Grok: Final Report

Brendan Higgins

April 24, 2015

Abstract

Modern trends in programming languages have moved toward mixed paradigm (i.e. functional and imperative) languages. Grok attempts to uniquely combine functional style type systems with imperative style syntax. In particular, Grok has a fully algebraic type system that achieves polymorphism purely through ad-hoc polymorphism; while allowing mutable variables and static evaluation.

Contents

1 Introduction

1.1 Background

In recent years, mixed paradigm programming languages have become increasingly popular, combining elements from functional programming languages and more mainstream imperative and object oriented programming languages. To be clear, no precise set of features is entailed by either the term "functional" or "imperative", so, necessarily, I must define my terms.

The term "functional" in the context of programming languages is somewhat nebulous; however, for my purposes I consider a functional to be associated with several concepts: functions as first class objects, lazy evaluation, immutable variables, everything is an expression (as opposed to the existence of statements), algebraic data types, ad-hoc polymorphism as opposed to inheritance base polymorphism. In essence, a functional programming language attempts to root itself as being rigorously mathematically defined, as opposed to mirroring the underlying behavior of the hardware.

On the other hand, "imperative" is typically associated with mutable variables and strict evaluation, mirroring the underlying behavior of the hardware. Nevertheless, object oriented concepts have been mainstream for a while now and have become associated with the imperative paradigm in a perceived functional-imperative dichotomy. In truth, the actual landscape of programming languages is much more complicated, but is nonetheless unimportant in the discussion below. For my purposes, the important ideas to be familiar with are that the dominant languages of the past decade have these certain set of properties, which I have mentioned above, and will henceforth be referred to as imperative.

Accordingly, a programming language which mixes are large set of features from both of these categories will be referred to as "mixed-paradigm." Scala, Rust, Go, and Ruby (Scala in particular) are examples of languages in this camp. Grok would also, in theory, fall into this category.

Nevertheless, Grok is unique. Unlike other mixed paradigm programming languages Grok combines a functional style type system with imperative style syntax. In particular, Grok has an ad-hoc polymorphic, algebraic type system with mutable variables and strict evaluation, which will be explained in more detail below.

1.2 Project Goals

Grok is a project which I plan on working on long after this class is over. Accordingly, I have many long term goals for Grok which I will not be able to achieve as part of this project. Ultimately, I would like to be able to compile Grok to native machine code complete with a fairly sophisticated runtime; nevertheless, this is not a very achievable goal for a 14 week project class. Thus, by the end of this class I will have compiler which will compile Grok to an intermediate language (a form of three address code which can be used as a basis for native machine code generation)

1.3 Related Languages

As I mentioned earlier, Scala, Rust, Go, and Ruby are all languages that occupy the same domain as Grok. However, Grok's existence is still meaningful because Grok is unique; Grok has goals which are separate from that of the languages that I mentioned. In particular, Grok attempts to combine a simple algebraic type system with a highly expressive, yet imperative syntax. In addition to a number of features, Grok stands alone in this category and as such is inherently valuable if, for nothing else, a way to test this unique combination.

1.4 Advantages

Nevertheless, if we wanted to stack up Grok against these other languages, there are several places where Grok shines. In particular, each of these mentioned languages have failed in some fundamental aspect.

Of the languages I have slated Grok against, there is one language that I have definite favor towards. Not only is Scala my favorite language that I use regularly, I have drawn a tremendous amount of

inspiration and ideas for Grok from Scala. Particularly, I love Scala's syntax, but I think that Grok will be superior in this regard as Scala's syntax is too free. Scala makes it too easy for someone to be clever or lazy (implicit functions and variables are a great example of this which are called and applied *implicitly*); however, Grok's syntax will be greatly simplified. Outside of syntax, the field becomes much easier for Grok. Scala's type system is terrible. First off, Scala has an inheritance based type system over which I argue an algebraic type system is superior; however, Scala's type system fails in other regards; Scala has the classic JVM problem with generics (it practices type erasure), but this will not be a problem with Grok. Grok is being created with generics in mind. Lastly, despite the fact that Scala tries to promote strong runtime guarantees; it cannot fundamentally because it is built on (and is interoperable with) Java and so is forced to deal with things like type casts, exceptions, null values, etc. However, Grok is truly strongly type and as such will not allow null values or type casts; also, it will not have exceptions.

The next strongest contender is Rust. Rust is not nearly as popular as Scala and is not nearly as stable, but I agree with it most in principle. Rust is truly strongly typed. Its type system is algebraic and actually quite similar to Grok, but here it goes too far. Rust tries to use its type system to guarantee that resources are not leaked and to know the lifetime of a variable and even who owns a piece of memory at a given point in time. Although I think that it is super cool, I found it to be unwieldy in use. Rather than allowing greater expression, the type system is annoying to work in as so many properties of variables must be provided to make sure that the entire life of an object is accounted for. Worse, the compiler does not provide good error reporting; its errors are often more cryptic than the worse templating errors in C++.

I do not have very many interesting things to say about Go and Ruby. Starting off, Ruby is dynamically typed, which I think is inherently inferior to static typing when trying to make a good general purpose language. Also, Ruby is interpreted which makes it unsatisfactory for the types of jobs that I intend to make Grok suited for. Furthermore, similar to Scala, I think that Ruby is too unrestrictive. Go on the other hand is probably the most boring language. It is beautiful in its simplicity; however, I argue it is too simple. Go does not have generics. Go relies heavily on casting which defeats the purpose of type safety.

1.5 Disadvantages

Nevertheless, Grok, as proposed, is not without disadvantages. In the case of Scala, Go, and Ruby, these are all languages that are fairly popular, as such there are high quality libraries written in these languages; furthermore, they all have established niches. Scala has most certainly evolved to fit the space it is filling, and so in some sense these languages have already struck a balance in features and properties that Grok may not have in practice. To be clear, I have never written a Grok program of any complexity and so what looks good on paper might not actually work out as well in practice; I cannot know this until I try.

In the case of Scala and Ruby, there are most certainly cases in which one would desire to have that extra power. For example, because of Ruby's incredibly strong metaprogramming DSLs like Rails have been created which would otherwise be difficult to make. I myself have enjoyed a number of Scala's extreme power in several personal projects.

Although I criticized Scala for incorporating too much of Java, I cannot deny that Scala's interoperability with Java is not also an advantage. Scala can import any Java library as a Scala library without any special bindings. This can not be dismissed, if I want to have Grok have any success at any level (even for my own personal use) I am going to have to make a system for bindings, which is something that I do not currently have a plan for.

1.6 Progress

I have managed to implement nearly everything that I promised in my first report. The only things that are missing are methods and higher order functions (functions which take functions as parameters). This means, since my last report, I rewrote the Semantic Analyzer, I wrote the Type Checker, the Code Generator and the Virtual Machine. Nevertheless, I hope to have higher order functions at least, but

hopefully methods as well by the final submission deadline; if this is the case the I will have finished all that I set out to accomplish.

1.7 Updates

As I mentioned, I rewrote the Semantic Analyzer, this mostly had to do with the symbol table not providing an effective way for storing symbol definitions the way I needed for the Type Checker (I will talk more about this in the Methodology section). Aside from the things that I just mentioned in progress, I formally defined a three-address-code (TAC) specification for Grok, I also wrote a formal specification for everything in the current version of Grok below in the specification section.

2 Description

Grok's type system is algebraic; this means that concrete types are composed in two ways: as product types, and as sum types, which I refer to as a "struct" and a "union". A struct is just like a struct in C or Go, it is just a list of other types with names that are contained in the new type. A union is similar to C's notion of a union, in that a union may be one of any of the type which make up that union; however, Grok's unions are distinct in that Grok "remembers" what the underlying type is. In Grok, a method is just a special function which takes one of its arguments on the left hand side of the function name; as such, a method is separate from the definition of a concrete type.

As methods are not part of a concrete type's definition, there would be no way for a type to inherit a method from another type. In fact, Grok has no concept of inheritance. Instead, Grok relies on ad-hoc polymorphism; this means that concrete types have to satisfy a certain contract (promise to have a certain set of methods defined on them) in order to be considered to be polymorphic. In Grok, these contracts are called "interfaces." In Grok, a type may be declared to satisfy one of these interfaces at anytime by declaring an "instance," which contains a set of methods satisfying the interface. This and other details will be described in detail below in the Language Specification section.

As I mentioned above, Grok's syntax is primarily of an imperative style; what I mean by this is that Grok is strictly evaluated, meaning that code is executed in a linear order with all expressions being evaluated as soon as they are assigned to a variable or used as a parameter in a function. By imperative style, I also mean that Grok has both mutable and immutable variables.

2.1 Constraints

My project for this course is a compiler and VM for Grok that will allow Grok code to be executed on my computer. My compiler will accept compile any code which is within the language (specified below) and compile it to an intermediate state which may be executed by the VM.

2.2 Challenges

The main challenge will be that Grok is a fairly complicated language. Its grammar is straightforward being LL(*); however, it is semantically complicated in that types, interfaces and functions may be defined in any order. The hardest thing is going to be the type system.

First off, Grok supports type inference anywhere in a code block. Secondly, Grok's union operation may allow for ambiguity in function resolution. Basically, a type may be a member of two unions which could each have a function with a similar signature defined on them making it impossible to resolve which function should be called.

Grok also has a minor challenge in that it has newline terminated statements, taken at face value this, combined with the fact that almost any expression could also be a statement, would make the grammar context sensitive and not LL(*).

2.3 Proposed Solutions to Challenges

Although making such a type inferencer will be a large undertaking, it is a solved problem with a large amount of research behind it (other languages have this "problem"). Nevertheless, the solution for unions is more complicated; C++ actually has a similar problem in that it applies "conversion constructors" and "conversion operators" implicitly. C++ solves the problem by simply reporting an ambiguity as a compile time error and forces the programmer to fix it; I think this is the best solution as I do not see this being a common problem and do not want to sacrifice any freedom in how unions are used.

Nevertheless, solving the issue with newline terminated statements is actually a solved problem for my purposes. The solution is called "semicolon insertion;" wherein, the lexer simply recognizes places where a statement may be terminated and then inserts a special end of statement token which is ignored by the parser unless it actually needs to end a statement. Some languages which do this include JavaScript and Go.

3 Requirement Specification

3.1 Language Specification

3.1.1 Grammar Specification

The grammar for Grok is specified below:

```
compilationUnit
  : topLevelStatement* EOF
  ;

topLevelStatement
  : functionDefinition
  | methodDefintion
  | structDefinition
  | unionDefintion
  | interfaceDefinition
  | instance
  | statement
  ;

functionDefinition
  : 'func' typeParameters? Identifier funcParameters ':' type '=' expression
  ;

methodDefintion
  : 'meth' typeParameters? '(' type ')' Identifier funcParameters ':' type '=' expressio
  ;

structDefinition
  : 'struct' typeParameters? Identifier '{' field* '}'
  ;

unionDefintion
  : 'union' typeParameters? Identifier '{' type* '}'
  ;

interfaceDefinition
```

```
  : 'interface' typeParameters? Identifier ('extends' type)? '{' methodStub* '}'
  ;

instance
  : 'instance' typeParameters? type 'of' type '{' instanceMethod* '}'
  ;

typeParameters
  : '<' type+ '>'
  ;

type
  : Identifier typeParameters? ('=>' type)?
  ;

funcParameters
  : '(' ((funcParameter ',')* funcParameter)? ')'
  ;

funcParameter
  : Identifier ':' type
  ;

field
  : modifier=('var' | 'val') Identifier ':' type
  ;

methodStub
  : 'meth' typeParameters? Identifier funcParameters ':' type
  ;

instanceMethod
  : 'meth' typeParameters? Identifier funcParameters ':' type '=' expression
  ;

statement
  : variableDeclaration
  | variableAssignment
  | structAssignment
  | ifExpression
  | whileExpression
  | matchExpression
  | functionCall
  ;

variableDeclaration
  : modifier=('var' | 'val') Identifier (':' type)? '=' expression
  ;

variableAssignment
  : Identifier '=' expression
  ;
```

```
structAssignment
  : Identifier '.' Identifier '=' expression
  ;

expression
  : primaryExpression
  | lambda
  | booleanExpression
  | arithmeticExpression
  ;

primaryExpression
  : accessableExpression '.' accessor
  | accessableExpression
  ;

accessableExpression
  : '(' expression ')'
  | ifExpression
  | whileExpression
  | matchExpression
  | functionCall
  | block
  | variable
  | thisExpression
  ;

accessorExpression
  : functionCall
  | variable
  ;

accessor
  : accessorExpression '.' accessor
  | accessorExpression
  ;

ifExpression
  : 'if' booleanExpression block ('else' (block | ifExpression))?
  ;

whileExpression
  : 'while' booleanExpression block
  ;

matchExpression
  : 'match' expression '{' matchCase+ '}'
  ;

matchCase
  : 'case' Identifier ':' type '=>' expression
  ;
```

```
functionCall
  : Identifier arguments
  ;

thisExpression
  : This
  ;

block
  : '{' statement* expression? '}'
  ;

lambda
  : lambdaParameters '=>' expression
  ;

lambdaParameters
  : lambdaParameter
  | '(' ((lambdaParameter ',')* lambdaParameter)? ')'
  ;

lambdaParameter
  : Identifier (':' type)?
  ;

booleanExpression
  : booleanProduct '||' booleanExpression
  | booleanProduct
  ;

booleanProduct
  : booleanInverse '&&' booleanProduct
  | booleanInverse
  ;

booleanInverse
  : (inverse='!')? booleanTerm
  ;

booleanTerm
  : '(' booleanExpression ')'
  | comparison
  | primaryExpression
  | BooleanConstant
  ;

comparison
  : arithmeticExpression
    operation=( '=='
    | '!='
    | '<='
    | '>='
    | '<'
```

```
    | '>'
  ) arithmeticExpression
  ;

BooleanConstant
  : 'true'
  | 'false'
  ;

arithmeticExpression
  : arithmeticProduct operation=('+' | '-') arithmeticExpression
  | arithmeticProduct
  ;

arithmeticProduct
  : arithmeticTerm operation=('*' | '/' | '%') arithmeticProduct
  | arithmeticTerm
  ;

arithmeticTerm
  : '(' arithmeticExpression ')'
  | primaryExpression
  | arithmeticConstant
  ;

arithmeticConstant
  : IntegralConstant
  | FloatingPointConstant
  ;

IntegralConstant
  : [0-9]+
  ;

FloatingPointConstant
  : [0-9]+ '.' [0-9]*
  | [0-9]* '.' [0-9]+
  ;

variable
  : Identifier
  ;

This
  : 'this'
  ;

arguments
  : '(' ((expression ',')* expression)? ')'
  ;

argument
  : Identifier
```

```
    ;

Identifier
    : [a-zA-Z0-9_]+
    ;
```

The grammar below is some of the ANTLR code I am using to generate the parser. I decided to show this instead of the BNF version of the grammar as it is more concise (and I think easier to read); it is a superset of BNF with the addition of the Kleene Star, Kleene Plus, question mark (behaves same as question mark in a regular expression), and grouping operator. NOTE: The grammar I have provided here has two simplifications that will prevent it from compiling. For readability, I simplified the definition of an expression that creates a left recursion. I also removed the code that handles line termination.

What is not specified in the grammar above is that a special end of statement token is generated by the lexer after any potentially valid statement (*topLevelStatement*) followed immediately by a newline or a ';'.

3.1.2 Type Specification

Grok has three distinguishable kinds of types: Primitive Types, Function Types, and Compound Types.

Primitive Types are types that are built in to the language. These types are Int, Float, Bool, and Unit. The Int type represents a 32 bit signed integer value. The Float type represents a 32 bit floating point value equivalent to the IEEE's binary32 floating point value as defined in IEEE 754-1985. The Bool type represents boolean values. The Unit type serves as the top type in the language (as will be described in the type derivation section). The Unit type is the type of definitions and statements. A Unit type is Grok's only *unrealizable* type, meaning that no object can be realized having an actual type of Unit and thus no operations can be performed on the Unit type. Char, Array, and String will be supported soon, but are not currently.

Function Types are the types that Functions posses. Any function has a type:

```
* => * ... * => *
```

where each '*' is a valid type representing an argument taken by the function, except the last '*' which represents the return type; for example:

```
func negate(value: Int): Int = 0 - value
```

has the type:

```
Int => Int
```

A function with no return value is said to have a return value of Unit with its full type as:

```
* => * ... * => Unit
```

A function that takes no parameters is still said to have one formal parameter of type Unit expressed as:

```
Unit => *
```

As the Unit type is unrealizable no actual parameter may be provided, even if the object is formally assigned the type Unit. This means that even statements and definitions may not be passed into a function which has a Unit parameter type. The only valid way to express a function which accepts the Unit type is with no parameters:

```
func foo(): * = ...
```

(again, '*' is a stand in for any valid type, including Unit). A return type must always be specified; if the function has no return type, it is said to have a return type of Unit and is expressed as:

```
func foo(param0: *, ..., paramN: *): Unit = ...
```

There are two kinds of Compound types: Union types and Structs type. A Struct type may be difined as follows:

```
struct Foo {
  val field0: *
  .
  .
  .
  var fieldN: *
}
```

where 'val' means the field is immutable, it is fixed at the time that the Struct is constructed; 'var' means the field is mutable, it may reassigned to any valid value at any time; 'fieldN' and 'Foo' may be any valid identifier (see the grammar specification); and * is any valid type, except Unit. In addition to these requirements and the requirements specified by the grammar, field names must be unique within the Struct definition. In the language of algebraic type systems, Structs correspond to the concepts of Product types.

A Union type may be defined as follows:

```
union Foo {
  *
  .
  .
  .
  *
}
```

where '*' may be any valid type with the exception of Unit with the additional requirement that any given type may appear in the body of a given Union at most once; and 'Foo' is any valid identifier. In the language of algebraic type systems, Unions are Sum types.

When a Struct or a Union is defined, functions are defined known as *constructors*. In the case of Structs, a single constructor is defined which has the same name as the struct and takes its fields as arguments; for example, the following Struct definition:

```
struct Foo {
  val baz: Baz
  var bar: Bar
}
```

results in the creation of the following function definition:

```
func Foo(baz: Baz, bar: Bar): Foo = ...
```

this function is created automatically by the compiler and cannot be replaced by a user defined constructor.

In the case of Unions, several constructors are defined, one for each member type. For example, the following Union definition:

```
union Foo {
  Bar
  Bool
}
```

yields the following constructors:

```
func Foo(bar: Bar): Foo = ...

func Foo(bool: Bool): Foo = ...
```

3.1.3 Type Derivations

In Grok, some types are said to *derive* other types; this means that this type is implicitly convertable to the other type. There are several simple rules that govern this behavior: first, all types derive Unit; Int derives Float; and any type derives any Union it is a member of (in this last case, the Union's constructor will be implicitly called). There are three places in which a type may be converted to some other type: an assignment, if the declared type derives the assigned type; a function call, if the assigned type of an expression derives the declared type of its corresponding function parameter, a conversion will occur; finally, if an Int is used in a Float arithmetic expression, it will be converted into a Float. The fact that all types derive Unit is not used for conversions; rather, it is used to decide whether an expression evalutates to anything and whether it may be treated as a statement.

3.1.4 Control Flow

There are several control flow constructs in Grok:

if expressions are much like if-statements in most other programming languages except they are expressions:

```
if (condition0) {
  expr0
} else if (condition1) {
  expr1
} else {
  expr2
}
```

where 'conditionN' is anything of type Bool; exprN is any valid expression; there may be arbitrarily many 'else if's; and the 'else' expression is not required. The type of this expression is the least upper bound, in terms of derived types, of the body expressions; if there is no else case, the expression evaluates to type Unit.

while expressions are just like while statements in most other languages, except they are expressions:

```
while (condition) {
  expr
}
```

where 'condition' is anything of type Bool; and expr is any valid expression. The type of this expression is the type of 'expr'.

match expressions are for deconstructing Unions, they are defined as follows:

```
match (unionExpression) {
  case member0: Member0 => expr0
  .
  .
  .
  case memberN: MemberN => exprN
}
```

where unionExpression is some expression that evaluates to a Union type;

memberN: MemberN => exprN

is a lambda that takes one of the member types of unionExpression (there must be one lambda for each of the member types of the union). The type of this expression is the least upper bound of the return types of the member lambdas.

3.1.5 Scoping and Declaration

Grok uses static scoping like most popular programming languages. Anything that is not in some higher scope is in the global scope; which, unlike other scopes, is the only scope that is not strictly statically evaluated. Instead, definitions are evaluated at the same time; this is to make it more convenient when defining functions and types which may depend on each other. The global scope also does not allow arbitrary expressions. Higher levels of scope are created in function definitions and blocks. Function definitions, with the exception of lambdas, are only allowed to be defined at the global scope. Blocks are syntactic elements denoted by:

```
{
   ...
}
```

Blocks may be created in any expression and are required in certain constructs (like if-expressions, see the grammar specification for where blocks are required).

Symbols (functions and variables) may not be redeclared in the same scope; however, a symbol may be redeclared at a higher scope; in this case it does not replace the previous symbol, it merely *shadows* it; this means that when this higher scope is exited the original symbol becomes visible again. However, multiple functions of the same name may be declared provided that they have the same return type, but different parameters; this applies to compiler defined constructors, but not lambdas; this is because lambdas do not actually have a name, even if they are immediately assigned to a variable, the name belongs to the variable not the lambda.

When declaring a variable, it is *required* to initialize the variable to some value; however, it is optional to specify the the type. If the type is not specified, the compiler will infer the type to be the type of the expression it was initialized to. Only variables declared with 'var' may be reassigned to some other value.

3.2 TAC Specification

The compiler generates a list of instructions known as three-address-code (TAC). There are three types of TAC elements: Labels, Operands, and Instructions.

Labels are simply a index that corresponds to a possition on the list of instructions.

Operands are arguments taken by instructions; there are four kinds of Operands: IntOperand, FloatOperand, BoolOperand, and ReferenceOperand. These are further subdivided into three additional types each: RealOperand, TempOperand, and ConstOperand. The only difference between RealOperands and TempOperands is that RealOperand derive there identifier from an actual variable or parameter in the code that was compiled; whereas, TempOperands are distinguished by a number. ConstOperands represent an constant value; ReferenceOperands do not have a ConstOperand.

Instructions closely resemble machine code operations in that they usually only take a few operands, they specify what memory a result is written to, if any, and they do not need any context to be interpreted. Most instructions specify some operation that takes one or two operands, yeilds a result and stores the result in another operand. Those are:

- Gotos which manipulate the pointer counter. The ConditionalGoto jumps to the label if the condition is satisfied and the ConditionalGotoNot jumps to the label if the condition is not satisfied.
- CompoundAccess instructions read a field belonging to an object (the object's position + offset) into the specified operand.
- CompoundAssign writes the value of an operand into an object (the object's position + offset).
- HeapAllocate creates an object in the heap and returns a reference to it. The list of operands is just a convenient way to keep track of the size; it is just the list of types that belong to that object (Int, Float, Bool and Reference). The selector in HeapAllocateUnion denotes which field is populated; it is not an Operand because it is known at compile time.

- Push instructions take Operands and put them on the stack to be referenced as parameters by a subsequent function call (see CallFunc).
- Pop removes *size* instructions from the stack, it removes the function parameters that were previously pushed.
- CallFunc instructions push an Operand, if any, to receive a return value, push the return address, push a stack frame, and jump to a label.
- Return instructions pop the stack frame, pop the return address and jump to it, and assign the return value to a preallocated operand, if any, (see CallFunc).
- Print takes an operand and prints it out; this is not something that would usually be in TAC, but I put it in for demonstration purposes.

The list of all instructions and operands is listed below:

```
IntTempOperand(id: Int)
IntRealOperand(name: String)
IntConstOperand(value: Int)
FloatTempOperand(id: Int)
FloatRealOperand(name: String)
FloatConstOperand(value: Float)
BoolTempOperand(id: Int)
BoolRealOperand(name: String)
BoolConstOperand(value: Boolean)
ReferenceTempOperand(id: Int)
ReferenceRealOperand(name: String)

// Basic Operations.
AssignInt(result: IntOperand, value: IntOperand)
BinaryOperationInt(result: IntOperand, left: IntOperand,
  op: ArithmeticOperator, right: IntOperand)

ToFloat(result: FloatOperand, left: IntOperand)
AssignFloat(result: FloatOperand, value: FloatOperand)
BinaryOperationFloat(result: FloatOperand, left: FloatOperand,
  op: ArithmeticOperator, right: FloatOperand)

AssignBool(result: BoolOperand, value: BoolOperand)
BinaryOperationBool(result: BoolOperand, left: BoolOperand,
  op: BooleanOperator, right: BoolOperand)
Not(result: BoolOperand, left: BoolOperand)

ComparisonInt(result: BoolOperand, left: IntOperand,
  op: ComparisonOperator, right: IntOperand)
ComparisonFloat(result: BoolOperand, left: FloatOperand,
  op: ComparisonOperator, right: FloatOperand)

AssignReference(result: ReferenceOperand, value: ReferenceOperand)

Goto(label: Label)
ConditionalGoto(condition: BoolOperand, label: Label)
ConditionalGotoNot(condition: BoolOperand, label: Label)

// Compound data types.
```

```
CompoundAccessInt(result: IntOperand, reference: ReferenceOperand, offset: Int)
CompoundAccessFloat(result: FloatOperand, reference: ReferenceOperand, offset: Int)
CompoundAccessBool(result: BoolOperand, reference: ReferenceOperand, offset: Int)
CompoundAccessReference(result: ReferenceOperand,
  reference: ReferenceOperand, offset: Int)

CompoundAssignInt(value: IntOperand, reference: ReferenceOperand, offset: Int)
CompoundAssignFloat(value: FloatOperand, reference: ReferenceOperand, offset: Int)
CompoundAssignBool(value: BoolOperand, reference: ReferenceOperand, offset: Int)
CompoundAssignReference(value: ReferenceOperand,
  reference: ReferenceOperand, offset: Int)

HeapAllocateStruct(result: ReferenceOperand, fields: List[Operand])
HeapAllocateUnion(result: ReferenceOperand, selector: Int, members: List[Operand])

// Functions.
PushInt(value: IntOperand)
PushFloat(value: FloatOperand)
PushBool(value: BoolOperand)
PushReference(value: ReferenceOperand)

Pop(size: Int)

ParamFromStackInt(result: IntOperand, offset: Int)
ParamFromStackFloat(result: FloatOperand, offset: Int)
ParamFromStackBool(result: BoolOperand, offset: Int)
ParamFromStackReference(result: ReferenceOperand, offset: Int)

CallFuncInt(result: IntOperand, label: Label)
CallFuncFloat(result: FloatOperand, label: Label)
CallFuncBool(result: BoolOperand, label: Label)
CallFuncReference(result: ReferenceOperand, label: Label)
CallFunc(label: Label)

ReturnInt(value: IntOperand)
ReturnFloat(value: FloatOperand)
ReturnBool(value: BoolOperand)
ReturnReference(value: ReferenceOperand)
Return

// Debugging
Print(value: Operand)
```

It should be noted that this TAC is not entirely typical and is most certainly not in its final form. Grok is ultimately intended to run natively; however, in order to reduce the scope of the project for the purposes of this class I had to stop at intermediate code generation. As a result, I simplified my assumptions about the memory and added instructions to support making the VM easier to write.

3.3 User Interface Requirements

The compiler shall be a command line program which will be called in the following format:

grok [FLAGS]... [FILE.grok]...

with flag options being:

1. –version: prints out the version of the compiler, this is important as language spec will change in the future.
2. –compiler_type=[COMPILER_TYPE]: allows a selection of what compiler to use. The options are:
    - default: same as tac, this is the option that is slected by default.
    - vm_compiler: runs the VM on the generated code.
    - interpreter: runs interpreter on AST.
    - tac: prints out the generated three-address-code (TAC) in a human readable format.

and *FILE.grok* is a file containing valid Grok source code.

There are a couple of changes I made above: first off, I decided to role the VM into the compiler and just provide an option for executing the generated code after compiling it. The reason for this change is that I found when I was testing that it provided a more effective workflow to just execute the TAC when it was generated; having a separate executable for the VM and generating compiled files only makes sense in the context of multifile projects which Grok currently does not support.

3.4 Functional Requirements

The calling the compiler on a grok source file will cause the compiler to read in the contents of the file, compile it into the TAC format described above and execute it on its internal VM.

3.5 Methodology

Since my last report, the biggest challenge I faced was actually the same problem I described in the previous report with the Type Checker: Inferring the type on lambdas is a much harder problem, the problem arises because Grok allows lambda parameter types to be omitted in certain cases:

```
val double: Int => Int = {num => num * 2} // OK: can deduce types from type on variable.
list.map(elem => elem.toString()) // OK: can infer type from the method map.
```

The first case is not the difficult case and is not an important feature of Grok. However, the second case is a very important feature, it is the primary reason for the lambda notation: for use in higher order functions. Also, inferring the types in this case is much more difficult. The problem is that looking up the method map is dependent the type of the arguments being passed in; however, we need the signature of map in order to infer the signature of the lambda being passed in.

My proposed solution is this: create an *intermediate type* for lambdas which includes the number of expected arguments as well as the return type, from the above example the *intermediate type* of the lambda in map would be assigned:

```
* => String
```

where $*$ would be a special compiler specific *bottom type* and so could be coerced into any other type. The type inferred for the result would be as strict as possible to guarantee it would fit any possible signature. Then, the function may be looked up as normal with the lambda finally receiving an *actual type*.

The proposed solution worked; however, implementing it was harder than I thought. The way that the old Semantic Analyzer and symbol table worked left the AST, the Semantic Analyzer and the symbol table too weakly coupled in order for this method to work because I actually had to attempt to type check a lambda with several parameter options to determine which ones are compatible with the set of respective functions. Although, I could have just passed some context object in, I decided that it was too messy as I did not want to have Type Checker code inside the symbol table. So I had to rewrite it.

Other than that the only real challenge I faced was determining how memory should be layed out for objects in the VM, this was not really as much of an exercise in something I had never really thought about much before.

3.6 Software Design and Development

The compiler behaves according to the typical compiler architecture:

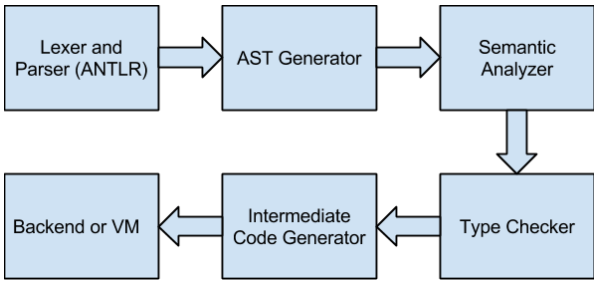

Figure 1:

In the first stage (Parsing and Lexing) the compiler reads in the code to compile and generates a parse tree. These details are handled for me by my parser generator, ANTLR. ANTLR takes a grammar file that specifies Grok's grammar and generates a lexer and parser.

In the second stage, I walk the parse tree and generate an AST (Abstract Syntax Tree). This stage is technically not necessary; however, it is easier for me to work with an AST that is specialized for the following stages.

In the third stage, I verify that the semantics, which the parser can not check, are correct. This involves building a symbol table at each level of scope and verify that symbols are appropriately defined (defined before use, not redeclared, etc.)

The forth stage takes the AST and the symbol table and infers all unspecified types and verifies their consistency.

The fifth stage of the compiler takes the AST and the symbol table along with the type information and generates intermediate code in the above specified format (see TAC Specification).

Finally the VM executes the generated code.

4 Project Management

I am not really sure what to put here other than I found that my project management plan was reasonable in that I was able to accomplish nearly everything I set out to by the times that I had specified. I found that working on multiple sections at once and striving for a minimal functioning compiler was ultimately more constructive that trying to build it in modules as I originally proposed, at least at the end (I will talk more about this under lessons learned); nevertheless, I found my project management plan to have worked pretty well.

5 Testing and Evaluation

I still have not written any formal tests; however, I have written several programs and not only verified that they produced the TAC I expected, I also verified that the generated code correctly executed on the VM, which shows, in these cases at least, that the TAC represents the same program that I put in.

6 Lessons Learned

I learned a tremendous amount of things from this project both in techical things regarding compilers as well as things regarding managing a large project.

6.1 Technical Learning

I have written interpreters before, but they have all been for either toys or DSLs that were pretty simple. This is the first real compiler that I ever tried to write. Nevertheless, things went pretty smoothly as I progressed through the Lexer and Parser, the AST Generator, and to some extent the Semantic Analyzer. I did learn about semicolon injection, but the first place that I really started to expand my knowledge was when I got to the unknown territory of Type Checking; I found that the symbol table I wrote which was filled by the Semantic Analyzer was not at all suitable for what I needed.

In the past, the languages that I worked with were usually dynamically type, but definitely did not perscribe complex type inference. In this case, I could not use the usual type derivation algorithms. They worked in most cases; however, function lookups, and lambda type inference were particularly challenging. In the end, I had to develop my own type derivation method. In fact, this is still something in need of work; nevertheless, I feel as though my grasp of type theory has been substantially deeped over the course of this project and has actually inspired some new ideas that I hope to get into one of the next versions of this project.

I also learned a tremendous amount about memory organization and code generation from working on my code generator and virtual machine. Even though I did not decide to use raw memory for my virtual machine, I could have done something substantially easier when it came to representing user defined objects. I chose to only allow myself to use three primitive types and one that is tantamount to pointers. I have written assembly before, and not just the mandatory MIPS. I have written x86 and even mixed C and assembly. I knew what cdecl was, but actually starting from, more or less, scratch, I believe has taught me a lot.

6.2 Design Learning

There are very few experience I have had at Case that have taught me about engineering and designing things out in the wild, this is one of those experiences. This project was teaching me from the very begining, I remember sitting and deciding what would be a part of this project and I already decided that it had to be something achievable. I tend to set goals too high for myself, which has ultimately served me well, but I knew I could not do this in this case. For my own fulfillment I had to produce a mostly finished project with all major goals met. So I decided that writing the entire compiler for the entire language that I had envisioned was not possible (memory management, native code generation, linear type system); nevertheless, the whole point doing this as a class project was to make progress toward the true goal. So I had to figure out what I could cut from, not only the compiler, but the language to a size that was feasible for this class without making wasting work writing a compiler that did not have any useful parts for what I am ultimately aiming to make.

I also learned a tremendous amount about project management. I tried to follow the timeline I planned for myself rather strictly, and although I managed to stay on time, I found that I wasted a tremendous amount of work. As I mentioned above, I wrote the Semantic Analyzer, and then found I basically needed to rewrite it because it was too simple. I then saw myself repeating my mistake with the Type Checker and I realized that I did not know exactly what it was supposed to do because I did not know what the exactly what the Code Generator was going to do. Not that I did not have a clear idea in my head of what it was supposed to do, but rather there were just too many details to keep track of the way I was trying to do it. Instead, I let myself hack something together, just to get *something* working that I could see run. I found this to be a tremendous success, once I had a framework to build on I found myself making really astounding progress. I know this probably sounds like a major plug for agile developmet; however, I see this more as a lesson concerning about organizing my thoughts, allowing myself to make an attempt at something without trying to make it perfect, and how to keep myself motivated about something.

7 Conclusions, Final Status, and Future Work

I managed to accomplish almost everything that I set out to do. At the time of writing this report I did not have methods or higher order functions working. Which were roughly outlined in the original submission. I certainly planned to have these working; however, I am pretty close on higher order functions. I only have one other thing to work on this weekend. So I would be willing to bet money that I would be able to at least finish higher order functions by Tuesday. I am going to try to finish methods by then as well, but it might be too ambitious. Other than that, everything works amazingly well.

I have a very extensive goal planned out for this project. I do not want to discuss it all here, but I will offer a brief outline. My first step would of course be filling in the last few things that were not completed for this project (higher order functions and methods). The next step would be implementing Interfaces, type parameters and other primitive types (string, array, char, uint8, uint16, etc). After this, the plan becomes much more ambitious, I plan on adding linear types and special reference for which the compiler can verify are safely managed for their entire lifetime. Finally, I plan on making a backend which compiles Grok to native code.

8 Appendices

8.1 Programmers' Manual

8.1.1 Requirements

- Linux, BSD, or OSX
- SBT version 0.13
- git

8.1.2 Installation

Clone the git repo:

```
git clone https://github.com/bjh83/grok.git
```

Download dependencies and compile the project:

```
sbt compile
```

Currently the best way to run grok is as follows:

```
sbt "run [flags] [path-to-source]"
```

8.2 Third Party Software

- ANTLR4 - lexer and parser generator
- sbt-antlr4 - plugin to SBT to use ANTLR4

Everything else needed to perform work on the program is described in detail in the Language Specification.