A Programming Language for Coding Competitions

MIHAI SPINEI, University of Twente, The Netherlands

The most popular languages for online and on-site coding competitions are C++, Java, and Python. While C++ and Java offer exceptional efficiency in handling algorithms and data structures, they require more verbose code. Python, conversely, allows for concise code with high-level features such as list comprehensions and compound literals but trades off memory efficiency and speed. In this paper, we introduce a new programming language that integrates the strengths of these languages, specifically tailored to allow participants of such competitions to translate their solutions into code easily. We present a prototype compiler for this language and evaluate its performance and effectiveness based on execution time, memory utilization, and code conciseness using a series of contest problems.

Additional Key Words and Phrases: Programming Languages, Compiler Construction, Coding Competitions

1 INTRODUCTION

Programming contests and competitions are events where participants develop solutions for a list of algorithmic problems set by the organizers. While the ranking strategies vary from contest to contest, the most common metrics are:

- The number of problems the participant solved correctly
- The time taken to develop each solution
- The number of attempts the participant needed to arrive at a correct solution

Each attempt at a solution is verified against a list of tests for correctness, speed, and memory efficiency. A submission is only accepted once all these requirements are met. As a result, both the efficiency and speed of the development process, as well as the efficiency and speed of the resulting program, are valued by participants.

Therefore, when choosing a language, participants have two options. They can either opt for a high-level language with simple, easy-to-write syntax, such as Python, at the expense of runtime speed, or they can choose a low-level language, such as C++, which guarantees top performance at the expense of a more verbose syntax, resulting in slower development.

In this research project, we propose to define and develop a language specifically designed for programming contests. This language will offer high-level features and easy-to-write syntax without sacrificing speed and efficiency.

We define the following two goals for this project:

- **Goal 1:** Define the syntax of the language, along with additional features, aimed at simplifying the process of writing solution programs for coding competitions;
- **Goal 2:** Implement a cross-compiler from our defined language to a low-level language to analyze and validate the use of our language in this context.

To achieve these goals, we will start by analyzing how the syntax of existing languages is utilized in coding contests, identifying areas for improvement, and adapting our syntax to address these inefficiencies. We will then define additional language features that would be useful in this context. We will discuss the design used to develop the prototype compiler and its limitations. Finally, we will compare existing correct solutions for competitive problems with solutions developed in our language, analyzing both in terms of code conciseness and performance compared to the developed prototype.

As a result of this research, we aim to answer the following research questions:

- **RQ1:** How can a programming language be defined to facilitate competitive coding and problem-solving effectively and easily?
- **RQ2:** How can a compiler for such a language be designed and implemented in order to maintain efficiency and speed?

Through this research, we aim to provide an exact specification of the language, including the motivations for the design choices. Additionally, we aim to develop a prototype compiler for the language and provide an analysis of its efficiency, particularly measuring the performance overhead imposed by using a layer of abstraction compared to coding directly in the target language.

2 DOMAIN ANALYSIS

Answering the first research question requires analyzing how current contest participants utilize features of their preferred programming languages. In Table 1 are presented the language statistics of a previously most popular online coding contest, Google Code Jam[1]. We can see that the participants have a clear preference for C++, followed by Python and Java. We choose to focus on these three languages for further analysis.

Language	Nr. competitors	Advanced next round
C++	20062	18322
Python	8550	7599
Java	2908	2522
PyPy (Python)	1155	1094
C	539	401
C#	316	283
JavaScript	284	246
Kotlin	191	173
Go	155	140
Rust	139	131

Table 1. Google Code Jam 2022 Qualification Round

[1] https://codingcompetitionsonair.withgoogle.com/

TScIT 41, July 5, 2024, Enschede, The Netherlands

To better understand which language features are most commonly utilized and identify possible areas of improvement, we analyzed submissions from recent online coding contests. Our observations on how participants currently use certain language features will guide our design of the proposed language. Specifically, we analyzed submissions made on the online coding contest platforms Codeforces [2] and AtCoder [3].

To process and identify trends, we collected a random sample of 20 recent submissions over recently posted contests and problems from Codeforces and AtCoder respectively, focusing on solutions written in C++, Java, and Python. We gathered submissions from experienced participants (rating above 3000) as well as beginner/intermediate participants (rating below 2000). Each submission was analyzed to extract patterns in code structure, use of language features, and common coding practices. This involved both automated work (using `comm` [4] to find common lines between submissions) and manual inspection to ensure the accuracy and relevance of our findings.

We discuss the identified trends in the following subsections.

2.1 Code Templates

The most common trend among C++ submissions is the use of preprocessor macros, type definitions, and template functions to create a code template. These templates help participants make their code more concise and easier to write. Although templates vary from person to person, the following are some of the most commonly utilized shortcuts found in over 50% of the selected submissions:

```
#define all(x) (x).begin(),(x).end()
#define rep(i, s, e) \
for(int i = (s); i < (e); ++i)
#define rrep(i, s, e) \
for(int i = (s); i > (e); --i)
#define each(i, a) for(auto &i : (a))
using ll = long long;
using pi = pair<int,int>;
using vi = vector<ll>;
template <class T>
using vc = vector<T>;
```

These macros and type definitions significantly reduce the verbosity of code. For instance, the macro `all(x)` allows writing `sort(all(xs))` instead of `sort(xs.begin(), xs.end())` to sort a vector `xs`. Similarly, type definitions like `using pi = pair<int,int>;` and `using vc = vector<T>;` simplify complex data structures, reducing `vector<map<pair<int, int>, bool>>` to `vc<map<pi, bool>>`.

2.2 Literals and Ranges

In contrast, Python submissions lack explicit code templates and instead rely on native support for collection literals, ranges, and slices. Python's ability to express complex operations succinctly is a significant advantage. Features such as list comprehensions, dictionary comprehensions, and generator expressions allow participants to write powerful one-liners that would require more verbose code in other languages.

Original verbose code

```
nums = input().split()
nums = [int(x) for x in nums]
nums.sort(reverse=True)
for num in nums: print(num)
```

Using literals, ranges, higher order functions

```
for num in sorted(map(int, input().split()))[::-1]:
    print(num)
```

3 PROPOSED LANGUAGE

3.1 Motivation

In the process of designing the language, we strive to achieve the following set of goals:

- Concise and simple syntax for common language features
- Similarity to popular languages used for contests
- Simple management and handling of data for algorithms

The language needs to provide users with a simple interface for working with basic data structures such as lists, sets, and maps/dictionaries, as these are commonly used in coding contest solutions, as discussed previously. More complex structures, such as binary trees, graphs, and heaps, are less common and can be implemented using the existing data structures. Therefore, native language support for them is not a main priority.

The language is intended to be statically typed, meaning each variable in the program is assigned a single type at compilation, such as `int, float, bool, char`, and types cannot be changed during the runtime of the program. This decision is motivated by the higher bug proneness, decreased performance, and increased complexity associated with dynamic typing [5]. To keep the code concise, the language should allow programmers to omit type definitions for most variables, relying instead on a type inference algorithm to deduce them. This approach should also apply to collection types, reducing the need for preprocessor macros in the language, as seen in C++.

The language should provide common control flow constructs, similar to those in other languages, but also allow for flexibility in their usage to maintain conciseness and readability.

3.2 General Description

The following section will provide a description of the language named Slash, designed to meet the set list of goals and answer **RQ1**.

The most simple statement of the language is variable declaration. Variables are required to be initialized at declaration.

Variables

```
    <name> : <type> = <value>
    <name> := <value>
```

[2] https://codeforces.com/
[3] https://atcoder.jp/
[4] https://man7.org/linux/man-pages/man1/comm.1.html

Functions follow a similar syntax for declaration, except with the additional syntax for specifying the arguments preceded by a backslash. The syntax for calling functions is similar to most imperative languages, with a set of parenthesis and a comma separated list of values passed in as arguments.

Functions

```
// function definition
<fname> \ <arg1> <arg2> : <type> = <result>
// function call
<fname>(<arg1>, <arg2>)
```

Structure types have the same definition as functions, except instead of returning a certain expression, their result is a scope. The members of the structure are the variables and functions declared directly inside the body of the scope, as well as the arguments passed to the function. Therefore, the function defining a structure type serves both as a declaration and a constructor for that type.

Structures

```
    // function definition
    f \ x := { return (x + 1) }
    f(1) == 2
    // structure definition
    g \ x := { h \ y := x + y }
    g(1).h(2) == 3
    // structure definition
    Pair \ x y := {}
    Pair(1, 2).y == 2
```

In the example above, the function `f` returns the increment of the argument, while the function `g` returns a structure containing `h` as a member function and `Pair` returns a structure with two member variables `x` and `y`.

Compound types such as lists, maps and sets can be defined using literals:

Literals for compound types

```
numbers : list[int] = [1, 2, 3]
fruits : set[str] = #["Apple", "Pear", "Plum"]
capitals : map[str, str] = #{
    "France" : "Paris",
    "UK" : "London"
}
```

Such compound type variables will be accessed by reference and stored in heap memory of the program, as opposed to the primitive types which will be passed by value and stored in stack memory, similarly to how C++ handles compound types such as std::vector, std::set, std::map.

Working with collections

```
a := [1, 2, 3]
println(a.size) // 3
a.add(1)
println(a.size) // 4
println(a@0, a@1, a@2, a@3) // 1 2 3 1
```

Control flow constructs such as `if`, `while`, and `for` are used similarly to other languages but also allow for expressive and flexible usage to maintain conciseness and readability.

Control Flow

```
while (a < 10): a = a + 1
if a % 2 == 0 then println("Even")
    else println("Odd")

for i in 1..10: println(i)
```

Control flow constructs like `if` and `while` can be used as expressions. An `if` expression evaluates to the value in its `then` branch if the condition is true, or the value in its `else` branch if the condition is false.

A `while` expression normally evaluates to the value in its `else` branch when the loop finishes. However, if a `break` statement with a value is used inside the loop, the expression evaluates to that value instead and the `else` branch is skipped.

'Expressive' control flow

```
// finds the index of x in the list ls
// otherwise is -1
i := 0
index := while i < ls.size: {
    if ls@i == x then break(i)
        else { i = i + 1 }
} else -1
```

For-loops can be used to iterate over collections. In the case of lists and sets, the identifier value goes over the values stored in the collection. In the case of a map, the identifier value goes over the set of keys used to access the value in the map.

Collection iteration

```
for i in [1, 2, 3]: print(i)
// 123

for k in #{"One" : 1,"Two" : 2,"Three" : 3}:
    print(k)

// OneTwoThree
```

3.3 Further features

In this section we will describe additional features of the language that would be useful in the context of contest problem solving, but are not a crucial part of the language implementation.

3.3.1 Test case parsing. Because most competitive coding problems require the programmer to parse in the input of the test case for the problem before applying the algorithm, an important feature of the language should be providing a simple way for the programmer to define parsing complex inputs. To achieve this the following syntax was defined for parsing input:

Parsing input

```
name, age, height := input \ string int float
n := input \ int
xs := input \ list(n, int)
```

In this case, input represents the standard input of the program, while the identifiers after the symbol \ represent the parsers that will sequentially run on the input to produce the values of the respective types. This can be later extended to allow the programmer to define the implementation of their own parsers as a function taking in the input and producing a value of the resulting type.

3.3.2 Test case generation. A common practice for debugging solutions to such coding problems is to generate arbitrary test cases and verify properties of the solution. This is done manually by the programmer, usually by writing a separate program to generate the test case inputs. Since in our language the parsed types are clearly stated by the parsers ran on the input, this provides the opportunity to define a test case generator which would replace the normal standard input, and instead provide randomly generated values of the respective types. This feature could be further extended to allow the addition of constrains on the input which would verify if the generated test case is valid.

Test case generation

```
n := input \ int
#constraint ( n <= 100 )
xs := input \ list(n, int)
for x in xs: {
    #constraint (0 <= x && x <= 1e6)
}
#assert brute_force(xs) == greedy(xs)
```

4 COMPILER PROTOTYPE

In this section we discuss how a compiler for the defined language can be designed, as well as describe the prototype implemented along with this research. This serves as our answer to **RQ2**.

In order to validate the efficacy of the language in coding competitions we set the goal to design and develop a minimal compiler that could be used to solve most types of competition problems. The compiler will generate C++ code and rely on the GNU GCC compiler to produce executable files according to the C++20 standard. The generated C++ source file can then be used as a submission to online contest judges, which using the predefined test cases verify that the solution implemented fits the time and memory restrictions. This also allows for the comparison between solutions written in other languages using the same environment and test cases.

4.1 Parsing

The parser for the language was implemented using Tree-sitter – a parser generator and incremental parsing library. It allows to easily specify the grammar rules of the language in the form of a DSL. It then generates either LR(1) or GLR parsers and compiles them to a Shared Object file which can be imported into our compiler to handle parsing input files. The parser generates an Abstract Syntax Tree object, which is then used for type inference and code generation.

A reduced version of the grammar of the language is available in appendix A.

4.2 Type inference algorithm

I order to provide an easy and fast development experience for the user, specifying most types in the language is not required. The compiler will deduce the type of symbols and expressions based on their utilization in the program. To do so, the compiler implements a type inference algorithm inspired by the Hindley-Milner type system (HM) [2] [4] It utilizes Robinson's Unification Algorithm [6] to sequentially unify and deduce the types of the symbols in the program. The HM type system functions by defining a context or environment Γ consisting of a list of typings of the form $e : \tau$ (meaning expression e is of type τ), and a set of inference rules which combine typings to deduce new typings. For example the following is the rule for function application:

$$\frac{\Gamma \vdash f : \tau' \to \tau \quad \Gamma \vdash x : \tau'}{\Gamma \vdash f(x) : \tau}$$

The HM system distinguishes between monotypes – types that designate an exact type, and polytypes or type-schemes – types that contain at least one 'for-all' quantifier (meaning their type is flexible to function as various monotypes, but the general structure of their type is fixed). An expression like `[1, 2, 3]` would have the monotype `list[int]`, while the expression `[]` would have the polytype $\forall\alpha.$`list[`α`]` because it can be used to initialize lists of any type.

The type inference algorithm [3] starts by initializing the type of all expressions in the Abstract Syntax Tree (AST) to a free variable. Then a bottom up pass is performed through the AST during which certain types are unified based on their relation. For example, upon reaching the node `x + 1` in the AST, the compiler will try to unify the types of `x` and `1`. In the case that the underlying types cannot be unified, the result type will be an `Error` type, which will be relied to the user through an error message.

For the purposes of our language, user defined constructs in the language need to have a monotype after the type pass is performed, meaning their type signature should be fixed. This means their type should be either a type constant such as: `int, bool, char`; or a parametric type, such as: `list[int], map[char, int]`. The code generation step of the compiler will fail in case it encounters a user defined symbol with a polytype (for example a function which allows any type of input as parameter) [5]. Language constructs such as operators on the other hand are defined as polytypes, meaning that their type depends on the type of their arguments. For example the plus operator `+` has the polytype $\forall\alpha.\alpha \to \alpha \to \alpha$, while the equality operator `==` has the polytype $\forall\alpha.\alpha \to \alpha \to$ `bool`. This allows the compiler to reuse operators for different types, while ensuring that the type pass will result in a correct deduction of monotypes for expressions. Upon reaching a member access operation in the AST, for example `pair.first`, the type inference algorithm requires the

[5]This restriction is imposed to simplify both the type inference and code generation steps of the compiler because of limited time and resources available for the project

left operand's type to be either parametric constructor (such as `list`, `map`) or define a structure. This ensures that the type inference able to deduce the type of the resulting expression from the name of the member accessed. In case the type of the left operand does not fit the requirements, the type inference algorithm may not continue and throws an error suggesting the user to specify the type of the left operand manually. Special functions of the programming language, such as: `print, input, return, break` , may not even have an exact polytype, since they can allow for a varied number of arguments, therefore need to be handled separately in the type pass.

An additional step to be taken when unifying the types of two expressions is an occurrence check. For example in the code snippet below 1, when the type inference reaches the expression `a.add(a)` it tries to unify the types $\forall\alpha.$`list[`α`]` and α, which leads to the substitution $\{\alpha \rightarrow$ `list[`α`]`$\}$ being applied recursively which in turn does not terminate. As a result, when unifying two expressions, we must also make sure that the free variable being substituted does not occur in the type it is substituted with. In the case it does, the unification algorithm fails with an `Error` type notifying the user where the substitution failed.

Recursive list

```
a := []
a.add(a)
```

4.3 Code generation

The code generation step is done using a top-down approach. Each syntactical scope of the source language is represented through a code generation node which is responsible for processing the variable definitions, function definitions, and expressions inside. Each node manages declaration, definition and execution code separately, only combining them in the last step of the generation process. The AST generated at this step including the type information could be utilized as a high-level Intermediate Representation and transformed into more low level representations such as LLVM IR (similarly to how Python AST with type information can be translated [1]), but for the scope of our prototype compiler this step only includes generation of C++ code.

Most expressions can be easily translated to their C++ counterparts recursively. The following describes the generated C++ code from the respective code in the source language, where `T` is the recursive translation function.

Source code	Generated C++ code
`a op b`	`T(a) op T(b)`
`if c then a else b`	`T(c) ? T(a) : T(b)`
`{ s }`	`({ T(s); })`
`f(a, b)`	`T(f) (T(a), T(b))`
`input(a, b)`	`cin>>T(a)>>T(b);`
`println(a, b)`	`cout<<T(a)<<T(b)<<endl;`
`while(c): b`	`while(T(c)) { T(b); }`
`for i in c: b`	`for(auto& i:T(c)) {T(b)}`

In the case the while loop has a non-unit result type, the while loop needs to be broken down into multiple statements in C++. Therefore the following code is generated instead:

'Expressive' while-loop

```
// Source code
r : R = while (c): a else b

// C++ code
R r = ({
    R _res = {0};
    while(true) {
        if (!T(c)) {
            _res = T(b);
            break;
        }
        T(a);
    };
    _res;
});
```

This makes use of the GNU GCC compiler "Statement Expression" [6], which allows including statement blocks inside of expression, without the need to extract the block into a separate function. A similar result could be achieved using an inline lambda function, which would be called immediately, but would lead to more cluttered code.

4.3.1 Memory model of generated code. The defined language lacks the required tools to allow the user to manage the memory of the program, therefore it is the responsibility of the compiler to ensure that the generated code accesses and manages memory correctly. This imposes restrictions on what the user can express in code such that the compiler is able to translate it into valid C++ code. For example, in the original language description the user is allowed to nest structure definitions in each other, while the prototype compiler will warn the user that this is disallowed in the current version. Additionally, as a consequence of not having native language constructs for memory management, defining structures which possibly contain members of their own type (such as nodes of a binary tree) is not as trivial as in other languages. A possible workaround is having a global list of values of the recursive type, and instead of using references or pointers between such objects, instances of the type could reference each other by index in the global list.

C++ Binary Tree

```
struct BN {
    int value;
    struct BN* l;
    struct BN* r;
};
```

Binary Tree

```
BN \ value := {
    l := -1
    r := -1
}

all_nodes:list[BN]=[]
```

5 EVALUATION

In this section we will evaluate the resulting language and compiler, both in terms of conciseness and efficiency. To evaluate the language and the prototype compiler example solutions were developed for a set of online coding problems. For this purpose we used Project Euler

[6] https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Statement-Exprs.html

Archive [7] problems and a set of problems from Codeforces contests[8]. As part of testing a total of 10 Project Euler Challenges were solved, as well as a set of 5 Codeforces problems. In this section we will provide some examples which showcase the difference between solutions written in our language and example solutions in python.

Solution for Project Euler Problem 5 Slash

```
nums := 1..20
r := 1
gcd \ x y := {
        if x == 0 then return(y)
        if y == 0 then return(x)
        return (gcd(y%x, x))
}
lcm \ x y := (x * y) / gcd(x, y)
for x in nums: r = lcm(r, x)
println(r)
```

Solution Project Euler Problem 5 Python

```
nums = range(1, 21)
r = 1
def gcd(x, y):
    if x == 0: return y
    if y == 0: return x
    return gcd(y%x, x)
def lcm(x, y): return (x * y // gcd(x, y))
for x in nums: r = lcm(r, x)
print(r)
```

The previous two code snippets contain solutions in our defined language and in Python. Both solutions have a similar amount of tokens and lines of code. For the given problem execution time of both programs is virtually the same since the size of the list `nums` is negligible.

Codeforces Round 952 Problem G Solution Python

```
M = int(1e9+7)
t = int(input())
for _ in range(t):
    l, r, k = map(int, input().split())
    print((pow(9//k+1,r,M)-pow(9//k+1,l,M)+M)%M)
```

Codeforces Round 952 Problem G Solution Slash

```
M := 1000000007
t := 0
input(t)
pow \ b n := {
    if n == 0 then return(1)
    if n % 2 == 1 then return((pow(b,n-1)*b)%M)
    else {
        c := pow(b, n / 2) % M
        return ((c * c)%M)
    }
}
for ts in 1..t:{
    l := 0
    r := 0
    k := 0
    input(l,r,k)
    println((pow(9/k+1,r)-pow(9/k+1,l)+M)%M)
}
```

The first solution in Python was developed by the problem author [9] as the official example solution. The second solution was developed as an adaptation of the intended Python solution into our language. The major difference between the provided solutions is the presence of a binary exponentiation function in the standard Python library, meaning we do not have to implement it in the Python version. Both solutions have a runtime complexity of $O(T * log(R))$, and constant memory usage. Problem constraints specify that test cases have the following restrictions: $1 \leq t \leq 10^4$, $0 \leq l < r \leq 10^9$. Because of the relatively small constraint on t, the execution time difference between the Python solution and our solution is not as considerable, but we can still conclude that our solution has a better performance.

Performance tests were done using Intel(R) Core(TM) i3-8100 CPU @ 3.60GHz on Codeforces Polygon Judge System. Each submission was ran once on the set of tests for each problem, taking a max aggregate over all test for execution time and memory usage (rounded to nearest 100 KB). Because of the isolated remote environment of each test the affects of noise on the results are expected to be minimal. Results can be seen in tables: 2, 3, 4.

	Max. Time	Max. Memory	Nr. Lines
Slash (compiled to C++)	109 ms	100 KB	18
Python 3	180 ms	<100 KB	5
PyPy 3-64	180 ms	5000 KB	5

Table 2. Codeforces Round 952 Problem G Sumbissions

A similar test was done for Problem A from Round 200 [10] and Problem G1 from Round 849 [11]. (Solutions for both available in the appendix B.2)

	Max. Time	Max. Memory	Nr. Lines
Slash (compiled to C++)	280 ms	<100 KB	13
Python 3	342 ms	8200 KB	7
PyPy 3-64	436 ms	7000 KB	7

Table 3. Codeforces Round 200 Problem A Submissions

	Max. Time	Max. Memory	Nr. Lines
Slash (compiled to C++)	360 ms	600 KB	22
C++ (official solution)	100 ms	7800 KB	44
Python 3	203 ms	26800 KB	14
PyPy 3-64	171 ms	26400 KB	14

Table 4. Codeforces Round 849 Problem G1 Submissions

[7] https://projecteuler.net/archives
[8] https://codeforces.com/contests
[9] https://codeforces.com/blog/entry/129620
[10] https://codeforces.com/contest/344/problem/A
[11] https://codeforces.com/contest/1791

In this case we can see a similar performance difference compared to the previous problem. Problem A from Round 200 relies on parsing a large amount of lines ($n < 1e5$ lines of input) and a simple $O(N)$ traversal of the input to generate the result. For problem G1 on the other hand the Python solutions takes the upper hand in execution time. Comparing our solution to the native solution written in C++ the most notable difference is memory management. The native solutions utilizes the same global array of values for all testcases, only requiring one allocation per test, while our solution allocates and populates a new array of values for each new testcase which can lead to up to 1000 allocations per test.

5.1 User Experience

As a combined evaluation of both the language design and the prototype compiler, a coding contest participant on the online platform Codeforces utilized our developed language during Round 954 [12]. They were introduced to the language shortly prior to the round. During the round they managed to solve the first proposed problem (problem A) in the first 13 minutes of the contest, and the second problem (problem B) in the proceeding 20 minutes.

During the testing phase of the prototype a set of issues regarding the design and implementation of the prototype became apparent. Parsing input requires more steps when compared to languages like Python in which parsing a line of input usually takes one line of code. Using control flow constructs as expressions can lead to unexpected parse trees, which commonly require the addition of extra parenthesis or braces to ensure the program is parsed correctly. Additionally, in the case the compilation step from our source language to C++ succeeds, while compilation of the generated C++ code fails, tracing the exact cause of the compilation error is not trivial, as it requires the user to understand both the generated C++ code, and have an understanding of how the compiler functions. As such, the prototype should guarantee the correctness of the generated code, or produce a useful error for the user.

5.2 Limitations

The prototype compiler and language is limited in the following ways:

- The compiler does not check for memory safety, meaning that the user can easily create memory leaks or access memory out of bounds.
- The compiler does not verify left value expressions, meaning that the user can easily assign to a non-assignable expression.
- The compiler does not very the correctness of the generated code, meaning that the user can easily write code that does not compile.
- The compiler does not provide a way to debug the generated code, meaning that the user has to rely on the error messages provided by the GNU GCC compiler.
- The compiler does not provide useful error messages when parsing, type inference or compilation fails.
- The language does not coerce types, meaning that the user has to manually cast types when needed.
- The language does not provide a way to define operators or hashing functions for custom types (which are required for set and map collections).
- The language does not provide some additional control flow features, such as `continue` and `switch` statements.
- The language does not provide a way to define custom parsers for input.
- The language does not allow for collection unpacking.
- The language does not have an extensive standard library, meaning that the user has to implement most of the common utilities themselves.

These limitations are a result of the limited time and resources available for the project, and were left to be addressed in future iterations of the language and compiler as they were not crucial for the initial evaluation and research of the language.

6 CONCLUSION

In this paper, we discussed the domain of coding contests and analyzed the strengths and weaknesses of currently popular languages. We identified possible areas of improvement, and proposed a language designed to solve some of the short-comings. We designed its syntax to be simple and understandable to participants already familiar with other imperative programming languages. We designed and developed a prototype compiler for the proposed language, which showcases the important features of the language and serves as a proof-of-concept. Additionally, we highlighted potential areas of improvement, either in terms of further language features or extensions to the prototype. Finally, we evaluated both the syntax of the language, as well as the produced code from the compiler, using actual coding competition problems and similar coding challenges. The layer of abstraction created by our compiler provides a concise way to write solutions to coding problems, but this comes at a slight cost in performance when compared to native C++ solutions, and a more complicated debugging process for the user. Exploring the idea of designing a programming language which is specifically intended to be used in coding competitions proved to result in a viable language that achieved the goals set forward at the beginning.

7 FURTHER RESEARCH

Because of the limited time spent of evaluating the compiler of the language we cannot draw concrete conclusions about its performance compared to other language. Further extensive testing and analysis of performance would provide more exact findings and could be used as a stronger base for comparison. Currently, the proposed language is in a very early stage of development and design. Further iteration on the syntax of the language could be done by testing it using various types of contest domain problems (for example dynamic programming, graph theory, geometry) and extending the lacking parts of the syntax. Such extensions could include allowing the user to define generic functions and types, which allow for easier handling of collections.

As mentioned in the description of the compiler, the produced AST structure could be compiled to other low-level representations. The addition of a compilation target to LLVM IR would allow for

[12] https://codeforces.com/contest/1986

further testing of the performance of the language, irrespective of the chosen back-end.

Exploring additional language extensions, such as native memory management, garbage collection, or multi threading could open up further pathways for research and improvement of the language.

8 ACKNOWLEDGEMENTS

I would like to thank Peter Lammich for his guidance and support throughout the research process. Additionally, I would like to thank KRKevin [13] for testing the prototype and providing valuable feedback regarding the usability of the language.

REFERENCES

[1] Mateusz Bysiek, Mohamed Wahib, Aleksandr Drozd, and Satoshi Matsuoka. 2018. Towards portable high performance in Python: Transpilation, high-level IR, code transformations and compiler directives. 2018, HPC-165 (2018).

[2] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

[3] Adam Gundry, Conor Mcbride, and James Mckinna. 2010. Type inference in context. (09 2010). https://doi.org/10.1145/1863597.1863608

[4] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[5] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 155–165. https://doi.org/10.1145/2635868.2635922

[6] J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (jan 1965), 23–41. https://doi.org/10.1145/321250.321253

A SIMPLIFIED LANGUAGE GRAMMAR

```
program :: = (statement)*
statement :: = (
    variable_definition
    | function_definition
    | expression
) '\n'

variable_definition :: =
    identifier ':' type? '=' expression
function_definition :: =
    identifier '\' argument* '=' expression
argument :: =
    identifier | '(' identifier ':' type ')'
expression :: =
    expression op expression
    | uop expression
    | 'if' expression 'then' expression
        ('else' expression)?
    | 'while' expression ':' expression
        ('else' expression)?
    | 'for' identifier 'in' expression
        ':' expression
    | '{' statement* '}'
    | identifier '(' expression* ')'
    | identifier
    | expression '.' identifier
    | literal
literal :: =
    Number
    | String
    | Char
    | Boolean
    | Float
    -- List
    | '[' expression* ']'
    -- Set
    | '#[' expression ']'
    -- Map
    | '#{' (expression ':' expression) * '}'
    -- Tuple
    | '#(' expression* ')'
```

[13] https://codeforces.com/profile/KRKevin

B SAMPLE SOLUTIONS

B.1 Project Euler

Problem 1

```
r := 0
for i in 1..999:
    if i % 3 == 0 or i % 5 == 0 then r = r + i
println(r)
```

Problem 2

```
L := 4000000
solve \ a b := if b > L then 0
    else b + solve(a+2*b, 2*a+3*b)
print(solve(1, 2))
```

Problem 3

```
number := 600851475143

is_prime \ x := {
    i := 2
    return( while i * i < x: {
        if x % i == 0 then break(false)
        i = i + 1
    } else true)
}
r := 0
t := 2
while t*t < number: {
    if (number % t == 0) then {
        if is_prime(number/t) and r < number/t then {
            r = number / t
        }
        if is_prime(t) and r < t then { r = t }
    }
    t = t + 1
}
println(r)
```

B.2 Codeforces

Codeforces Round 200 Problem A Solution Python

```
n = int(input())
s = [input() for _ in range(n)]
c = 1
for i in range(1,n):
    if s[i] != s[i-1]:
        c += 1
print(c)
```

Codeforces Round 200 Problem A Solution Slash

```
n := 0
input(n)
s := []

k := 0
for i in 1..n: {
    input(k)
    s.add(k)
}

c := 1
for i in 1..n-1: {
    if not (s@i == s@(i-1)) then c = c + 1
}

println(c)
```

Codeforces Round 849 Problem G1 Solution Python

```
t = int(input())

for _ in range(t):
    n, c = map(int, input().split())
    tp = list(map(int, input().split()))

    for i in range(n):
        tp[i] += i + 1

    tp.sort()

    i = 0
    ans = 0
    while i < len(tp) and tp[i] <= c:
        ans += 1
        c -= tp[i]
        i += 1

    print(ans)
```

Codeforces Round 849 Problem G1 Solution Slash

```
t := 0
input(t)

for i in 1..t: {
    n := 0
    c := 0
    input(n, c)
    tp := []
    for j in 1..n: {
        a := 0
        input(a)
        tp.add(a + j)
    }

    tp.sort()

    it := 0
    ans := 0
    while it < tp.size and tp@it <= c: {
        ans = ans + 1
        c = c - tp@it
        it = it + 1
    }

    println(ans)
}
```