I thank my mentor Ante Đerek for continous assistance and help during the process of writing this thesis.

I thank my family and friends, whose support and understanding has meant a great deal to me.

CONTENTS

1. Introduction **1**
1.1. Example program . . . 2
1.2. Paper organization . . . 4

2. Design decisions **6**
2.1. Syntax . . . 6
2.2. Type System . . . 7
2.3. Memory management . . . 8
2.4. Compiler implementation . . . 9

3. The AGT Programming Language **10**
3.1. Lexical analysis . . . 10
3.1.1. Literals . . . 12
3.1.2. Identifiers . . . 13
3.1.3. Operators . . . 13
3.1.4. Special operators . . . 14
3.2. Syntactic analysis . . . 14
3.3. Semantic analysis . . . 18
3.4. Type inference and code generation . . . 19
3.4.1. Concrete type request . . . 20
3.4.2. Function type request . . . 20
3.4.3. Lifetime semantics . . . 22
3.4.4. Code generation . . . 25

4. AGT program examples **28**
4.1. Limiting polymorphic behaviour . . . 28
4.2. Compile time computation . . . 30
4.3. Heap object allocation . . . 31

4.4. Raw heap allocation 32
4.5. A simple string class 33
4.6. Ownership semantics and shared pointers 36

5. Future improvements **41**
5.1. Pass-by-reference 41
5.2. Syntactic and lexical improvements 41
5.3. Multiple source file support and namespaces 42
5.4. Portability 42

6. Conclusion **43**

Bibliography **44**

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 author's motivation for development of this language stems primarily from learning opportunity this development provides.

The AGT programming language is a statically and strongly (Barbara Liskov, 1974) 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 such as C++ (Stroustrup, 2013a); 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 augment the runtime environment itself (kernel modules, for example). This decision was made primarily because the goal of the thesis was to study the language from the application programmer's perspective. This goal requires that AGT is running on a rich runtime environment, 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 version 3.9.5 (but should work on all versions starting from 3.7), 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 then is turned into executable by a chain of tools for compiling LLVM-IR and linking the resulting object files into executables.

The installation and use guide, thesis and the accompanying source code will be available at: `https://github.com/PetarMihalj/AGT/tree/thesis_version`. Note that this version might become outdated, so check out the main repository (`https://github.com/PetarMihalj/AGT`) for the latest version.

1.1. Example program

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

Listing 1.1: Example AGT program

```
fn outnl(i){
    out(i);
    out('\n');
}

fn mul(a, b) -> a{
    return a*b;
}

fn main()->i32{
    let a = in<i32>();
    let b = in<i8>();

    let c = mul(a, cast<i32>(b));
    outnl(a);
    outnl(b);
    outnl(c);
}
```

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. There is no `void` type for functions that don't return a value; just skip specifying the return type to achieve this behaviour.

A `let <x> = <y>;` construct is an *initialization assignment statement*. This roughly translates to an allocation of memory on the stack (which can be referred 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, to functions. They are different than *value arguments*, which carry value too, along with a type. The *passing* of type parameters is purely a compile-time concept; types can't be referred to during runtime.

The function `in` is a built-in, 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>` built-in 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, notice that line 6 is terminated by `-> a`. This is a *return type specification*; it states that this function will return a value with a type being equal to whatever is type of `a`.

You might have noticed a certain feature while inspecting the code; namely the absence of explicitly stated types in the signature of function definition `mul`. While some *dynamically typed* languages (for example Python) allow for objects of any type to be passed 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 either from user-supplied function definitions, or from built-in function type generators (more on the this later). Consequently, you can think of the definitions in source code as *templates* for synthesis of actual code. In other words, for each function definition there might be more than one function type inferred from it.

These definitions lack types, and can't be considered in isolation. In this example, it is the call of function `mul` which causes the compiler to try to infer a function type. Parameters of `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 either 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 depends mainly on the compiler's 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 succeeds to resolve function type `out` which takes in a value argument of type `char`. Since this version of `out` is also a built-in, inference of function types for both versions of `mul` succeeds.

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 interchangeably, 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 exhibits a superficially similar behaviour, but its function body compatibility checks occur at runtime. The programming style which utilizes this language capability is often called *Duck Typing* (Fred L. Drake Jr.) in dynamically typed languages.

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 syntactic decisions, to subtle semantic ones.

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. The 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 AGT code examples. These will further deepen reader's understanding of AGT. The examples describe implementation of common programming paradigms, constructs and patterns in AGT, with a detailed explanation 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 upon 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 positive and negative sides 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 limiting function polymorphism according to types). Implicit typing allows the programmer to focus on logic instead of specifying redundant type annotations (such as in C), or keywords that signal the compiler to perform automatic *type inference*, such as `auto` in C++ (Stroustrup, 2013b). Type inference is a process in which the compiler deduces types of programming constructs, which can be of varying complexity.

On the other hand, the absence 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 subtle problems, such as poorly designed function interfaces. 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 syntactic 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*) functions (`__add__`, `__init__`, `...`). We prefer 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 a common definition of what type system actually is, and what we refer to when discussing it. There are many definitions of type system, but the one which captures the essence most precisely is the following:

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute (Pierce, 2002).

AGT's type system serves multiple purposes. One is to allow for compile time computation. The most important purpose 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 dispatch* using *virtual methods* (Stroustrup, 2013c) — 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 is that it is *strongly typed.* Strongly typed language, in most definitions, is a language with a following property:

whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function.(Barbara Liskov, 1974).

Thus, a language is strongly typed if it rarely or never implicitly converts types - this behaviour elicits more compile-time errors, and thus 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, at least 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).

AGT's main polymorphism driver, `enable_if`, is motivated by a similar C++ construct (Stroustrup, 2013d). This construct is, unlike in C++, integral driver of type system, rather than a clever trick to remove functions from overload resolution.

2.3. Memory management

AGT aims to be as fast 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 ending. While this storage type is highly efficient, it prevents the use of numerous 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 referring 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 frees 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. Syntactic analysis — rule-based grouping of tokens into an *abstract syntax tree*
3. Semantic analysis — transformation of the syntax tree into a *semantic tree*
4. Type inference and code generation — inference of the `main` function type and LLVM-IR generation

The process is successful if all the steps succeed.

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 grouped into tokens, according to regular expressions and ambiguity resolution rules.

Lexical analyzer tool used for the AGTC is PLY — Python Lex-Yacc (Beazley, a). Python Lex-Yacc is a tool which attempts to port the functionality of well know lexical and syntactic analyzers Lex and Yacc into Python. The main difference is that PLY does not *generate* the concrete lexer and parser. Instead, the PLY user defines the rules fed to PLY by specifying programming constructs (functions, for example) 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 syntactic analysis.

Let us check out the main lexing module of AGTC in listing 3.1, which specifies rules, states and precedence that PLY uses to perform lexical analysis for AGT. Some parts of the lexing module are left out, which is indicated by `#...` in source code.

Listing 3.1: Lexing module

```
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 multi-line comments) and `strchar` (for *char array literals* or *char literals*).

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

Rules for tokens which are defined with more complex regular expressions, like integer 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 regular expression 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 multi-line 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 types of comments, `/*multi-line comment*/` and `//single-line comment` are discarded after being parsed.

3.1.1. Literals

Char array literals and char 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`
- `5i6`
- `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). The discussion about this feature is better delayed 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 function, derived from operator's special name, is an example of what we call a *dunder* or *magic* function (according to Python folklore). *Dunder* comes from double-underscore which both precedes 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 built-in 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, `arg == @i8` will evaluate to `i1` (truth), only when arg is a pointer to `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. the context of a type. For example, dereference operator applied to type `@i8` will evaluate to `i8`.
- Index (bracket) operator must be used on a pointer, and be given any integer 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 the size of the `char` type.

Precedence rules of operators will be stated in section 2.2. An interested reader will find more details concerning the implementation in the source code supplied as an addition to the thesis.

3.2. Syntactic analysis

Parsing is a process of

analyzing (a string of characters) in order to associate groups of characters with the syntactic units of the underlying grammar." (Dictionary.com authors).

In programming language literature, parsing is often refered to as *syntactic analysis*, *characters* in the above definition are commonly called *tokens*, while *string of characters* is a *stream of tokens*.

Formal rules of AGT grammar are analyzed by an LALR parser supplied with PLY package, which attempts to imitate the functionality of well known tool called Yacc. Analogous to the lexing part of PLY package, the 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', 'OR'),
    ('left', 'AND'),
    ('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 syntax analysis of the statement `let n = in<i32>();` is given in listing 3.3.

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

```
Statement(
- statement = InitStatement(
- - nameexpr = Expression(
- - - expr = IdExpression(
- - - - id = str(n)
- - - )
- - )
- - expr = Expression(
- - - expr = UnaryExpression(
- - - - expr = ParenthesesCallExpression(
- - - - - expr = Expression(
- - - - - - expr = UnaryExpression(
- - - - - - - expr = AngleCallExpression(
- - - - - - - - expr = Expression(
- - - - - - - - - expr = IdExpression(
- - - - - - - - - - id = str(in)
- - - - - - - - - )
- - - - - - - - )
- - - - - - - - expr_list = [
- - - - - - - - - Expression(
- - - - - - - - - - expr = IdExpression(
- - - - - - - - - - - id = str(i32)
- - - - - - - - - - )
- - - - - - - - - )
- - - - - - - - ]
- - - - - - - )
- - - - - - )
- - - - - )
- - - - - expr_list = [
- - - - - ]
- - - - )
- - - )
- - )
- )
)
```

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 syntactic 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

```
class BinaryExpression(ParserRule):
    """BinaryExpression : Expression ADD Expression
                        | Expression SUB Expression
                        | Expression MUL Expression
                        | Expression DIV Expression
                        | Expression MOD Expression
                        | Expression LE Expression
                        | Expression GE Expression
                        | Expression LT Expression
                        | Expression GT Expression
                        | Expression EQ Expression
                        | Expression NE Expression
                        | Expression AND Expression
                        | Expression OR Expression
    """

    def __init__(self, r):
        self.left = r[0]
        self.op = r[1]
        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 the 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 definition, 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: Semantic analysis rule example

```
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
```

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

The part of semantic analyzer we have just seen corresponds to the contextual nature of syntactic elements in 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.

Semantic analysis also handles type expression substitutions 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.

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

3.4. Type inference and code generation

Type inference and code generation is the most complex phase of AGT compilation process. Before we go head-first into its description, we will lay down some definitions and motivate them.

AGT's type inference engine (type engine, TE) can be modeled as a pure mathematical function, because its main goal is to evaluate *type requests*. This *purity* 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 preceded by copying of its arguments. However, some built-in functions, like copy functions of pointers, should not copy their arguments (because of infinite recursion). 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 immediately 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 the programmer's intentions.

On the other hand, consider the definition of `power` at line 9. In this case, `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 roughly 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 the lifetime model; we will check out an example called *Ownership semantics* in chapter 4. 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 an 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 an 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 referring to same identifier `<v>` are L-values.

On the other hand, function call expressions such as `gcd(3,2)` are R-values; their result (function return value) can't be referred 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 referred 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* copies only the literal memory contents.

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

Definition 4 (Copy function)**.** A 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)**.** A 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

```
fn outnl(i){
      out(i);
      out('\n');
}
```

```

struct B{
    let member = i32;
}

fn __init__(bp, i){
    type _ = enable_if<bp==@B>;
    type _ = enable_if<i==i32>;

    outnl("Init B started");
    bp!.member = i;
    outnl("Init B finished");
}

fn __copy__(bp1, bp2){
    type _ = enable_if<bp1==@B>;
    type _ = enable_if<bp2==@B>;

    outnl("Copy B started");
    bp1!.member = bp2!.member;
    outnl("Copy B finished");
}

fn __dest__(bp){
    type _ = enable_if<bp==@B>;

    outnl("Dest B started");
    __dest__(@(bp!.member));
    outnl("Dest B finished");
}

fn main() -> i32{
    let b = object<B>(5);
    outnl(b.member);
    return 0;
}
```

Listing 3.9: Lifetime semantics - output

```
Init B started
Init B finished
5
Dest B started
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.

3.4.4. Code generation

Code generation is a process which occurs parallel with type inference, in which we generate the necessary LLVM-IR code. We will go through an example of code generation corresponding to member indexing expression (dot operator).

Listing 3.10: Code generation for dot operator

```
1  @add_method_te_visit(sa.MemberExpression)
2  def _(self: sa.MemberExpression, tc: TypingContext,
3        fc: context.FunctionContext):
4      e = self.expr.te_visit(tc, fc)
5      self.lvalue = self.expr.lvalue
6
7      if not isinstance(fc.types[e], ts.StructType):
8          raise ierr.RuntimeExpressionError()
9
10     if self.member not in fc.types[e].members:
11         raise ierr.RuntimeExpressionError()
12
13     if not self.expr.lvalue:
14         put_dest(self, fc, tc, e)
15
16     return ir.get_member(e, self.member, fc)
```

Listing 3.10 describes the process.

First, we have to get a *stack symbolic register* for a structure we are indexing upon (line 4). *Symbolic registers* are names which are assigned to only once on a lexical level. We use a term *stack symbolic register* to denote symbolic registers which contain pointers to stack, where the actual values are held. This approach, in which we always manipulate the pointers to the stack is suggested in LLVM documentation, and greatly simplifies the design. While the reader might suspect this will cause unnecessary overhead, it is important to note that LLVM infrastructure optimizer will analyze the code and remove the indirection where possible.

Since an indexed member is an lvalue only when the structure being indexed is, we transfer the `lvalue` property to the indexed member (line 5).

After checking for some common errors, we introduce destruction code (line 14) if and only if the expression is an rvalue (since nothing else refers to it, we have to destruct it now).

Lastly, we call upon a `get_member` function, which will insert the necessary code to get a stack symbolic register (a pointer) to the member, and return its name.

You can also notice the variables `fc` (of type `FunctionContext`) and `tc` (of type `TypingContext`). The `FunctionContext` carries information about the current function being inferred, while the `TypingContext` carries information about the data global to the whole program (other functions, structs,...)

Listing 3.11: `get_member` function

```
1  def get_member(src: str, member_name: str,
2          fc: context.FunctionContext):
3      dest = fc.scope_man.new_tmp_var_name("member")
4      fc.types[dest] = fc.types[src].types[member_name]
5
6      stmn = fc.types[src].mangled_name
7      ind = fc.types[src].members.index(member_name)
8
9      fc.code.extend([
10         f"\t%{dest}␣=␣getelementptr␣inbounds␣" +
11         f"%{stmn},␣%{stmn}*␣%{src},␣i32␣0,␣i32␣{ind}"
12     ])
13     return dest
```

The `get_member` function will create a new and unique name (line 3) for a stack symbolic register which contains a pointer to a requested member.

Then it will update its type in this function context (line 4).

Afterwards, it will extract the *mangled name* of a type being operated upon (struct type), which is actually its symbolic name (line 6). The index of a member will be found using the member name (line 7).

Finally, the code will be extended by a LLVM-IR command (lines 9 to 12), and the name of a newly generated stack symbolic register will be returned.

All functionality of AGT is implemented in a similar way, and an interested reader will surely want to consult the source code if he wants to learn more.

4. AGT program examples

In this chapter we will introduce the reader to AGT by presenting, explaining and discussing example programs.

4.1. Limiting polymorphic behaviour

Polymorphism is a phenomenon of function calls referring to different functions (or in AGT language - function types) when invoked with different sets of argument types.

In order to demonstrate the limiting of polymorphic behaviour to certain types, we provide an example in listing 4.1, in which we calculate nth Fibonacci number recursively.

Listing 4.1: Fibonacci with recursion

```
fn outnl(i){
    out(i);
    out('\n');
}

struct is_int<T> -> R{
    type R = (T==i64 | T==i32 | T==i16 | T==i8);
}

fn fib<T>(n) -> T{
    type _ = enable_if<is_int<T>>;
    type _ = enable_if<n == i32>;

    if (n<=2){
        return cast<T>(1);
    }
    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 referred 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, a compilation error will be reported. 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 *substituting struct definition*. Substituting 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 substituted 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 type of `n`.

Listing 4.2 shows the output of our program.

Listing 4.2: Fibonacci with recursion - output

```
-112
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 built-in 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 nth Fibonacci number. The reader should now study the substituting struct definitions in lines 6, 9 and 13. It is important 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

```
100
144
-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

```
fn outnl(i){
    out(i);
    out('\n');
}

struct rect<T>{
    let a = T;
    let b = T;
}

fn main() -> i32{
    let a = in<i32>();
    let b = in<i32>();

    type irect = rect<i32>;
    let rect_heap = heap_object<irect>(a,b);

    outnl(rect_heap!.a);
    outnl(rect_heap!.b);
    heap_free(rect_heap);
    outnl(rect_heap!.a);
22      outnl(rect_heap!.b);
23  }
```

Listing 4.6: Heap object allocation - input

```
15
16
```

Listing 4.7: Heap object allocation - output

```
15
16
2305
0
```

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 members `a` and `b`, we `heap_free` the allocated memory, and print out the members again. While this is a nondeterministic behaviour, we have kept an output where members are 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

```
fn printpos(ptr, pos){
    out(ptr[pos]!);
    out('\n');
}

fn setpos(ptr, pos, val){
    ptr[pos]! = val;
}

fn main() -> i32{
    let H = heap_alloc<i32>(3);
    setpos(H, 1, 123);
    setpos(H, 2, 456);
    printpos(H, 0);
    printpos(H, 1);
    printpos(H, 2);
    heap_free(H);

    return 0;
}
```

Listing 4.9: Raw heap allocation - output

```
0
123
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 char array 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 char array 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 *class* for string management. By class, we mean a struct which holds the required data, along with necessary functions for its use.

String struct should be initialized with a char array literal. The initializer should allocate more memory, this time on the heap, which will be copied to from the initial char array literal. The copy method should perform a deep copy, since these strings are meant to be mutable.

Finally, special functions should be constructed to allow for controlled getting and setting of string characters and also for printing the string out.

We have implemented all of these in the following example, and urge the reader to study it throughly.

Listing 4.10: String class

```
fn outnl(i){
    out(i);
    out('\n');
}

fn len(cptr) -> i32{
    type _ = enable_if<cptr == @char>;
    let k = 0;
    while (cptr[k]! !='\0'){
        k=k+1;
    }
    return k;
}

struct string{
    let arr = @char;
    let len = i32;
}

fn __init__(sptr, cptr){
    type _ = enable_if<sptr == @string>;
    type _ = enable_if<cptr == @char>;

    let k = len(cptr);

    sptr!.len = k;
    sptr!.arr = heap_alloc<char>(k);
    while (k >! 0){ // >! is strictly greater than
        sptr!.arr[k-1]! = cptr[k-1]!;
        k=k-1;
    }
}

fn __copy__(sptr1, sptr2){
    type _ = enable_if<sptr1 == @string>;
    type _ = enable_if<sptr2 == @string>;

    sptr1!.len = sptr2!.len;

    let k = sptr1!.len;

    sptr1!.arr = heap_alloc<char>(k);
    while (k >! 0){
        sptr1!.arr[k-1]! = sptr2!.arr[k-1]!;
        k=k-1;
    }
}

fn __dest__(sptr){
    type _ = enable_if<sptr == @string>;

    heap_free(sptr!.arr);
}

fn get(sptr, i) -> char{
    type _ = enable_if<sptr == @string>;
    type _ = enable_if<i == i32>;

    return sptr!.arr[i]!;
}

fn set(sptr, i, c){
    type _ = enable_if<sptr == @string>;
    type _ = enable_if<i == i32>;
    type _ = enable_if<c == char>;

    sptr!.arr[i]! = c;
}

fn out(s){
    type _ = enable_if<s == string>;

    for(let i=0; i <! s.len; i=i+1;){
        out(s.arr[i]!);
    }
}

fn main() -> i32{
    let a = object<string>("Hello world!");
    let b = a;

    outnl(a);
    outnl(b);
    outnl(get(@a,1));
    set(@a, 1, 'E');
    outnl(a);
    outnl(b);

    return 0;
}
```

Listing 4.11: String class - output

```
Hello world!
Hello world!
e
HEllo world!
Hello world!
```

4.6. Ownership semantics and shared pointers

A recurring problem in all languages without automatic garbage collection are memory related issues. *Smart pointers* are an approach which is widely used in languages such as C++. A *shared pointer* is a struct which holds a non-exclusive ownership of an underlying resource. Holding ownership implies that this struct could be responsible for releasing the resource after it is no longer used. Non-exclusivity of ownership means that other objects could also be responsible for its release. By limiting the owning objects to shared pointers only, and using them in a well-defined way, we prevent almost all of these memory issues (we will mention a problem with this approach at the end of the section).

A shared pointer will copy the pointer which is passed to it, and free the memory when needed. Freeing the memory will be done when no other shared pointers have access to same pointer, and the one performing the check is being destructed. This will be achieved by using a shared counter (see line 8 in listing 4.12). The counter will be incremented when making copies of the shared pointer, and decremented when destructing them. Finally, the last shared pointer which gets destructed (the check at line 27) will release the ownership (in this case, destruct the object and free the memory).

The reader should study the listing 4.12 and program output in listing 4.13.

Listing 4.12: Shared pointer

```
 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){
        outnl("RELEASE");
        __dest__(sptr_ptr!.item_ptr);
        heap_free(sptr_ptr!.item_ptr);
        heap_free(sptr_ptr!.count);
    }
}

fn __copy__(sptr_ptr_dest, sptr_ptr_src){
    outnl("COPY");
    type T = enable_if_resolve<sptr_ptr_dest!.T>;
    type _ = enable_if<sptr_ptr_dest == @shared_ptr<T>>;
    type _ = enable_if<sptr_ptr_src == @shared_ptr<T>>;

    sptr_ptr_src!.count! = sptr_ptr_src!.count! + 1;

    sptr_ptr_dest!.item_ptr = sptr_ptr_src!.item_ptr;
    sptr_ptr_dest!.count = sptr_ptr_src!.count;
}

fn make_shared(item_ptr) -> shared_ptr<inner>{
    type inner = enable_if_resolve<item_ptr!>;
    return object<shared_ptr<inner>>(item_ptr);
}

fn main() -> i32{
    let a = make_shared(heap_object<i8>(15i8));
    outnl("Before B is initialized");
    outnl(a.count!);
    outnl(a.item_ptr!);
    {
        let b = a;
        outnl("After B is initialized");
        outnl(a.count!);
        outnl(a.item_ptr!);
        outnl(b.count!);
        outnl(b.item_ptr!);
        b.item_ptr! = 13i8;
        outnl("After B is changed");
```

```
66          outnl(a.count!);
67          outnl(a.item_ptr!);
68          outnl(b.count!);
69          outnl(b.item_ptr!);
70      }
71      outnl("After B is destructed");
72      outnl(a.count!);
73      outnl(a.item_ptr!);
74
75      return 0;
76  }
```

Output:

Listing 4.13: Shared pointer - output

```
INIT
Before B is initialized
1
15
COPY
After B is initialized
2
15
2
15
After B is changed
2
13
2
13
DEST
After B is destructed
1
13
DEST
RELEASE
```

We have added messages to signal when certain lifetime operations get executed. The main thing to notice is that variable b has a lifetime which ends at line 70, the closing of a block. This is the exact time when its destructor is called. One can thus notice that the `count` gets decreased from 2 to 1 after destruction. Finally, at the end of the `main` function, lifetime of `a` ends too, and the resource is released.

This approach resolves most of the memory issues, but as mentioned earlier, can result in a memory leak nonetheless. This can occur only in presence of *circular references* — an interested reader can find more about this phenomenon in various sources on the web.

5. Future improvements

Since AGT is a new project, it has not yet matured enough to be used in complex applications. The author lists 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 (at least 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, AGT would allow for references. This would benefit the ease-of-use on a syntactic 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 lexical 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 repeated some common function definitions (`outnl`, for example) quite often. This is because AGT doesn't have a multiple source file support. There are problems which need to be addressed before implementing it.

Firstly, a common problem with multiple source files in other languages is multiple inclusion and multiple definition problems which come 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 and line). An alternative is to ignore all function definitions which are *the same*, that is, equivalent on either a lexical or a syntactic level.

Secondly, a convention regarding file search paths must be adopted. Although there are many examples of this being done quite successfully, 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 code bases accompanied by it would require AGT to have some sort of name separation policies. These are often called namespaces. Implementation of namespace 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 develop 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), with Python3 and 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 usable programming language.

The developed language, AGT, was shown to be expressive, yet safe. AGT's syntax is particularly strict, which prevents the programmer from making hard-to-find mistakes. The type system and inference process, with this flavour, are a novel concept which 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. AGT is a compiled language and is a candidate for use in low-level applications which require low overhead or primitive runtime environment.

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

The author concludes that AGT's design, implementation, and principles have shown to be functional and practical when applied to common tasks an application programmer performs. Thus, the goal of this thesis has been successfully accomplished.

Bibliography

[Aho u. a. 2007] Aho, Alfred ; Lam, Monica ; Sethi, Ravi ; Ullman, Jeffrey: *Compilers Principles, Techniques, & Tools.* 01 2007

[Barbara Liskov 1974] Barbara Liskov, Stephen Z.: Programming with Abstract Data Types. (1974), S. 56–57

[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`

[Dictionary.com authors] Dictionary.com authors: *Parsing - definition.* – URL `https://www.dictionary.com/browse/parse`. – [Online; accessed 6-June-2021]

[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/`

[Pierce 2002] Pierce, Benjamin C.: *Types and Programming Languages.* 01 2002

[Stroustrup 2013a] Stroustrup, Bjarne: *The C++ Programming Language, 4th Edition.* 05 2013. – 481–526 S

[Stroustrup 2013b] Stroustrup, Bjarne: *The C++ Programming Language, 4th Edition.* 05 2013. – 163–164 S

[Stroustrup 2013c] STROUSTRUP, Bjarne: *The C++ Programming Language, 4th Edition*. 05 2013. – 585–588 S

[Stroustrup 2013d] STROUSTRUP, Bjarne: *The C++ Programming Language, 4th Edition*. 05 2013. – 795–802 S

Design of a strongly-typed programming language

Abstract

The goal of this thesis is to develop a new programming language by combining desirable properties of other widely used ones. We have defined the AGT programming language, a compiled, statically and strongly typed language which has a powerful type system, capable of performing compile-time computation and providing different forms of static polymorphism. We have implemented a Python based AGT compiler, which uses LLVM compiler infrastructure as a compiler backend.

We describe the AGT programming language syntax and semantics, and also provide numerous examples of effective AGT use in common programming scenarios. The thesis is concluded with plans for the future development of AGT.

Keywords: Programming languages, Compilers, Polymorphism

Oblikovanje strogo tipiziranog programskog jezika

Sažetak

Cilj ovog diplomskog rada razvoj je novog programskog jezika, kombinirajući poželjna svojstva drugih često korištenih jezika. Definirali smo AGT programski jezik, prevođeni, statički i strogo tipiziran jezik sa snažnim sustavom tipova, sposobnim izvesti izračune tijekom prevođenja, koji podržava različite oblike statičkog polimorfizma. Razvili smo prevoditelj za AGT u programskom jeziku Python, koristeći LLVM infrastrukturu prevoditelja za pozadinsko prevođenje. U radu smo opisali sintaksu i semantiku AGT programskog jezika, a dani su i brojni primjeri efikasnog korištenja AGT-a u čestim programerskim situacijama. Diplomski rad završili smo planovima za budući razvoj AGT-a.

Ključne riječi: Programski jezici, Prevoditelji, Polimorfizam