,
    is_inout : bool,
    var_name : std::string,
    is_array : bool
// ASSERT 1st Statement is Block,
// ASSERT 2nd Statement is FnArgs
%option MakeFunction { MakeFunctionStmt } =>
    fn_body : Statement,
    [ params : Statement ],
    # fn_name : std::string, ret_type_name : std::string
// ASSERT Statement is MakeStruct
%option MakeProperty { MakePropertyStmt } =>
    [ opts : Statement ],
    # prop_name : std::string
// ASSERT Statement is Block
%option ElifCondition { ElifConditionStmt } =>
    elif_cond : Expression,
    elif_body : Statement
// ASSERT 1st Statement is Block,

```

```

// ASSERT 2nd Statement is Block,
// ASSERT 3rd Statement is ElifCondition
%option Condition { ConditionStmt } =>
    if_cond : Expression,
    if_body : Statement,
    else_body : Statement,
    [ elif_stmt : Statement ]
// ASSERT 2nd Statement is Block
%option ForLoop { ForLoopStmt } =>
    init_part : Statement,
    check : Expression,
    final_expr : Statement,
    forloop_body : Statement
// ASSERT Statement is Block
%option WhileLoop { WhileLoopStmt } =>
    cond : Expression,
    whileloop_body : Statement
%option Return { ReturnStmt } =>
    return_expr : Expression
%option Break { BreakStmt } =>
%option Continue { ContinueStmt } =>
%option CaseClause { CaseClauseStmt } =>
    case_body : Statement,
    case_expr : Expression,
    # is_wildcard : bool
%option Match { MatchStmt } =>
    matched_expr : Expression,
    [ cases : Statement ]

```