UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 2568

Design of a strongly-typed programming language

Petar Mihalj

Zagreb, May 2021.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 12 March 2021

MASTER THESIS ASSIGNMENT No. 2568

Student:	**Petar Mihalj (0036497900)**
Study:	Computing
Profile:	Computer Science
Mentor:	asst. prof. Ante Đerek

Title: **Design of a strongly-typed programming language**

Description:

Programming languages play a central role in the development of software. However, there is no silver bullet programming language, which would be efficient, easy to use, portable and consistent. The goal of this graduate thesis is to design a new programming language with the emphasis on how strong typing can address common drawbacks. The language specification should be described in a formal form. Design should be followed-up by a careful analysis of some aspects of the language, for example to prove the desirable properties of the type system. The final goal of the thesis is to implement the tool chain, which primarily consists of a compiler, but can include other tools, such as language servers and formatters. The thesis should be accompanied with the source code of all developed software. All references should be clearly cited. Any assistance received should be clearly acknowledged.

Submission date: 28 June 2021

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

DIPLOMSKI ZADATAK br. 2568

Pristupnik:	**Petar Mihalj (0036497900)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	doc. dr. sc. Ante Đerek

Zadatak: **Oblikovanje strogo tipiziranog programskog jezika**

Opis zadatka:

Programski jezici centralna su karika u razvoju programske potpore. Ipak, ne postoji programski jezik najbolji za sve primjene, u isto vrijeme efikasan, jednostavan, prenosiv i konzistentan. Cilj ovog diplomskog rada dizajnirati je novi programski jezik, s naglaskom na prednosti strogih tipova u rješavanju čestih problema. Specifikacija jezika treba biti opisana u formalnom obliku. Nakon dizajna jezika potrebno je provesti pažljivu analizu nekih svojstava jezika, na primjer pokazati poželjna svojstva sustava tipova. Posljednji cilj rada implementacija je skupa alata, kojeg primarno čini prevoditelj, ali koji može uključivati i druge alate, poput jezičnih poslužitelja (language servers) i formatera koda. Rad treba uključivati izvorni kod razvijenih alata. Sve reference trebaju biti jasno citirane. Sva primljena pomoć treba biti jasno obznanjena.

Rok za predaju rada: 28. lipnja 2021.

I thank everybody...

CONTENTS

1. Introduction

The programming language developed as a part of this thesis is called AGT. The name AGT stands for an unfortunate fact that the names of all precious stones are taken by other programming languages, hence "All Gems are Taken".

The AGT programming language is a statically and strongly typed language, with a highly expressive type system. The type system is used for three main purposes:

1. types determine the memory footprint of values
2. types allow for polymorphic behaviour of operators and functions on all *concrete types* (built-in or struct types)
3. type system lets the programmer perform compile-time computation

The language also allows for implementation of object lifetime constructs, such as those seen in languages as C++ or Rust; object creation, copying and destruction.

The reference AGT compiler (AGTC) produces executable binaries only. This is in sharp contrast to some other languages, which can produce object files, to later be linked into executables for a particular runtime, or used to augument the runtime environment itself (kernel modules, for example). This decision was made primarely because the goal of the thesis was to study the language from the application programmer's perspective. This goal implies that AGT is running on a rich runtime environement, where heap memory management and standard I/O are available. Future updates to AGT specification are meant to allow AGT to be compiled into linkable files compatible with various platform specific ABIs.

The compiler frontend is implemented using Python3 programming language, while the backend uses LLVM compiler infrastructure (Lattner und Adve, 2004). Allowing for some simplification, AGT is first compiled into LLVM intermediate representation (LLVM-IR), and than is turned into executable by a chain of tools for compiling LLVM-IR and linking the resulting object files into executables.

1.1. Example program

Let us first check out an example AGT program in listing 1.1:

Listing 1.1: Example AGT program

```
1  fn outnl(i){
2      out(i);
3      out('\n');
4  }
5
6  fn mul(a, b) -> a{
7      return a*b;
8  }
9
10 fn main()->i32{
11     let a = in<i32>();
12     let b = in<i8>();
13
14     let c = mul(a, cast<i32>(b));
15     outnl(a);
16     outnl(b);
17     outnl(c);
18 }
```

Every AGT program has to have a function definition named main, which returns a 32-bit-wide integer, and has no parameters. Some functions, like `outnl`, don't return anything. It is important to note that you can't specify that a function returns `void` like in C for example.

A `let <x> = <y>;` construct is an *initialization assignment statement*. This roughly translates to an allocation of memory on the stack (which can be refered to as the *location* of <x> from now on), and copying the value of expression <y> to the location of identifier <x>.

Note that the function call `in` (line 11) is *parametrized* by a *type argument* `i32`. Type arguments are used to pass types, but not values. They are different than *value arguments*, which carry value too, along with a type. Note that *passing* of type parameters is purely a compile-time concept; types can't be refered to during runtime.

The function `in` is a builtin, and the `i32` is used to signal the compiler that we want an instance of this function which returns an `i32`, instead of a `bool`, for example.

Of course, user-defined functions can also have *type parameters*, along with *value parameters*; we will get to these later.

Notice that we used the `cast<i32>` builtin function to convert `b` to `i32` before passing it to function `mul`. If we didn't do that, the compiler would signal that it can't compute the expression `a*b`, since multiplication is predefined only for integer types of same size. Apart from the `i8` and `i32` we used in this example, `i16` and `i64` are also available.

Also, line 6 is terminated by a `-> a`. This is a return type specification; it states that this function will return whatever is type of a.

You might have noticed a certain feature while inspecting the code; namely the abscence of explicitly stated types in the signature of function definition `mul`. While some *dynamically typed* languages (ex. Python) allow for object of any type to be passsed to the function, AGT behaves quite differently.

Every time AGTC (AGT compiler) encounters a function call, it tries to infer a *function type*. Function type is inferred from function definitions. Consequently, you can think of the definitions in source code more as *templates* for synthesis of actual code, than representation of code. These definitions lack types, and can't be considered in isolation. In this example, the call of function `mul` causes the compiler to try to infer a function type.

Parameters of this `mul` function definition (a, b) can be fit with objects of any type; it is after this fitting that the compiler will determine that passed types can or can't be used (due to their incompatibility with a function definition body, for example).

It is important to emphasize that one function definition can be a source of many function types. The `outnl` (output with newline) function is a prime example of this behaviour. It is called three times, with argument of types `i32`, `i8` and `i32`. Thus, the compiler has to infer two function types, one which can be fed with an `i8`, and one with `i32`. The success of this inference corresponds with compilers ability to infer functionality of body of the given function definition.

In this case, inference of function definition body will be possible if both function calls on lines 2 and 3 can be inferred. Since `out` function is a built-in for both i32 and i8, line 2 won't be problematic for inference. Line 3 will be inferred correctly if AGTC suceeds to resolve function `out` which takes in an argument of type `char`. Since this is also a built-in, inference of function types for both version of `mul` suceeds.

The property of AGT **function calls** which can make them refer to different function types, when called with different sets of arguments, is called *polymorphism*. Some sources label polymorphic the very **function definitions** which, when called upon,

can result in such behaviour. We will use these conventions interchangably, since the definition itself is non-rigorous.

Function definition of `outnl` introduces a special type of polymorphism, *parametric polymorphism*. This polymorphic behaviour is said to be present if the function definition can be a source for potentially unlimited number of types, as long as those types are compatible with function definition body.

Python programming language exibits a superficially similar behaviour, but it's function body compatibility checks occur at runtime. The programming style which utilizes this language capability is often called "Duck Typing" (Fred L. Drake Jr.).

1.2. Paper organization

This master thesis will gradually introduce the reader to AGT.

Chapter 1 has already introduced the reader to basic syntax and semantics, using a simple example and emphasizing the surface-level properties of type system and inference.

The following chapter 2 will lay out and justify author's language design decisions. The discussion includes many diverse topics, ranging from syntactical desisions, to suptle semantic onces.

After the surface-level exploration of the language, chapter 3 will provide an in-depth description of AGT. This chapter will have many references to the AGTC reference implementation, rather than to a separate formal specification. Since AGT is still not a finalized product, formal specification does not exist, and it's functionality is determined by a reference compiler. We will mostly focus on the type system, since it is unorthodox and is the main contribution of the author. Author will justify the decisions which had been made on all levels of AGT design process.

Chapter 4 will supplement the reader with plenty of examples of AGT code. These will further deepen reader's understanding of AGT. The examples describe impementation of common programming paradigms, constructs, patterns in AGT, with a detailed explaination of more esoteric parts of the source code.

In chapter 5 we will address the issues AGT has both on definition and implementation levels, and discuss the various features that can be improved or added to the language.

Chapter 6 will conclude the thesis and describe future work the author will undertake in order to improve the language.

2. Design decisions

In this chapter we will describe and justify the main design decisions concerning AGT, weighing their pros and cons against one another.

2.1. Syntax

Main inspiration for AGT's syntax have been well known languages C and C++. Since most programmers are familiar with these, it is only logical not to diverge from their syntax, whenever possible. Of course, AGT is way more implicit in typing than both C and C++, that is, types have to be specified only when necessary (for example, when parametrizing functions by types). Implicit typing allows the programmer to focus on logic instead of typing redundant type annotations (such as in C), or keywords that signal the compiler to perform automatic type inference (`auto` in C++).

On the other hand, the abscence of explicit type specification can make the program more unreadable, due to the lack of hints these types provide.

The author would argue that such problems are rooted in more suptle problems, such as poorly designed function interface. If the function definition is named `square`, it is only natural to assume that a supplied argument can be multiplied by itself. Function definitions that are more convoluted than square can explicitly state their requirements in the beginning of function definition body, thus giving the programmer a hint for their usage.

The Rust Programming Language has had a significant impact on AGT's syntactical features too. Mainly, it's use of clear function definition syntax and `let` statement syntax is embedded deeply into AGT.

The Zen of Python (Peters, 2004) is a set of guidelines for writing clean, *pythonic* code. One of these guidelines - *Explicit is better than implicit*, has had a huge impact on AGT. Starting from clean function definition syntax, to explicitly named operator (*dunder*) methods (`__add__`, `__init__`, `...`), AGT places explicitness over surface-level practicality. The author would argue that explicit code is more readable,

maintainable and safe over long term, which is a big win over a small expense of writing a few more characters. Object creation is also as explicit as can be, whether on the stack or the heap.

Another guideline from The Zen of Python is *There should be one and preferably only one obvious way to do it*. AGT tries to inhibit programmer's choice in a fundamentally non-limiting way, for example, by having only one syntax for initializing objects. This does not decrease the language's expressiveness, but helps avoid confusion and errors.

2.2. Type System

Before going into decisions regarding the AGT type system, we will first state common definitions of what type system actually is, and what we refer to when discussing it.

There are many definitions, but the one which captures the essence most precisely is the following:

A type system is a logical system comprising a set of rules that assigns a property called a type to the various constructs of a computer program, such as variables, expressions, functions or modules ... (Wikipedia contributors, 2021).

AGT's type system serves multiple purposes. One is to allow compile time computation. The most important one is to ensure the correctness of programs, both on compile time and run time.

AGT's type system is *static*, that is, all values have types which can be, and are, determined at compile time. Furthermore, all polymorphic behaviour is resolved in compile time, unlike in C++, where the programmer can use *dynamic polymorphism* - a run-time dynamic lookup of functions. This makes AGT even more type-safe, by preventing common mistakes which happen when utilizing dynamic polymorphism. The author argues that object oriented principles which heavily rely on dynamic polymorphism (such as inheritance), can mostly be eliminated by better program design.

Another feature of AGT's type system is that it is *strongly typed*. Most definitions consider a language to be strongly typed if it rarely or never implicitly converts types, which leads to more compile-time errors, but at the same time provides more safety. AGT never converts types implicitly, which can, to someone coming from languages such as C, appear tiresome to deal with. On the other hand, explicitly stating type conversions will have greater benefits for the programmer in the long run.

Type inference engine which supports AGT's type system can appear difficult to get used to, but in fact is, atleast on the theoretical level, rather simple. There are no complex inference rules; every time an inference has to be done, all the inputs

required to perform the process must be available. For example, function's return type is inferred from the return type expression. This type expression is evaluated just before the first runtime statement in function definition body, which ensures that the function's return type is known in case the function is recursive (directly or not).

2.3. Memory management

AGT aims to be as performant as possible, which rules out having a dedicated *garbage collector*, present in languages such as Java and Python. AGT's storage is split into two main parts, stack allocated and heap allocated storage.

The stack storage is used to support function execution, and is managed implicitly - allocated on function call, deallocated on function return. While this storage type is highly efficient, it prevents the use of a number of programming patterns. To remedy this situation, languages such as C use heap storage.

Heap storage is an on-demand storage. Having this storage can allow the programmer to define objects which outlive the functions they are defined in, using pointers as references to this storage, rather than implicitly refering to storage location through the use of names. While heap storage allows for many useful design patterns, it can bring many difficult-to-diagnose problems to the table, namely:

- Memory leaks - programmer forgets to free unused memory
- Double free - programmer free's unused memory multiple times
- Dangling pointers - programmer uses a pointer which refers to already freed memory

These problems have been a major source of headache for programmers over the years and can't be solved without certain effort on programmer's part. A typical attempt to remedy these is the use of smart pointers in languages such as C++. We will explore and implement this approach in chapter 4.

2.4. Compiler implementation

The reader should be aware that AGT is not yet a complete project, but rather an experiment, a work in progress. This state of affairs had author decide to use Python3 as a compiler implementation language. Rapid prototyping features of Python, along with numerous easy-to-use and available libraries make it a sound choice for the implementation language. Since Python is slow compared to compiled languages, a reasonable direction

to go in, after AGT matures, would be to switch from Python to one of the compiled languages, such as C++.

3. The AGT Programming Language

AGT compilation process consists of 4 main phases:

1. Lexical analysis - tokenization of input source code text
2. Syntactical analysis - rule-based grouping of tokens into an *abstract syntax tree*
3. Semantical analysis - transforming the syntax tree into a *semantical tree*
4. Code generation - inference of the `main` and code generation

The process is successful if all the steps suceede.

In this chapter we will describe each phase in great detail, contrasting the newly defined terms against their use in other languages.

3.1. Lexical analysis

The AGT Lexical analysis phase is a standard regular expression parsing process. Lexical analyzer is fed a stream of characters, which get groupted into tokens, according to regular expressions and ambiguity resolution rules.

Lexical analyer tool used for the AGTC is PLY - Python Lex-Yacc (Beazley, a). Python Lex-Yacc is a tool which tries to bring the functionality of well know lexical and syntactical analyzers Lex and Yacc into Python. The main difference is that PLY does not *generate* the concrete lexer and parser. Instead, PLY user defines the rules fed to PLY by specifying programming constructs like functions that correspond to these rules. PLY then inspects these constructs using reflective capabilities of Python, and acts according to these rules when performing lexical or syntactical analysis.

Let us check out the main lexing module of AGTC, which specifies rules, states and preceedence that PLY uses to perform lexical analysis for AGT. Some parts of the lexing module are left out, which is indicated with `#...` in source code.

Listing 3.1: heyy

```
class Lexer():
    states = (
        ('mlc', 'exclusive'), ('strchar', 'exclusive'),
    )
    #...
    tokens = (
        'INTL', 'BOOLL', 'ID',
        'IF', 'ELSE', 'BREAK', 'FOR', 'WHILE', 'RETURN',
        #...
    )
    t_ADD = r'\+'
    t_SUB = r'-'
    t_MUL = r'\*'
    #...
    reserved = {
        'type': 'TYPE',
        'fn': 'FN',
        'let': 'LET',
        #...
    }
    def t_ID(self, t):
        r'[a-zA-Z_][a-zA-Z_0-9]*'
        t.type = self.reserved.get(t.value, 'ID')
        return t

    def t_INTL(self, t):
        r'(\d+)(|i8|i16|i32|i64)'
        return t
    #...
```

This lexer module has 3 states: `main` (implicit), `mlc` (for multiline comments) and `strchar` (for *char arrays* or char literals).

Rules for simple operators are given in a string form, in a form of `t_<TOKEN> = <REGEX>`.

Rules for more tokens which are defined with more complex regular expressions, like int literals or identifiers, are defined using functions. Those functions can also alter the behaviour of the lexer when such a token is parsed. For example, whenever

an identifier is parsed, the function `t_ID` looks up whether the identifier is in a list of reserved words, and if it is, changes the token's type. This neat trick greatly simplifies regular expressions, and is taken from official PLY documentation (Beazley, b).

PLY documentation states that priority is given to regular expressions defined by functions, in the order in which they are given. After function-based rule definitions, priority is given to string defined rules (the ones with greater regex length first).

We employ the *state* feature of PLY lexing module, which allows us to change the lexer's behaviour when it encounters, for example, a beginning of a multiline comment. In that particular case, lexer goes into `mlc` state. When in `mlc` state, lexer parses and caches every symbol until it gets to the end of the comment. Both type of comments, `/*multiline comment*/` and `//singleline comment` are discarded after being parsed.

3.1.1. Literals

String and character literals are given in double and single quotes, respectively. They can include escaped characters: (\0, \n, \t, \", \'). Character literal must consist of exactly one character or escaped character.

Integer literals are characterized by both the value and the size of integer containing this value. Possible integer sizes are 8, 16, 32 and 64.

Example integer literals are:

- `5` (implicitly i32)
- `5i32`
- `516`
- `5i8`
- `5i64`
- `324243` (implicitly i32)

Boolean literals are:

- `true`
- `True`
- `false`
- `False`

3.1.2. Identifiers

Identifiers consist of a letter or an underscore, followed by any number of letters, underscores or digits. Reserved words are excluded from identifiers.

Reserved words are:

- 'fn', 'struct',
- 'let', 'type',
- 'if', 'else', 'break', 'for', 'while', 'return',

3.1.3. Operators

We will now list the operator families, without discussing their semantics in detail, because

- Built-in versions of these operators will be listed in later chapters, and the reader can intuitively reason about which ones are available.
- User-defined versions of these operators can adhere to a number of different semantics (which is not recommended, but is possible).
- Operators can be applied on types during compile time (to produce other types). This discussing is better to be left to later chapters.

Apart from the operators themselves, we will list their special names. These names will be used to translate operator applications to function calls. For example, when `+` is applied to value `3` and `5` (`3+5`), compiler will insert a call to `__add__(3,5)` instead. This method derived from operator's special name is an example of what we call a *dunder* or *magic* method (according to Python folklore). *Dunder* comes from double-underscore which both preceeds and follows the special name.

Arithmetic operators	**Comparison operators**	**Boolean operators**
+ (add)	== (eq)	& (and)
- (sub)	!= (ne)	| (or)
* (mul)	>! (gt)	∼(not)
/ (div)	<! (lt)	
% (mod)	>= (ge)	
	<= (le)	

Note that, even though we call the last group the *boolean* operators, those operators can be evaluated on operands which are not booleans, as long as the corresponding

dunder functions are defined in the program.

The same goes for arithmetic and comparison operators; a programmer can use builtins functions for all integer types, but can define additional dunder functions for `rectangle` objects, for example.

3.1.4. Special operators

Special operators can't be user-defined, and have a fixed meaning:

- *Address of operator* (@) can be used on any L-value (explained in later chapters), and evaluates to a pointer to that value. For example, if `i` is an `i8`, then `@i` is a pointer to `i8`. We can also denote this pointer as `@i8` - exactly the way it can be used in the context of a type. For example, if you want to check whether a function argument is of `@i8` type (to get polymorphic behaviour), you would start this function with a line: `type _ = enable_if<arg == @i8>;`.
- Dereference operator (`!`) can be used on any pointer value, and evaluates to a value which is being pointed to. For example, if `p` is an `@i8`, then `p!` is a value of type `i8`. This operator can also be used in the context of a type. For example, if you want to check whether a function argument is a pointer (to whatever type), and capture the pointed-to type, you would start this function with a line: `type pointed_to = enable_if_resolve<arg!>;`. A new special type expression `enable_if_resolve<...>` exibits similar behaviour as `enable_if<...>`, in sense that it lets you create polymorphic behaviour. This one will succeed if the argument is successfully evaluated (it doesn't have to evaluate to `i1`). If it succeeds, it will also act as if it wasn't there. In this case,
- Index (bracket) operator must be used on a pointer, and be given any integrer type as an argument. It will offset the pointer depending on the underlying type. For example, if `p` is a `char` pointer, p[5] will evaluate to a pointer with a location $lp + 5 \cdot s$, where lp is `p`'s value and s is size of char type.

Precedence rules of operators will be stated in section 2.2. An interested reader will find more details concrening the implementation in the source code supplied with this paper.

3.2. Syntactic analysis

Syntactic analysis (also called parsing in some sources) is *"a process of analyzing a string of symbols... conforming to the rules of a formal grammar"* (**?**).

Formal rules of AGT grammar are analyzed by an LALR parser supplied with PLY package, which tries to immitate functionality of well know Yacc tool. Analogous to the lexing part of PLY package, parser will also use reflection to generate parsing tables, which are used to run the stack machine.

Tokens resulting from lexical analysis are fed into a stack machine which, depending on the state of the top of the stack and next token in the list, either pushes the next token on top of the stack (*shift*) or converts number of tokens on the top of the stack to another token (*reduce*). A LALR parser can, without ambiguities, resolve only certain kinds of context-free grammars. This problem is taken care of by defining a conforming grammar and by using precedence rules.

Listing 3.2: Precedence rules

```
precedence = (
    ('left', 'COMMA'),
    ('left', 'AND'),
    ('left', 'OR'),
    ('left', 'ADD', 'SUB'),
    ('left', 'MUL', 'DIV', 'MOD'),
    ('left', 'LE', 'GE', 'LT', 'GT', 'EQ', 'NE'),
    ('left', 'DOT', 'DEREF', 'NOT'),
    ('right', 'ADDRESS'),
    ('left', 'LPAREN', 'LBRACE', 'LANGLE'),
)
```

Precedence rules are similar to those found in C programming language. They specify the associativity (left or right) and priority (from lower to higher).

We won't go into formal discussion of the syntactic analysis, an interested reader can find details in a standard reference book for compiler theory (Aho u. a., 2007).

AGT contains a rule which goes as follows:

```
InitStatement : LET Expression ASSIGNMENT Expression
                SEMICOLON
```

Once the five tokens on the right side of the colon character in this rule end up being on top of the stack, parser can remove them, and substitute a new `InitStatement`

in their stead. Parser has implicitly created a tree node of type `InitStatement`, which captures both expressions that this statement is made of (assigned-to expression, assigned-from expression).

This process finishes when stack machine ends up having the starting symbol on top of the stack, and no more tokens to parse. The output of a syntactic analysis phase is an abstract syntax tree - a tree structure which denotes applications of grammar rules to tokens. An example syntax tree, which results from the statement `let n = in<i32>();` is given in listing 3.3.

Listing 3.3: AST resulting from `let n = in<i32>();`

```
1  Statement(
2  -  statement = InitStatement(
3  -  -  nameexpr = Expression(
4  -  -  -  expr = IdExpression(
5  -  -  -  -  id = str(n)
6  -  -  -  )
7  -  -  )
8  -  -  expr = Expression(
9  -  -  -  expr = UnaryExpression(
10 -  -  -  -  expr = ParenthesesCallExpression(
11 -  -  -  -  -  expr = Expression(
12 -  -  -  -  -  -  expr = UnaryExpression(
13 -  -  -  -  -  -  -  expr = AngleCallExpression(
14 -  -  -  -  -  -  -  -  expr = Expression(
15 -  -  -  -  -  -  -  -  -  expr = IdExpression(
16 -  -  -  -  -  -  -  -  -  -  id = str(in)
17 -  -  -  -  -  -  -  -  -  )
18 -  -  -  -  -  -  -  -  )
19 -  -  -  -  -  -  -  -  expr_list = [
20 -  -  -  -  -  -  -  -  -  Expression(
21 -  -  -  -  -  -  -  -  -  -  expr = IdExpression(
22 -  -  -  -  -  -  -  -  -  -  -  id = str(i32)
23 -  -  -  -  -  -  -  -  -  -  )
24 -  -  -  -  -  -  -  -  -  )
25 -  -  -  -  -  -  -  -  ]
26 -  -  -  -  -  -  -  )
27 -  -  -  -  -  -  )
28 -  -  -  -  -  )
```

```
29 - - - - - expr_list = [
30 - - - - - ]
31 - - - - )
32 - - - )
33 - - )
34 - )
35 )
```

Concerning the implementation, grammar rules are given in a different way than the one standard for PLY. PLY uses functions by default, while AGTC uses classes which have to adhere to certain structure. These class definitions are turned into functions (using reflective programming) and finally dynamically injected into syntactical parser class definition. A reader who is interested in details can consult the source code.

An example of an implementation of a rule for binary expressions is given in the listing 3.4.

Listing 3.4: Binary expression rule implementation

```
1  class BinaryExpression(ParserRule):
2      """BinaryExpression : Expression ADD Expression
3                          | Expression SUB Expression
4                          | Expression MUL Expression
5                          | Expression DIV Expression
6                          | Expression MOD Expression
7                          | Expression LE Expression
8                          | Expression GE Expression
9                          | Expression LT Expression
10                         | Expression GT Expression
11                         | Expression EQ Expression
12                         | Expression NE Expression
13                         | Expression AND Expression
14                         | Expression OR Expression
15     """
16
17     def __init__(self, r):
18         self.left = r[0]
19         self.op = r[1]
20         self.right = r[2]
```

3.3. Semantic analysis

The need for semantic analysis is caused by a fact that AGT's syntax is highly contextual, that is, expressions that appear in one place can have a radically different meaning than same expressions which appear in another place.

For example, in the code listing 3.5, expression `a` is used in two distinct contexts.

Listing 3.5: Contextuality of expression `a`

```
1 fn zero(a) -> a{
2     return a-a;
3 }
4
5 fn main()->i32{
6     out(zero(3));
7     return 0;
8 }
```

In the ending of the first line, `a` is a type expression, that is, a compiler is required to produce a type which the function returns. In the second line, `a` is a value expression, which means compiler is required to produce a value of that expression, rather than a type. We haven't yet specified what means to produce either a type or a value, but these implementation details will be dealt with in section 3.4.

A part of semantic analyzer is presented in listing 3.6. In line 3, semantic analyzer checks whether it is currently in function (or struct) definition, and acts accordingly. In case of function definiton, analyzer initializes type context (line 5), starts parsing the expression on the right side (line 6), and strips type context at the end (line 7). In the case of struct definitions, analogous process takes place.

Listing 3.6: a

```
1  @add_method_parse_semantics(pr.InitStatement)
2  def _(self, se: SE):
3      if not se.in_func:
4          name = self.nameexpr.expr.id
5          se.add(SS.TYPE_EXPR)
6          a = self.expr.parse_semantics(se)
7          se.pop()
8          return MemberDeclarationStatement(name, a)
9      else:
10         name = self.nameexpr.expr.id
```

```
11            se.add(SS.VALUE_EXPR)
12            a = self.expr.parse_semantics(se)
13            se.pop()
14            return InitStatement(name, a)
```

Another example of contextual nature of syntactical elements is the let <x> = <y>; statement. When stated in function definition body, this statement results in an allocation of new memory on the stack, and copying of value of `<y>` to location of `<x>`. On the other hand, when stated in struct definition body, this indicates that struct will have a member named `<x>` of type `<y>`. Thus, `<y>` will be interpreted in a type context.

Another task which is being done in this part is substitution of operators with *dunder* functions or structs. In value context `5+a` gets substituted with a function call `__add__(5,a)`.

On the other hand, if analyzer is currently in type context, `i8+a` gets substituted with *type angle expression* `__add__<i8, a>`. If `a` is of type `i3`, than this type angle expression will evaluate to `i11`. AGT provides built-in operators on all integer types, like the + you have just seen, for all other arithmetic and comparison operators. Comparison operators == and `!=` are defined on all types, even user-defines ones, and work as one would expect.

Note that the programmer can specify all nonnegative-size integers in type context, not just `i8, 16, i32, i64`, for example `i3, i0, i342123`. This allows for powerful compile-time computation examples, some of which will be discussed in chapter 4.

3.4. Code generation

Code generation is the most complex phase of AGT compilation process. Before we go heads first into it's description, we will lay down some definitions and motivate them. Code generation includes both type inference and LLVM intermediate representation generation.

AGT's type inference engine (type engine, TE) can be modeled as a pure mathematical function, because it's main goal is to evaluate *type requests*. This *pureity* of evaluation means that repeated evaluations always yield the same result. In other words, TE can be thought of as a block box which provides two evaluation functions, *concrete type request* and *function type request*.

3.4.1. Concrete type request

The first function which TE supplies is *Concrete type request* (CTR). This function maps a name and a list of concrete types (*type arguments*) into a new concrete type. For example,

1. CTR("char", []) -> CharType
2. CTR("i32", []) -> IntType(32)
3. CTR("rectangle", [i32]) -> RectangleType(with `i32` side lengths)

How does the TE provide this mapping?

In the first part of a CTR request, TE collects *candidates*. It can collect the candidates by two means.

First way to collect candidates is by *concrete type generators*. They can be invoked with CTR arguments and decide whether they can provide a type, for example, a bool type generator can check whether the name is equal to "bool" and whether there are no supplied type arguments; if both conditions are satisfied, it can provide the concrete type.

The other way is to iterate over user-supplied struct definitions. For every struct definition whose name matches the supplied name, and whose number of type parameters matches the number of supplied type arguments, the inference is being done. Inference which is done by using struct definitions consists of evaluating the body of struct definition, which might need recursive calls to type engine itself. We ensure that these recursions are taken care of; for example, recursive structure definitions will fail.

When both of these ways are done (without errors) there has to be **exactly one** candidate type, otherwise, we have an ambiguity, which is an error. This candidate type is returned as a result of a concrete type request.

3.4.2. Function type request

The second inference function is a *Function type request* (FTR). This function maps a name and **two** lists of concrete types (*type arguments* and *value arguments*) into a new concrete type. For example,

1. FTR("main", [], []) -> FunctionType
2. FTR("fibonacci",[], [i32]) -> FunctionType

3. FTR("power", [fast], [i32, i32]) -> FunctionType

4. FTR("power", [slow], [i32, i32]) -> FunctionType

A FunctionType is a rather complex type; it needs to have all information needed for code synthesis, including LLVM code and return type. In our concrete implementation, LLVM code is not directly contained in the function type, but can be referenced via program-unique *mangled function name*.

By default, every function call is preceeded by copying of its arguments. However, some built-in functions, like copy functions of pointers, should not copy their arguments (because of infinite recursive loops). Only built-in functions can ignore this copying behaviour, and the flag which indicates whether to do so is also a part of a FunctionType.

Let us take a look at the code snippet in listing 3.7, which demonstrates a sort of polymorphic behaviour.

Listing 3.7: enable_if example

```
1  struct slow{;}
2  struct fast{;}
3
4  fn power<T>(b, p){
5      type _ = enable_if<T==slow>;
6      ...
7  }
8
9  fn power<T>(b, p){
10     type _ = enable_if<T==fast>;
11     ...
12 }
```

Suppose the type engine is invoked to compute an answer to a FTR("power", [slow], [i32, i32]). Since "power" is not in built-in functions, TE will only consider user-supplied function definitions.

When TE starts to infer a type from the definition at line 4, it will immidiatelly encounter a special construct `enable_if<...>` at line 5. This construct is the main driver of polymorphism in AGT. AGTC will first evaluate it's argument, `T==slow`. Since this is evaluated in the type context, operator == will evaluate to type `i0` (falsehood) or `i1` (truth), depending on whether the operands are the same. Since its argument represents truth, `enable_if` will act as if didn't exist; it will evaluate to the type

of its argument. Obviously, this type will then be assigned to a name on the left side. Since this name is a *discard token* (_), the type won't be assigned, but rather discarded.

The user has achieved polymorphic selection using enable_if. He signalled the compiler that this definition of power function fits programmers intentions.

On the other hand, consider the definition of `power` at line 9. In this line, `enable_if` argument will evaluate to `i0` (false in type context). This will cause the TE to abandon this function as a potential candidate for type resolution. Note that this behaviour is not a *critical* error (the one which gets reported and crashes AGTC), but rather a mechanism for polymorphic selection.

3.4.3. Lifetime semantics

Lifetime semantics is a broad term which rougly encompasses the following aspects of objects in programming languages:

1. Objects exist in a specific context, which is dependant on their

   lexical environment (the scope)

   runtime (the function invocation)

   For example, parameters of the function exist and can be referenced throughout the function body, but stop existing when that specific invocation of the function terminates.

2. The context in which objects exist is called their *lifetime*

3. Object's lifetime starts with *initialization* (*construction* in some sources).

   Having precise control over construction can allow programmer to restrict usage of the object or set up invariants

4. Object's lifetime ends with *destruction*. Having precise control over destruction allows programmer to deconstruct complex inner workings of the object, without relying on programmer to do it manually. For example, objects can *release* their resources when they stop existing.

5. Transferring or duplicating object's contents to other memory location can also be controlled, via `copy operation`. This allows us to, for example, deep copy the memory array, instead of shallow copying the pointer and thus incidentally creating unwanted memory aliases.

Various other semantics can be implemented on top of lifetime model, we will check out an example called `Ownership semantics` in the last section. Now we will describe how lifetime semantics fit into definition of AGT. Before we get into the details though, we have to throughly define types of values.

Definition 1 (L-value)**.** A value expression is said to a L-value, if the value it evaluates to **can** be referred to from other context.

Definition 2 (R-value)**.** A value expression is said to a R-value, if it is not L-value, that is if the value it evaluates to **can't** be referred to from other context.

For example, every statement `let <v> = ...` creates a new memory location, so the expressions `<v>` following it and refering to same identifier `<v>` are L-values.

On the other hand, function call expressions such as `gcd(3,2)` are rvalues; their result (function return value) can't be refered to from any other place, apart from this one.

The distinction between L-values and R-values is crucial when reasoning about copy operations. Since R-values can't be refered to from any other context, instead of copying their value, we can `memory copy` their value. The crucial distinction is that `copying` involves calling the `copy` function (which might be very expensive, for example for vectors), and *memory copying* does not copies only the literal memory contents.

Definition 3 (Initializer)**.** An initializer is a function whose name is `__init__`, and which has atleast one value parameter; a pointer to type being initialized. Other value parameters are objects which are needed for initialization.

Definition 4 (Copy function)**.** An copy function is a function whose name is `__copy__`, and which has two value parameters; first being a pointer to memory being copied to, and second one being a pointer to object copied from.

Definition 5 (Destructor)**.** An destructor is a function whose name is `__dest__`, and which has only one value parameter; a pointer to object being destroyed.

Check out the example of lifetime semantics in listing 3.8 and output in listing 3.9.

Listing 3.8: Lifetime semantics

```
1 fn outnl(i){
2         out(i);
3         out('\n');
```

```
4  }
5
6  struct B{
7          let member = i32;
8  }
9
10 fn __init__(bp, i){
11         type _ = enable_if<bp==@B>;
12         type _ = enable_if<i==i32>;
13
14         outnl("Init B started");
15         bp!.member = i;
16         outnl("Init B finished");
17 }
18
19 fn __copy__(bp1, bp2){
20         type _ = enable_if<bp1==@B>;
21         type _ = enable_if<bp2==@B>;
22
23         outnl("Copy B started");
24         bp1!.member = bp2!.member;
25         outnl("Copy B finished");
26 }
27
28 fn __dest__(bp){
29         type _ = enable_if<bp==@B>;
30
31         outnl("Dest B started");
32         __dest__(@(bp!.member));
33         outnl("Dest B finished");
34 }
35
36 fn main() -> i32{
37     let b = object<B>(5);
38     outnl(b.member);
39
40         return 0;
41 }
```

Listing 3.9: Lifetime semantics - output

```
1 Init B started
2 Init B finished
3 5
4 Dest B started
5 Dest B finished
```

You can clearly see that we have used `enable_if` construct in lifetime functions too. This is crucial to prevent multiple candidates for different concrete types.

These were the basics of lifetime semantics, we will introduce more use-cases in chapter 4.

4. AGT program examples

This section of the paper will introduce the reader to AGT by presenting, explaining and contemplating about example programs.

4.1. Parametric polymorphism

Parametric polymorphism is a phenomenon of function calls refering to different concrete functions (or in AGT language - function types) when invoked with different sets of argument types.

In the example in listing 4.1, we calculate n'th fibonacci number recursivly.

Listing 4.1: Fibonacci with recursion

```
1  fn outnl(i){
2      out(i);
3      out('\n');
4  }
5
6  struct is_int<T> -> R{
7      type R = (T==i64 | T==i32 | T==i16 | T==i8);
8  }
9
10 fn fib<T>(n) -> T{
11     type _ = enable_if<is_int<T>>;
12     type _ = enable_if<n == i32>;
13
14     if (n<=2){
15         return cast<T>(1);
16     }
17     else{
18         return fib<T>(n-1)+fib<T>(n-2);
```

```
19        }
20  }
21
22  fn main() -> i32{
23        let t = 12;
24        let f1 = fib<i8>(t);
25        let f2 = fib<i32>(t);
26        outnl(f1);
27        outnl(f2);
28        return 0;
29  }
```

The reader should first note the `outnl` function definition in line 1; this function takes a single argument, which is refered to as parameter named `i`, and this parameter can take any type (since we haven't limited the function definition with enable_if). Of course, if `out` in line 2 can't be called with that type, we will be reported a compilation error. This is an example of parametric polymorphism.

The other example of polymorphic behaviour occurs in definition of function `fib` in line 10. Here we explicitly state that this function is to be considered as a candidate only if the first type argument (which is assigned to type parameter `T`) is an integer type.

Integer selection behaviour is achieved by using *supstituting struct definition*. Supstituting struct definition is a struct definition which has no members, and *returns* a type (ending of line 6). When a struct type is inferred from such definition, it is supstituted with whatever is *returned* from it (`R` here), instead of being interpreted as an aggregate type, as ordinary structs are.

Semantics of `T` represent the integer size in which fibonacci number is to be returned. Similarly, our `fib` function always expects `i32` as a specifier of `n`.

Listing 4.2 shows the output of our program.

Output:

Listing 4.2: Fibonacci with recursion - output

```
1  -112
2  144
```

As expected, `i8` can't hold a value of `144` and we notice an overflow (negative result).

4.2. Compile time computation

We will perform the same computation as in the last section, but now using compile-time computation capabilities of AGT. Listing 4.3 contains our demo.

Listing 4.3: Compile-time Fibonacci

```
1  fn outnl(i){
2      out(i);
3      out('\n');
4  }
5
6  struct fib<T> -> i1{
7      type _ = enable_if<T == i1>;
8  }
9  struct fib<T> -> i1{
10     type _ = enable_if<T == i2>;
11 }
12
13 struct fib<T> -> R{
14     type _ = enable_if<T != i1>;
15     type _ = enable_if<T != i2>;
16     type R = fib<T-i1>+fib<T-i2>;
17 }
18
19 fn main() -> i32{
20     outnl(type_to_value<i100, i32>());
21     outnl(type_to_value<fib<i12>, i32>());
22     outnl(type_to_value<fib<i12>, i8>());
23     return 0;
24 }
```

Check out the line 20; we output the result of a call to `type_to_value<A,B>()`. This is a special builtin function, which will, for any integer type specified as the first type argument (`A`), return that integer's size in a value of type given as `B`. For example, in line 20, we will get an `i32` with value 100 as a return value from `type_to_value` function. Fundamentally, this function allows you to *transfer* your compile-time-computed value (in form of integer types) to runtime.

Having access to `type_to_value` function, all we have to do is compute an integer type corresponding to n'th Fibonacci number. The reader should now study

the *supstituting struct definitions* on lines 6, 9 and 13. It is imporant to note that they are disjoint in the sense that only one is a candidate for type inference (achieved with `enable_if`).

Listing 4.4: Compile-time Fibonacci - output

```
1  100
2  144
3  -112
```

The same overflow problem happens in this example, due to request for a value of type `i8`, which can't hold `144`.

4.3. Heap object allocation

We will now demonstrate object allocation and initialization on the heap.

Listing 4.5: Heap object allocation

```
1  fn outnl(i){
2      out(i);
3      out('\n');
4  }
5
6  struct rect<T>{
7      let a = T;
8      let b = T;
9  }
10
11 fn main() -> i32{
12     let a = in<i32>();
13     let b = in<i32>();
14
15     type irect = rect<i32>;
16     let rect_heap = heap_object<irect>(a,b);
17
18     outnl(rect_heap!.a);
19     outnl(rect_heap!.b);
20     heap_free(rect_heap);
21     outnl(rect_heap!.a);
22     outnl(rect_heap!.b);
```

```
23 }
```

Listing 4.6: Heap object allocation - input

```
1 15
2 16
```

Listing 4.7: Heap object allocation - output

In the listing 4.5, we first introduce a struct definition at line 1. After loading in two `i32` values, we *alias* `rect<i32>` as `irect`, so we can refer to it with this name in the future.

In line 16, we allocate an `irect` on the heap using a special method `heap_object<T>(...)`. Similarly to `object<T>(...)`, this method allocates space for object of type `T` and initializes it with an appropriate `__init__` function. Unlike `object`, this method does allocation on the heap, instead of the stack.

After allocating and displaying value of member `a`, we `heap_free` the allocated memory, and print out the member again. While this is a nondeterministic behaviour, we have kept an output where member is overwritten by different data to demonstrate unsafe nature of access to freed data.

4.4. Raw heap allocation

In the following example, we demonstrate the `heap_alloc<T>(n)` function, which allocates memory for `n` consecutive objects of type `T`, and returns a pointer to beginning of that memory.

Listing 4.8: Raw heap allocation

```
1 fn printpos(ptr, pos){
2     out(ptr[pos]!);
3     out('\n');
4 }
5
6 fn setpos(ptr, pos, val){
7     ptr[pos]! = val;
```

```
 8 }
 9
10 fn main() -> i32{
11     let H = heap_alloc<i32>(3);
12     setpos(H, 1, 123);
13     setpos(H, 2, 456);
14     printpos(H, 0);
15     printpos(H, 1);
16     printpos(H, 2);
17     heap_free(H);
18
19     return 0;
20 }
```

Listing 4.9: Raw heap allocation - output

```
1 0
2 123
3 456
```

Although the stack equivalent of this function is not yet available, there is no reason for it not to exist and can be implemented in AGT when needed.

4.5. A simple string class

The value of a string literal is a pointer to a character array which is allocated on the stack. Thus, you can't, for example, return a `char` pointer which you get from string literal; since the underlying memory will be overwritten.

If we want to use strings as they are used in other programming languages, we should write a struct which can be initialized with a string literal. The initializer should allocate more memory on the heap, which will be copied to from the initial string literal.

Listing 4.10: String class

```
1 fn outnl(i){
2     out(i);
3     out('\n');
4 }
5
```

```
 6 fn len(cptr) -> i32{
 7     type _ = enable_if<cptr == @char>;
 8     let cl = 0;
 9     while (cptr[cl]! !='\0'){
10         cl=cl+1;
11     }
12     return cl;
13 }
14
15 //
16
17 struct string{
18     let arr = @char;
19     let len = i32;
20 }
21
22 fn __init__(sptr, cptr){
23     type _ = enable_if<sptr == @string>;
24     type _ = enable_if<cptr == @char>;
25
26     let l = len(cptr);
27
28     sptr!.len = l;
29     sptr!.arr = heap_alloc<char>(l);
30     while (l>!0){
31         sptr!.arr[l-1]! = cptr[l-1]!;
32         l=l-1;
33     }
34 }
35
36 fn __copy__(sptr1, sptr2){
37     type _ = enable_if<sptr1 == @string>;
38     type _ = enable_if<sptr2 == @string>;
39
40     sptr1!.len = sptr2!.len;
41
42     let l = sptr1!.len;
43
```

```
44      sptr1!.arr = heap_alloc<char>(l);
45      while (l>!0){
46          sptr1!.arr[l-1]! = sptr2!.arr[l-1]!;
47          l=l-1;
48      }
49  }
50
51  fn __dest__(sptr){
52      type _ = enable_if<sptr == @string>;
53
54      heap_free(sptr!.arr);
55  }
56
57  //
58
59  fn get(sptr, i) -> char{
60      type _ = enable_if<sptr == @string>;
61      type _ = enable_if<i == i32>;
62
63      return sptr!.arr[i]!;
64  }
65
66  fn set(sptr, i, c){
67      type _ = enable_if<sptr == @string>;
68      type _ = enable_if<i == i32>;
69      type _ = enable_if<c == char>;
70
71      sptr!.arr[i]! = c;
72  }
73
74  fn out(s){
75      type _ = enable_if<s == string>;
76
77      for(let i=0; i <! s.len; i=i+1;){
78          out(s.arr[i]!);
79      }
80  }
81
```

```
82 fn main() -> i32{
83     let a = object<string>("Hello world!");
84     let b = a;
85
86     outnl(a);
87     outnl(b);
88     outnl(get(@a,1));
89     set(@a, 1, 'E');
90     outnl(a);
91     outnl(b);
92
93     return 0;
94 }
```

Listing 4.11: String class - output

```
1 Hello world!
2 Hello world!
3 e
4 HEllo world!
5 Hello world!
```

4.6. Ownership semantics

Check out the following example:

Listing 4.12: a

```
1  fn outnl(a){
2      out(a);
3      out("\n");
4  }
5
6  struct shared_ptr<T>{
7      let item_ptr = @T;
8      let count = @i32;
9  }
10
11 fn __init__(sptr_ptr, item_ptr){
12     outnl("INIT");
```

```
13      type T = enable_if_resolve<sptr_ptr!.T>;
14      type _ = enable_if<sptr_ptr == @shared_ptr<T>>;
15      type _ = enable_if<item_ptr == @T>;
16
17      sptr_ptr!.item_ptr = item_ptr;
18      sptr_ptr!.count = heap_object<i32>(1);
19  }
20
21  fn __dest__(sptr_ptr){
22      outnl("DEST");
23      type T = enable_if_resolve<sptr_ptr!.T>;
24      type _ = enable_if<sptr_ptr == @shared_ptr<T>>;
25
26      sptr_ptr!.count! = sptr_ptr!.count! - 1;
27      if (sptr_ptr!.count! == 0){
28          outnl("RELEASE");
29          heap_free(sptr_ptr!.item_ptr);
30          heap_free(sptr_ptr!.count);
31      }
32  }
33
34  fn __copy__(sptr_ptr_dest, sptr_ptr_src){
35      outnl("COPY");
36      type T = enable_if_resolve<sptr_ptr_dest!.T>;
37      type _ = enable_if<sptr_ptr_dest == @shared_ptr<T>>;
38      type _ = enable_if<sptr_ptr_src == @shared_ptr<T>>;
39
40      sptr_ptr_src!.count! = sptr_ptr_src!.count! + 1;
41
42      sptr_ptr_dest!.item_ptr = sptr_ptr_src!.item_ptr;
43      sptr_ptr_dest!.count = sptr_ptr_src!.count;
44  }
45
46  fn make_shared(item_ptr) -> shared_ptr<inner>{
47      type inner = enable_if_resolve<item_ptr!>;
48      return object<shared_ptr<inner>>(item_ptr);
49  }
50
```

```
51 // actual use
52
53 fn main() -> i32{
54     let a = make_shared(heap_object<i8>(15i8));
55     outnl("Before B is initialized");
56     outnl(a.count!);
57     outnl(a.item_ptr!);
58     {
59         let b = a;
60         outnl("After B is initialized");
61         outnl(a.count!);
62         outnl(a.item_ptr!);
63         outnl(b.count!);
64         outnl(b.item_ptr!);
65         b.item_ptr! = 13i8;
66         outnl("After B is changed");
67         outnl(a.count!);
68         outnl(a.item_ptr!);
69         outnl(b.count!);
70         outnl(b.item_ptr!);
71     }
72     outnl("After B is destructed");
73     outnl(a.count!);
74     outnl(a.item_ptr!);
75
76     return 0;
77 }
```

Output:

Listing 4.13: a

```
1 INIT
2 Before B is initialized
3 1
4 15
5 COPY
6 After B is initialized
7 2
8 15
```

```
 9  2
10  15
11  After B is changed
12  2
13  13
14  2
15  13
16  DEST
17  After B is destructed
18  1
19  13
20  DEST
21  RELEASE
```

5. Future improvements

Since AGT is a new project, it has not yet matured to be used in complex applications. The author list main features that have to be implemented and issues to be addressed in order to turn AGT into a practical language for widespread use.

5.1. Pass-by-reference

AGT is, by default, a *pass-by-value* language; when a programing construct refers to a parameter from inside the function, it refers to a copy which is localized to that function, and not to an original object which was passed to function on function call. Of course, this copying can be bypassed if the argument is an R-value.

A *pass-by-reference* is a contrasting method to pass-by-value. When a programming construct refers to a parameter which is passed-by-reference, it refers to argument which was passed on function call.

Some language (like C++) support passing-by-reference. Others, like Java, use it by default (atleast it appears so to the programmer).

AGT would greatly benefit from having this feature. The main use case would be lifetime functions like `__init__`. Instead of requiring the first argument to be a pointer, they would allow for references. This would increase the ease-of-use on a syntactical level.

5.2. Syntactic and lexical improvements

The syntax of AGT has evolved rather organically; when a new feature was needed, the syntax was modified accordingly. Even though there were general plans for syntax rules at the beginning of the project, we have evolved it beyond initial expectations, and syntax had to change accordingly.

The same argument stands for lexical improvements; many have been defined in a rather unusual way (`<!`) to simplify the design process.

These improvements are one of the first thing to consider going forward; a clear syntactic and lexic rules will ease the building of a fully mature language later on.

5.3. Multiple source file support and namespaces

The reader may have noticed that we have repeated some common function definitions (`outnl` for example) quite often. This is because AGT doesn't have a multiple source file support. There are few problems which need to be adressed before implementing it.

For first, a common problem with multiple source files is other languages is multiple inclusion and multiple definition problems which occur with it. In AGT this phenomenon would manifest as a presence of multiple candidates for type inference. A quick fix for this problem would be to ignore function definitions which come from the same lexical location (source file). An alternative is to ignore all function definitions which are *the same*, that is, equivalent on either a lexical or a syntactial level.

Secondly, a convention regarding file search paths must be adopted. Although there are many examples of this being done quite sucessfully, implications must be considered throughly, and thus we have decided to postpone the implementation to a later update of AGT.

Finally, multiple source file support, and growth of codebases accompanied by it would require AGT to have some sort of name separation policies. These are often called namespaces. Their implementation would most likely accompany multiple source file support.

5.4. Portability

We haven't discussed the interaction of AGTC with a backend compiler via generation of LLVM-IR. Since the goal of this thesis was the explore of an idea, rather than a fully mature system, we haven't taken portability into account. This means that AGTC likely won't run on systems other than those with Linux kernel and an accompanying set of software (gcc, llvm, Python3), with accompanying libraries (PLY).

Further development of AGT would require us to precisely define which platforms we want to support and whether to split the language into minimal and full implementations (to allow for embedded support). These are challenging decisions, and they have been postponed until further development.

6. Conclusion

The goal of this master thesis was to design a programming language with foundation on a strong type system. Learning from other examples, the author was to select a number of desirable properties, and combine them to produce a functional language.

The developed language, AGT, was shown to be expressive, yet safe. AGT's syntax is particulary strict, which prevents the programmer from making hard-to-find mistakes. The type system and inference process is a novel concept which, at least to the author's knowledge, has not been realized in this way before. AGT's type system allows programmer to use highly expressive polymorphic constructs, which greatly simplifies the program design process, reduces errors, and increases safety.

The author has provided many examples of AGT use, ranging from simple syntactic guides, to implementations of complex design patterns such as smart pointers.

The author can conclude that AGT's design, implementation, and principles have shown to be useful and functional when applied to common tasks an application programmer performs. Thus, the goal of this thesis was sucessfully accomplished.

BIBLIOGRAPHY

[Aho u. a. 2007] AHO, Alfred ; LAM, Monica ; SETHI, Ravi ; ULLMAN, Jeffrey: Compilers Principles, Techniques, Tools. (2007), 01, S. 1009

[Beazley a] BEAZLEY, David: *PLY (Python Lex-Yacc)*. – URL `http://www.dabeaz.com/ply/`

[Beazley b] BEAZLEY, David: *PLY (Python Lex-Yacc) documentation*. – URL `https://ply.readthedocs.io/en/latest/index.html`

[Fred L. Drake Jr.] FRED L. DRAKE JR., others: *Python3 documentation*. – URL `https://docs.python.org/3/glossary.html#term-duck-typing`

[Lattner und Adve 2004] LATTNER, Chris ; ADVE, Vikram: *LLVM: A compilation framework for lifelong program analysis transformation*. 2004

[Peters 2004] PETERS, Tim: *The Zen of Python*. 08 2004. – URL `https://www.python.org/dev/peps/pep-0020/`

[Wikipedia contributors 2021] WIKIPEDIA CONTRIBUTORS: *Type system — Wikipedia, The Free Encyclopedia*. 2021. – URL `https://en.wikipedia.org/w/index.php?title=Type_system&oldid=1024380108`. – [Online; accessed 24-May-2021]

Design of a strongly-typed programming language

Sažetak

Sažetak na hrvatskom jeziku.

Ključne riječi: Ključne riječi, odvojene zarezima.

Title

Abstract

Abstract.

Keywords: Keywords.