Implementing a general-purpose programming language in Haskell

Ben Moon

Abstract

Angle is a programming language written in Haskell. Angle is inspired by many languages, including Perl, Lisp, Python, Ruby, and Haskell. I cover the major areas of research which were required in order to be able to implement the language. I demonstrate Angle's basic functionality through the use of examples, whilst covering the major features of the language at the same time in the form of a language reference. I then go on to explain some of the design choices and how they affected the language, as well as providing an overview of the parser implementation. I conclude by revisiting my original expectations for the language, and compare them with the finished result.

Contents

Part I
Introduction

The following section introduces Angle, some implementation concepts and the software used to create it.

1 Reading this document

This document should be read in conjunction with the source files provided. Examples of code in this document are for illustration purposes and should be assumed to be non-functional - the source files should be referred to for working code.

Relevant modules will be stated before any sections describing the development of code.
Additional documentation is provided in the form of HTML files generated by Haddock. Any capable browser will be sufficient to view the documentation interactively.

1.1 Notation

Throughout this document, references to code, modules, and other features will be made using the notation described in the following table.

Element	Example	Represents.
Modules	`Angle.Parse.Token`	Reference to a Haskell module.
Code	`let x = 7`	Source code of various types.
Library	`tasty`	Reference to a Haskell library.
Command	`angle --help`	Command that can be entered at a terminal.

2 Background

What follows is an overview of external software and libraries used in this project.

2.1 Haskell

I have decided to write my project using the Haskell programming language [11]. Haskell is a statically-typed, functional, lazy language that supports type-inference.

Haskell's type system makes it very easy to develop projects quickly and safely as a project will not compile unless types match up.

2.2 Software

In developing Angle, I have used various software to deal with compiling, documenting, testing and many other aspects.

2.2.1 Cabal

Cabal [14] is a package manager for projects written in Haskell. It allows information about the package, including name, description, library information and test suites to be specified within a `.cabal` file, and can then be used to simplify the task of compiling, running tests and installing packages.

2.2.2 Haddock

Haddock [16] is a system for generating documentation from annotated Haskell source files. I chose Haddock due to it's intuitive syntax, ease of readability and integration with Cabal.

2.2.3 Glasgow Haskell Compiler

The Glasgow Haskell Compiler [17], or GHC, is the main actively-maintained compiler for Haskell. It is very stable and provides many additional features that may be used when required by a programmer.

2.2.4 Git

Git [26] is an open source version control system that allows changes in a project to be staged, committed and tracked. This means that changes to the project can easily be undone and reviewed, and allows changes to be labeled for future reference.

2.2.5 Libraries

Throughout the project I have used many libraries written by other developers. Some part of the Haskell Platform [12], others standalone. Below are some of the packages I have used.

QuickCheck - a testing library that can generate random tests for checking that specified properties hold when applied to different data. [6]

Tasty - a test framework that allows the combination of tests into a single test suite for running. [5]

Criterion - a benchmarking library for measuring the performance of code. [21]

3 A brief overview of language implementation

The following section introduces some concepts relevant to language design, along with a summary of which concepts are most appropriate to Angle.

3.1 Translator Software

Translator software is used to translate a program written in one language to another language, without loosing any functionality.
Although people often refer to languages as 'compiled' or 'interpreted', the translator software is distinct from the language itself, and thus a language could be both interpreted *and* compiled.[1]

3.1.1 Compilers

A compiler is a piece of software that takes a file containing the source code for one language, and produces output in another language. The output language is often in a form that can be executed by the system's CPU - namely object code.

3.1.2 Interpreters

Interpreters differ from compilers in that instead of programs first being translated to machine code before being executed, instructions are translated into pre-compiled subroutines and executed directly at run-time, thus possibly incurring a speed decrease when compared to a similar compiled language. [24]
One of the main disadvantages of an interpreter is that it is required every time a program needs to be run. Source code is also more transparent (which may or may not be a disadvantage), as the files are run directly by the interpreter.

3.1.3 Angle

Angle is an interpreted language and this project develops the `angle` command-line interpreter for executing Angle source files.

3.2 Programming Paradigms

Programming paradigms are a way of writing programs using a programming language. Most languages encourage the use of one or more paradigms (for example, Haskell strongly encourages the use of the functional paradigm), but will not usually *force* the programmer to use a single one.

Common paradigms Following is a brief description of some of the more common paradigms. [23]

3.2.1 Declarative

The declarative paradigm is one in which programs describe what actions should be performed, but don't explicitly state *how* this should be done.

[1] This is quite common with the more popular languages, for example: Python and Lisp both have compiler and interpreter implementations.

Common examples of declarative languages are database query languages such as SQL.

The programmer specifies that the required action is retrieving a column from a table in a database. No indication of how to achieve this is given.

```
SELECT column FROM table WHERE condition
```

3.2.2 Imperative

In the opposite spectrum of paradigms to declarative is imperative. In imperative languages the programmer specifies how to perform computations via a sequence of step-by-step instructions. [20] Contrary to declarative languages, the order of execution in imperative languages can greatly affect the outcome of the program.

3.2.3 Object-oriented

Object-oriented languages focus on the use of objects. Objects are structures that carry their own state (usually in the form of properties) and have methods that can be used to change or communicate this state.
Inheritance is usually present in object-oriented languages, and allows a parent-child relationship between different objects.

A python3 example of class-based object-oriented programming.

```
class User:
    access = "restricted"

    def __init__(self, name):
        self.name = name


class Admin(User):
    access = "unrestricted"

    def __init__(self, name):
        super().__init__(name)


sam = Admin("Sam")
```

In the above example, it could be said that `sam` is an `Admin` called `"Sam"`, and that a `Admin` is a type of `User`.
`access` is a class variable, and does not rely on an instance (`sam`) being created, thus `Admin.access == 'unrestricted'`, but also `sam.access == 'unrestricted'`.

3.2.4 Functional

In functional languages, computations are treated as the evaluation of mathematical functions [13]. Additionally, side-effects are usually low to non-existent, a result of referential transparency.[2]
Several features are prominent in functional languages:

functions as first-class citizens this means that functions are treated equally when used in place of other data structures (integers, lists etc...) and can thus be passed around as data.

higher-order functions functions that are higher-order can take functions as arguments or produce them as results. This makes sense when functions are first-class citizens as they should be generally indistinguishable from other data.

Haskell is an example of a purely-functional programming language.

3.2.5 Angle

Angle is intended to be primarily a functional language. To support this Angle provides first-class citizen functions and higher-order functions. Angle is not intended to be a purely functional language, and thus imposes no restrictions on where side-effecting code can be performed.

3.3 Type systems

Type systems broadly fall under two categories based on when type checking is performed.

3.3.1 Static type systems

In static type systems, certain type-related criterion must be met before a program can be executed. These criterion can ensure a program will have well-typed operations and that types are not used incorrectly. [15]
Statically-typed languages often require the programmer to annotate code, stating the types of variables and functions when they are declared; leading to increased verbosity.[3]

3.3.2 Dynamic type systems

In a dynamically-typed language, types are checked at run-time, this means that type-correctness is not ensured and errors may occur as a result of types being used in places for which they are not valid. [28]

[2] The same result should always arise with the same arguments.
[3] Java and C, for example.

3.3.3 Angle

Angle is dynamically typed. I chose this system for a few reasons: firstly, it can be very difficult and time-consuming to write a type checker, and due to time constraints this would not be feasible; secondly, one of the more prominent features of Angle - parameter constraints (Section 8.1.2) allows the programmer to ensure that only certain types of values can be passed into functions they define - thus allowing them to still reason about their code in a manner similar to if a static type system were in place.

3.4 Typing

There are two main types of type-system that are determined by the stage in which type-checking is performed; to accompany this, there are two main types of typing, strong and weak. These terms are not well-defined but an outline is provided below.

3.4.1 Strong

A strong type system is unlikely to perform any type conversions or coercions and is likely to be quite strict about which types can be used in different operations.

3.4.2 Weak

Weakly-typed languages are likely to perform type conversions or coercions and provide a lesser distinction between the different types. [29]

3.4.3 Angle

Angle is, in general, strongly typed; implicit coercion of types is rarely performed and the types of variables will never be changed without explicit instruction to do so.
Angle does provide some functions for type coercion such as the built-in 'asType' function that allows the programmer to, for instance, convert a string to a list of characters.

3.5 Angle Summary

Using the terms defined in Section 3, Angle can be summarized as the following:
Angle is intended to be a strong and dynamically typed, functional, interpreted, general-purpose programming language.

Strong generally types are preserved unless explicitly changed.

Dynamic type checking is performed at run-time.

Functional Angle supports functions as first class citizens as well as higher-order functions.

Interpreted Source files are executed directly using the `angle` command-line interpreter.

General-purpose There was no specific use-case in mind when developing Angle, as such it is intended to be able to cope with the implementation of a variety of different types of software.

4 Project Overview

This project aims to provide the following:

Angle (the library) A collection of Haskell modules that define Angle's internal structure.

angle (the software) An interpreter built upon the Angle library that allows programs written in Angle to be executed.

The Angle Reference This document - outlines Angle and its various features.

4.1 The library

The library defines all of Angle's features (See Part II); the scanner, parser and language representation.[4]
The library is independent of the interpreter, and thus it would be possible to write a stand-alone compiler for Angle built upon the library directly.

4.2 The software

The 'angle' program supports two main modes of execution: interactive and non-interactive.

4.2.1 Interactive Angle

When Angle runs in interactive mode, the user is presented with a prompt at which they may enter one or more lines of code which can then be executed. This mode allows a programmer to debug code, test short snippets and ensure code has the correct syntax before using it in a program.

4.2.2 Non-Interactive

When Angle runs in non-interactive mode, achieved by passing a file to be executed, a source file is read and run by the Angle interpreter. This mode is how programs should usually be run.

[4] See Part III for an overview of how the library was implemented.

Part II
Language Reference

5 Introduction

The following reference describes the language features, grammar and methodology of Angle. Each section describes a feature or ideal of Angle; sections introducing syntax features will include a grammar in Extended Backus Naur Form.

5.1 Notation

5.1.1 Extended Backus Naur Form (EBNF)

EBNF is an extended version of Backus Naur Form, a notation that can be used to express the grammar of formal languages. [7]
BNF can be used to describe context-free grammars, [19] which are grammars that consist of names and expansions (the components), meaning that it may be used to express a grammar for Angle.

6 Program structure

A program written using Angle is made up of statements: statements may in turn be made up of many more statements, a language structure or an expression.

6.1 Grammar

The building blocks of Angle programs are statements; statements themselves being made of assignments, expressions and language constructs.

```
stmt        = single_stmt  | multi_stmt                    ;

single_stmt = function_def | stmt_expr      | stmt_control
            | stmt_loop    | stmt_condition | stmt_assign
            | stmt_raise   | stmt_try_catch | stmt_comment ;

multi_stmt  = '{' { stmt } '}'                             ;
```

6.2 Comments

Comments represent code that will be ignored by Angle. Comments start with a `#` character and continue to the end of the line.
Comments should be used to document code or temporarily disable sections of code during development; Angle throws away comment contents before execution.

6.2.1 Grammar

```
stmt_comment = '#' { <any character except newline> } newline ;
```

6.3 Identifiers and reserved words

6.3.1 Reserved words

Reserved words (or keywords) are reserved identifiers that may not be used for variable names, and have a special meaning in Angle.
The following is a list of the reserved words in Angle:

break	catch	continue	defun	do	else
false	for	if	in	null	raise
return	then	true	try	unless	while

6.3.2 Reserved identifiers

Certain identifiers representing variables have predefined meanings in Angle, some of these may be overwritten, others may not.

Name	Can be overwritten?	Use
main	no	Whether the current program was invoked directly.
_it	yes	Holds the value of the last computation.
as_constr	no	Whether the current function was called as a constraint.
stdin	no	Handle for managing the program's standard input stream.
stdout	no	Handle for managing the program's standard output stream.
stderr	no	Handle for managing the program's standard error stream.

6.3.3 Grammar

```
alpha          = 'a'..'Z'                              ;
digit          = '0'..'9'                              ;

identifier     = simple_ident  | function_ident        ;

simple_ident   = (alpha | '_') { alpha | digit | '_' } ;
function_ident = '$' simple_ident                      ;
```

6.4 Literals

Literals allow the specification of constant values for some of Angle's built-in types. [18]

6.4.1 Strings

Two types of string literal are supported in Angle: backslash-escaped and non-backslash-escaped.

By default, escape sequences in strings will be recognized (for example, '\n' would be recognized as a newline), but the 'e' prefix can be used to treat all backslashes literally (thus '\n' would become a backslash followed by an 'n').

```
string = [ 'e' ] '"' { string_char } '"' ;

string_char = <any character allowed in a Haskell String> ;
```

Additionally, each character of a string may be represented as a literal.

```
char = ''' char_char ''' ;

char_char = <any character allowed in a Haskell Char> ;
```

Note that Angle uses the same character escaping as Haskell. [2]

6.4.2 Numeric literals

Two types of numeric are supported by Angle: floats (which should have at least the range and precision of the IEEE double-precision type), [3] and integers.

```
integer = [ '-' ] digit { digit }                          ;

float   = [ '-' ] digit { digit } '.' digit { digit }      ;
```

6.4.3 Booleans

Booleans represent truth values in Angle. They are used in conditional statements[5], and any other location which require a true or false value.

```
boolean = 'true'  | 'false'                                ;
```

6.4.4 Lists

Lists represent a collection of values which can be passed around in a single data structure. Angle's lists can contain any number of different types, and may even have nested lists. Section 19.1 describes some specific nuances of lists.

```
list = '[' { literal ',' } ']' ;
```

[5] Described in Section 15

6.4.5 Ranges

Ranges can be used to specify a set of values with a common difference without initially producing all the values. A range can then be evaluated later when required.

```
range = '(' literal '..' [ [ literal ] [ '..' literal ] ] ')' ;
```

Note that although other values exist in the language (namely handles) and have a show syntax, they have no read syntax and can thus only be obtained through the use of built-in functions and language features.

7 Data

Within a program it is important to be able to acquire, process, and output data. From a low level perspective, data is just represented in binary form, but this is very inconvenient for a programmer to work with. Thus many programming languages include what are called 'data-structures', which accommodate working with this data. Angle is no exception and provides several data-structures, outlined below.

Datatype	Use	Example
String	Representing sets of Unicode characters	`"string"`
Integer	Representing arbitrarily large integral values	`42`
Float	Representing floating-point values	`3.14`
List	Grouping values, lists in Angle are heterogeneous meaning that different types of data may be stored within a single list	`[1, "string", true]`
Boolean	Representing truth values	`true`
Character	Representing individual Unicode characters	`'c'`
Range	Representing an enumeration across values of a certain type	`(1..7)`
Null	Special void value for when a value must be returned but it doesn't make sense to return anything else	`null`
Lambda	Representing function bodies	`(() 1;)`
Keyword	Representing constant names without strings	`:keyword`
Handle	Referencing file descriptors	`{handle: file}`

More complex data structures can be built out of these basic types. For example, a conventional hash or dictionary could be represented by a list of lists, each of length two with a keyword as the key and any other data as the value.

```
hash = [[:key1, "value1"],
        [:key2, 200]
```

```
[:key3, (() null;)]
]
```

Then, using parameter constraints (see Section 8.1) one could define a predicate that determines whether a list represents a hash, and then be used as a constraint on any functions that should only accept a hash.
Of course this would not be nearly as efficient as the hash implementation in many languages, but the principle of being able to build more complex data from the standard set of types is an important one.

8 Functions

Functions and subroutines (when they don't have an explicit return value) are essentially blocks of code that can be reused. Angle supports functions in two forms: as bare lambdas (the actual block of code) and function-variables. There is no difference between the call-signature-body of a function and a lambda, but in general use, when a lambda has been given a name (through either explicit assignment or the `defun` statement) it is referred to as a function.

8.1 Parameters

8.1.1 Types of parameter

Parameters (the variables that take on the values of arguments) in Angle come in two forms, each with optional modifiers.
The first form is the standard parameter, these are position dependent and will take on the value of the argument placed at the equivalent position in a function call.
The second form is a catch parameter; catch parameters, when defined, will collect any additional additional arguments into a list that can be accessed normally as a list, or expanded within a function call to pass in the collected arguments.[6] Catch parameters allow the implementation of variadic functions.

8.1.2 Parameter Constraints

Parameter constraints are references to functions attached to a parameter. When an argument is passed to the function and the parameter in that position has a constraint, the argument is passed to the constraint function.
The function then acts as a predicate, and should return true if the argument satisfies the constraint and false otherwise.

Example of using a parameter constraint to prevent division by zero

```
# A simple predicate function, it will always return true or false.
```

[6] Variables containing lists may be expanded in a similar manner.

```
defun nonzero(x) {
  ^(== x 0);
}

# This function won't accept a zero divisor.
defun divide(x, y:@nonzero) {
  (/ x y);
}
```

Restrictions Not just any function can be used as a constraint; for the successful use of a function as a constraint it must follow certain rules for the constraint call:

- The function must be able to take a single argument after any additional arguments are supplied. See Table 1 for some examples of valid and non-valid constraint calls.
- For the given call, the constraint **must** return a boolean value.

Note that Angle won't check that a function will *always* behave well as a constraint - so a function may sometimes return a boolean, sometimes not, but still be a valid constraint for certain use cases.

Constraint parameters	Call form	Valid call
`foo(x, y, z)`	`x:@foo`	no
`foo(x)`	`x:@foo`	yes
`foo(x, ..xs)`	`x:@foo`	yes
`foo(x, y)`	`x:@foo(1)`	yes
`foo(x, ..xs)`	`x:@foo(1)`	yes

Table 1: Calling constraints

Multi-purpose functions Quite often it would make sense to have a function act one way when called as a constraint, and another way when called normally.
Angle supports this behavior by providing the `as_constr` variable. `as_constr` is a special variable that holds a boolean value which is `true` if the current context is a constraint call, and `false` otherwise.
As an example, `int`, defined below, will determine whether the passed value is an integer or not when used as a constraint, but will attempt to convert the given value to an integer when called normally.

```
defun int(x) {
  if as_constr then {
    try {
      return (== x asType(1, x));
    } catch :typeCast {
```

```
      return false;
    }
  }
  asType(1, x);
}
```

Constraints outside of parameters As constraints represent a very useful family of predicate functions, Angle allows the programmer to call them outside of parameter lists.
When called with a prefix @, a function will be called as if it were called as a constraint in a parameter list - with the exception of there being no implicit first argument.

```
int(2.0);
# 2

@int(2.0);
# false
```

Constraints with arguments When used as a constraint in a parameter list, the first argument to the function will be the current argument (e.g, in `(x:@foo)`, the first argument to `foo` will be `x`). In this form the calling parentheses can be omitted.
Constraints can be called with more arguments, allowing great flexibility in how constraints are used.

Given a function `foo` with parameters `(par1, par2, par3, ..., parn)`, the constraint call should look like `(x:@foo(arg2, arg3, ..., argn))`, where `argN` will bind with `parN`. Note that there is no `arg1`, the current argument (`x`) will still be passed as the implicit first argument.

```
# x is bigger than n?
defun larger_than(x, n) {
  (> x n);
}

defun increment_large(x:@larger_than(100)) {
  (+ x 1);
}
```

8.1.3 Annotations

Annotations allow the programmer to restrict the types of arguments that are passed into a function.
There are three possible annotations:

`$x` Requires the argument to be a lambda.

`!x` Requires the argument to be a non-lambda.

`x` Imposes no restriction on the argument.

Annotations make it easy to define higher-order functions.

```
defun map($f, xs) {
  for x in xs do {
    f(x);
  }
}
```

Annotations allow the programmer to quickly state whether the parameter should be a function, literal or any value. This makes it easier to reason about higher-order functions and quickly see where functions should be passed in - as well as having the function reject invalid arguments.

8.2 Closures

Closures are special lambdas that carry a snapshot of the scope at the time of their creation.

8.2.1 Declaring closures

There are two methods of defining closures in Angle:

Using Return When using the `return` statement with a function or lambda, a closure is returned instead, using the current scope.
If the value being returned is already a closure (already has an associated scope), then the current scope will be super-imposed onto the global scope of the closure - i.e, the current scope becomes the parent scope of the old global scope of the closure, and the current global scope becomes the new global scope of the closure.

Using the Dollar Operator Using the dollar operator before a bare lambda will declare the lambda to be a closure, and capture scope appropriately.
Thus:

```
defun foo() {
  return (() x;);
}

bar = foo();
```

Is the same as

```
bar = $(() x;);
```

8.2.2 Closure Example

```
x = 7;
defun foo() {
  defun bar(y) {
    return (+ x y);
  }
  return $bar;
}

fun = foo();

fun(1);
# > 8

x = 5;

fun(1);
# > 8

fun2 = foo();

fun2(1);
# > 6
```

In the above example, `foo` returns a closure `bar` which takes a single argument and returns the sum of its argument and the value of `x` as it was when the closure was produced.
When `fun` is defined, the global value of `x` is 7, thus in the scope of the closure, global `x` will always equal 7.
When `x` is redefined later on, this does not affect the value of `x` that is used by the closure assigned to `fun`. But when `fun2` is defined, as the scope that `bar` captures is different, so will its value of `x`.

8.3 Accessing lambdas

As every variable in Angle can have both a function *and* value definition, *and* functions are first-class citizens, this means that occasionally the lambda associated with a variable may need to be retrieved.
Angle provides the dollar operator (`$`) that when used on an identifier, will return an expression representing the lambda stored in the variable.

8.3.1 Producing values from functions

By default, when no return value is specified in a function, the last produced value will be returned. Otherwise, the `return` statement can be used to exit early from the function and set the produced value to that specified.

```
defun foo() {
  1;
}

foo = (() 1;);
# Equivalent to above.

foo = 2;

foo;
# 2

foo();
# 1

$foo;
# (() 1;)
```

`foo` is defined as both a function (that just returns the integer 1) and a value (the integer 2).

When using `foo` as an expression, without calling it, it just returns the value definition. Likewise, when called as a function it returns the expected value 1. However, when the dollar operator is used, the function's definition is returned.

8.3.2 Grammar

Function calls

```
arglist       = '(' { expr ',' } ')'          ;

function_call = [ '@' ] simple_ident arglist ;
```

The optional @ sign in front of the identifier indicates whether to call the function as a constraint or a regular function. [7]

Function definition

[7] See Section 8.1.2

```
function_def = simple_ident '(' { parameter ',' } ')' stmt ;

parameter    = [ '!' | '$' | '..' ]
               simple_ident
               [ ':@' simple_ident [ arglist ] ]           ;
```

9 Variables

Variables reference a location in memory that represents the data associated with them. In Angle, variables can have two sets of data: the first is a non-function value, such as an integer, string etc..; and the second is a lambda that represents the variable as a function (see Section 8.3 on how to access this value).

9.1 Resolving variables

Angle is lexically scoped - meaning that the location at which a variable is defined determines where it can be accessed.
Angle's standard method of resolving variables is to first check the current scope, then recursively check the outer scopes until the global scope has been checked or a definition for the variable has been found.
When no value is found an exception is raised.

9.1.1 Different methods of resolving variables

On top of the standard resolution method, Angle provides two built-in functions for resolving variables differently:

- `nonlocal` - will do the same as standard resolution, but starting from the parent scope instead.
- `global` - will only check the global scope.

9.2 Assignment

Angle supports three assignment operators: `=`, `|=`, and `||=`; which represent local assignment, nonlocal assignment and global assignment respectively.

9.2.1 Local assignment

When assigning to the local scope, if a definition for the variable exists then it is overwritten, otherwise the variable is newly defined.

9.2.2 Nonlocal assignment

When assigning to a nonlocal scope, first the given identifier is resolved in the local-most scope that is a parent of the current scope, then this variable is overwritten with the given value. If no nonlocal variable exists for the identifier, then the operation fails with an exception.

9.2.3 Global assignment

When assigning to the global scope, the same process as for local assignment is performed, but in the global scope instead of the current scope.

9.2.4 Grammar

```
stmt_assign     = local_assign
                | nonlocal_assign
                | global_assign           ;

local_assign    = simple_ident '='   expr ;
nonlocal_assign = simple_ident '|='  expr ;
global_assign   = simple_ident '||=' expr ;
```

10 Exceptions and exception handling

A number of unexpected things can go awry during runtime, things such as types being used incorrectly, functions being called with the wrong number of arguments, requesting files that don't exist, and attempting to access closed handles, to name a few.
Angle attempts to anticipate many of these issues, and when they occur it wraps them up, along with their critical information, and throws them as exceptions.
Once thrown, an exception will propagate up the stack (function calls) until it either reaches the top level, where it will be displayed as an error to the user, or it is caught in a `try...catch` statement.

10.1 Handling exceptions

The `try...catch` statement allows the programmer to handle exceptions that are raised during execution.

10.1.1 Try

The `try` keyword initializes a `try...catch` statement. Any code wrapped in the `try` section is executed sequentially until it either finishes normally, or an exception is raised. If the code finishes normally, then the rest of the `try...catch` statement is skipped and the next statement is executed.

If the code results in an exception, then the rest of the body will be skipped and the `catch` component will be passed the exception for handling.

10.1.2 Catch

The catch component of a `try...catch` consists of one or more `catch` keywords, each followed by the exceptions they handle, and the code that should be executed when they are handling an exception.

Exceptions Each `catch` clause takes either a single keyword, or a list of keywords that name the types of exceptions they can handle.
When an exception is passed to the `catch` block, each `catch` clause is tried, from top to bottom, until either a `catch` matches the current exception, or there are no more `catch` clauses.
In the case that the exception is matched by one of the clauses, the accompanying body of code is executed, then the `try...catch` statement is exited.
If none of the `catch` clauses matched the exception, then it is re-raised and will continue its propagation.

Break A special form of the `break` statement can be used in the body of a `catch` clause.[8]
`break :try` will exit the current `catch` clause, and repeat the `try...catch` statement from the beginning of the `try`. This removes the need for wrapping a `try...catch` in a `while` loop for repeated user input, as demonstrated below.

10.1.3 Example of handling exceptions

A common example of `try..catch` usage is with handling user input; this is usually a case that shouldn't crash the program, so it makes sense to handle it.

```
try {
  user_input = input("Enter an integer: ");
  res = asType(1, user_input);
}
catch :read {
  print("That wasn't an integer!");
  break :try;
}
```

In this scenario, an attempt is being made to convert the user's input (string) to an integer; when the user's input does not represent a valid integer string, a `:read` exception is thrown. This is then caught by the `catch` and the user is notified of their mistake before being asked to enter another integer.

[8] See Section 11.3.1 for other uses of `break`.

10.1.4 Grammar

```
stmt_try_catch = 'try '   stmt catch_spec { catch_spec } ;

catch_spec     = 'catch ' catch_keyword stmt             ;

catch_keyword  = litKeyword | '[' { litKeyword ',' } ']' ;
```

10.2 Raising exceptions

Angle allows the user to throw new exceptions or re-raise exceptions being handled.
The `raise` statement takes a single keyword and either raises a new exception (that can be caught by using the same keyword) or re-raises an exception, based on the circumstances.

10.2.1 Re-raising exceptions

When the `raise` statement is used within the body of a `catch` clause, and the exception being handled matches the exception being raised, then the old exception will be re-raised with all the original information.

Example An example of when a programmer may wish to re-raise exceptions is when some operations are to be performed on a resource which must be closed when it is finished with.
In the example below, the `catch` clause ensures that the handle gets closed should any exceptions arise in the `try` block.

```
try {
  my_handle = open("some_file.txt", "<");
  ...some code...
  close(my_handle);
} catch :error {
  close(my_handle);
  raise :error;
}
```

10.2.2 Raising new exceptions

When using `raise` in a scenario that doesn't fulfill the criteria required in Section 10.2.1, a new exception is created. User exceptions are limited to a single keyword name, but are otherwise treated the same as regular exceptions.

```
try {
  for i in (1..10) do {
    if (> i 5) then {
```

```
      raise :greaterThan5;
    }
  }
}
catch :greaterThan5 {
  print("Got a number larger than 5!");
}
```

In this case the user exception is `:greaterThan5`, and is handled directly.

10.2.3 Grammar

```
stmt_raise = 'raise ' litKeyword ;
```

10.3 Built-in exceptions

The following is a comprehensive list of the exceptions that are built into Angle.

10.3.1 Special exceptions

The `:error` keyword can be used to catch all exceptions.
The `:user` keyword can be used to catch user exceptions.

10.3.2 Type exceptions

Raised when issues arise with parameter constraints or types are used incorrectly. Can be caught with `:typeError`.

Keyword	When thrown
`:typeMismatch`	If two types should be the same but aren't.
`:typeUnexpected`	If a type was used in a place where another was expected.
`:typeNotValid`	The type used is not valid in the current situation.
`:typeCast`	There is no possible conversion from one type to another when attempting a type conversion.
`:typeMismatchOp`	Wrong types are used in an operation.
`:typeExpectConstr`	A constraint returns false for a given value (in the function head).
`:typeConstrWrongReturn`	A constraint returned a non boolean value.
`:typeAnnWrong`	A value doesn't satisfy a parameter annotation.

10.3.3 Name exceptions

Raised when issues arise with the naming of things. Can be caught with `:nameError`.

Keyword	When thrown
`:nameNotDefined`	An identifier cannot be resolved.
`:nameNotDefinedFun`	There is no lambda definition for a variable.
`:nameNotDefinedLit`	There is no value definition for a variable.
`:nameNotOp`	Not a valid operator.
`:assignToBuiltin`	An attempt was made to assign to a non-overwritable built-in.

10.3.4 Call exceptions

Raised when issues arise when calling functions. Can be caught with `:callError`.

Keyword	When thrown
`:wrongNumberOfArguments`	Function called with the wrong number of arguments.
`:builtin`	An issue occurred when calling a built-in function.
`:malformedSignature`	Bad operation call.

10.3.5 Keyword exceptions

Raised when keywords are used incorrectly. Can be caught with `:keywordError`.

Keyword	When thrown
`:returnFromGlobal`	`return` was used in the global scope.

10.3.6 Value exceptions

Raised when Angle's types are used incorrectly. Can be caught with `:valueError`.

Keyword	When thrown
`:indexOutOfBounds`	Attempting to access a non-existent index of a list.
`:badRange`	The types used in a range don't match up.
`:infiniteRange`	Attempting to fully evaluate a range that would produce an infinite (or extremely large) set of values.
`:nonEnum`	An enumerable type was expected but not received.
`:divideByZero`	Zero was used as the denominator in an expression.

10.3.7 IO exceptions

Raised when issues arise whilst performing IO. Can be caught with `:ioError`.

Keyword	When thrown
`:alreadyExists`	One of the arguments in an IO operation already exists.
`:doesNotExist`	One of the arguments in an IO operation does not exist.
`:alreadyInUse`	One of the arguments in an IO operation is a single-use resource and is already being used.
`:deviceFull`	The device is full.
`:eof`	End of file has been reached.
`:illegalOperation`	Operation was not possible.
`:permission`	User does not have sufficient privileges to perform the operation.

These closely follow the errors outlined in the Haskell's IO error system. [4]

10.3.8 Syntax exceptions

Raised when issues arise with reading code at runtime. Can be caught with `:syntaxError`.

Keyword	When thrown
`:syntax`	Invalid syntax.
`:read`	String does not represent a literal.

10.3.9 Include exceptions

Raised when issues arise when including other files. Can be caught with `:include`.

Keyword	When thrown
`:doesNotExist`	Attempting to include a non-existent file.
`:syntax`	File contains invalid syntax.

11 Looping Structures

Repeated execution of a block of code can generally be achieved by two methods: recursion and iteration, both of which are supported in Angle.

11.1 Recursion

Recursion is achieved by a function self-calling with reduced arguments. A base case exists which when satisfied will return a well-formed value.

For the `factorial` *function, the base case is when the argument equals 0, and the value being passed in is reduced by 1 each time.*

```
defun factorial(n) {
  if (== n 0) then return 1;
  return (* n factorial((- n 1)));
}
```

Other recursive forms exist (such as tail-recursion), but Angle provides a while loop (see Section 11.2) rendering such forms unnecessary.

11.2 Iteration

Angle supports two constructs for iteration: `for` loops and `while` loops.

11.2.1 For loops

For loops, when given a enumerable value such as a list or range, will pass over the contained values, assigning each element to a temporary variable for access in the body. *A non-recursive implementation of the factorial function.*

```
defun factorial(n) {
  res = 1;
  for i in (1..n) do {
    res = (* res n);
  }
  return res;
}
```

For loop result A special feature of for loops is that they will collect the final value of each iteration into a list that is then returned as the result at the end of the loop (providing there are no breaks). Thus `for i in (1..5) do i;` results in the list `[1, 2, 3, 4, 5]`.

11.2.2 While loops

Unlike their `for` loop counterparts, which traverse a series of values, `while` loops execute until some condition is met.

```
count = 1;
while (< count 10) do {
  print("Count is less than 10!");
  count = (+ count 1);
}
```

11.3 Controlling loops

There are times at which it may be useful to exit a loop before it would naturally finish, or skip the rest of the current execution.
The `break` and `continue` statements provide support for these cases respectively.

11.3.1 Break

The `break [val]` statement ends the execution of the current looping structure, and sets the value produced to `val`, when supplied.

```
nom = 0;
denom = 10;
while true do {
  try {
    print((/ nom denom));
  } catch :divideByZero {
    print("Oops!");
    break; # Stop the loop!
  }
  nom = (+ nom 1);
  denom = (- denom 1);
}
```

11.3.2 Continue

`continue` skips the rest of the current loop iteration, causing the looping structure to start its next cycle.

```
for i in (1..5) do {
  if (== i 3) then continue; # Won't print for 3
  print(i);
}
```

11.4 Grammar

11.4.1 Looping structures

```
stmt_loop  = loop_for | loop_while             ;

loop_for   = 'for'   ident 'in' expr 'do' stmt ;
loop_while = 'while'            expr 'do' stmt ;
```

11.4.2 Loop control flow

```
loop_control     = control_break
                 | control_continue    ;

control_break    = 'break'    [ expr ] ;
control_continue = 'continue'          ;
```

12 Input and Output

Although the logic of a program can be defined in terms of pure functions that do not interact with the outside world - a non-library program must perform some input-output operations in order to be useful.
Angle provides several functions for IO operations.

12.1 Interactive with the terminal

A common need in scripts written for use in the terminal is to be able to perform basic interaction with the user. Angle facilitates this with the built-in `print` and `input` functions.

`print(string)` prints the given string to `stdout`.

`input(string)` prints `string` to `stdout` before returning the response from `stdin`.

A great deal of basic user interaction can be achieved just through the use of these two functions.

12.2 Handles

Angle makes use of handles to perform IO operations. A handle is a reference to a resource on the system, such as a file descriptor which provides access to some IO resource for reading and/or writing.

12.2.1 Standard streams

On Unix systems there exist three standard file descriptors to access the three standard streams: `stdin` is the input stream; `stdout` is the output stream; and `stderr` which is used for error messages.
Angle provides handles to these three descriptors by default, in the form of the variables `stdin`, `stdout` and `stderr`.

12.2.2 Obtaining handles

The built-in `open` function, which takes the form `open(file_name, access_mode)`, returns a handle providing access to the file `file_name` in the specified `access_mode`.

Access modes There are two main access modes in general: read and write. Write is further split into append and write (clobber). 'Write (clobber)' is used when the contents of the file shouldn't be preserved and will overwrite the contents of the file when writing. 'Append' on the other hand will preserve the contents and add the new text to the end of the file.

The four access modes are summarized below:

Symbol	Read	Write	Append
`"<"`	Yes	No	-
`">"`	No	Yes	No
`">>"`	No	Yes	Yes
`"<>"`	Yes	Yes	No

For example, if you wanted to read from a file called "file.txt", `open("file.txt", "<")` could be used to obtain the necessary handle.

12.2.3 Reading handles

Angle's built-in `read` function provides various means of reading from a handle's character stream. `read(handle)` reads the entirety of the remaining text, `read(handle, int)` will read `int` lines, then the modifier `:char` can be appended to the call to read individual characters.

Form	Meaning
`read(handle)`	Reads from `handle` until it hits the end of the stream.
`read(handle, int)`	Reads `int` lines from `handle`.
`read(:char, handle)`	Reads a single character from `handle`.
`read(:char, handle, int)`	Reads `int` characters from `handle`.

12.2.4 Writing to handles

As mentioned previously, there are two main write-modes that can be used with handles: write (clobber) and write (append).
The clobber mode will overwrite the current contents of the resource on each write, whereas the append mode will add the string to the end of the resource.
In both cases, when writing to a non-existent file, the file will be created.

12.2.5 Closing handles

When a handle is no longer in use it can be closed for reading and writing with the `close` function. It is advisable to explicitly close handles when they are no longer needed to free up file descriptors and allow them to be accessed by other operations later on.

13 Including code from other files

It is essential for code-reusability and the creation of libraries to be able to load code from other files in a program. The ability to do this means that sets of functions can be defined in one file, then used by many other programs such that the functions do not need to be written again.
There are two main methods of achieving this in Angle:

13.1 Eval

The `eval` built-in function takes a string and attempts to parse it as Angle source - this can be useful for loading small sections of source from a trusted location on the fly.
There are a few issues with this - any use of `eval` with user input is risky, as there is the potential for malicious code to be injected, instead the `asType` built-in should be used for converting strings to other types.

13.2 Include

The best method of loading entire files is the built-in `include` function. `include` takes a filename (or handle) and attempts to execute the contained text. Syntax errors are handled by `include`.
`include`, when passed a filename, will first check the standard Angle library locations for the specified file, then the relative path. If the given filename happens to be a relative or absolute path (starts with `.`, `/` etc..) then the library locations will not be checked.

14 Operations

Operations (consisting of operators and operands) are the fundamental means of manipulating data in Angle.
Angle supports three types of operator: infix binary, prefix unary and prefix variadic. The infix binary operators (such as `=`) take two operands, one on either side. The prefix unary operators take a single operand and are placed before this operand. The prefix variadic operators are used as the first symbol within parentheses, and take a non-set number of operands, separated by whitespace, until the closing parenthesis.

14.1 Operator types

Operators in Angle mainly come under four categories: arithmetical, logical, relational and assignment.

14.1.1 Arithmetical Operators

Arithmetical operators, which are generally variadic, perform mathematical arithmetic operations such as addition, multiplication and division.

Operation	Symbol	Example	Result
Addition	`+`	`(+ 1 2 3)`	`6`
Subtraction	`-`	`(- 1 2 3)`	`-4`
Multiplication	`*`	`(* 1 2 3)`	`6`
Division	`/`	`(/ 1.0 2 3)`	`0.1666666...`
Exponentiation	`**`	`(** 2 4)`	`16`
Negation	`-`	`-x`	`-5` (if `x = 5`)

Table 2: Arithmetical Operators

14.1.2 Logical Operators

Logical operators perform logical operations on booleans and are also mostly variadic.

Operation	Symbol	Example	Result
OR	`|`	`(| false false true)`	`true`
AND	`&`	`(& true true false)`	`false`
NOT	`^`	`^(& true true false)`	`true`

Table 3: Logical Operators

14.1.3 Relational Operators

Relational operators perform comparison between different types, all the relational operators are variadic.

There is a pair-wise grouping with relational operators, thus `(< 1 2 3)` becomes `1 < 2 AND 2 < 3`, or `1 < 2 < 3`.

Operation	Symbol	Example	Result
Equality	`==`	`(== 1 2)`	`false`
Less than	`<`	`(< 1 2)`	`true`
Greater than	`>`	`(> 1 2)`	`false`
Greater than or equal	`>=`	`(>= 1 2)`	`false`
Less than or equal	`<=`	`(<= 1 2)`	`true`

Table 4: Relational Operators

14.1.4 Assignment Operators

Assignment operators are all infix binary, and the use-case is always the same; associate some data with an identifier. See Section 9.2 for a more detailed explanation on how assignment works.

In a new non-global scope.

Operation	Symbol	Example	Result
Local assignment	`=`	`x = 1`	Local `x = 1`
Nonlocal assignment	`|=`	`x |= 1`	Error
Global assignment	`||=`	`x ||= 1`	Global `x = 1`

Table 5: Assignment Operators

14.1.5 A word on operations

There are a few important features to note about Angle's operators:

Assignment is not an expression All operations represent expressions, with the exception of the assignment operators. This means that the assignment operators cannot be embedded within other operations.

Overloading Many of the variadic operators are designed to act differently depending on their arguments: for example, the addition operator (`+`) performs arithmetic addition when all the arguments are numeric, but will perform appending when the first element is a list.
Many of the special cases and miscellaneous operators are covered in greater detail in the documentation for `Angle.Exec.Operations`.

List expansion in variadic operations Many of the variadic operators support the special case of being passed a single list. When this happens, the list is expanded and the operator acts upon the contents of the list, thus `(+ [a,b,c,...,x,y,z])` becomes `(+ a b c ... x y z)`.

14.2 Grammar

```
operation =     unop    expr
          | '(' varop { expr } ')'          ;

unop      = '^' | '-'                       ;

varop     = '+' | '-'  | '/' | '**' | '*'
                | '|'  | '&' | '>=' | '++'
                | '<=' | '>' | '<'  | '==' ;
```

15 Conditionals

A common feature to almost all programming languages is a structure for conditionally evaluating code based on the result of an expression. The canonical example of this is the

'if' conditional, that executes the accompanying body if the given expression evaluates to 'true', an 'else' form is also usually present.
Angle is no exception to this trend and implements its own conditional statements: 'if', and its counterpart 'unless'.

15.1 Truth in Angle

Many languages have different notions of what constitutes a 'truthy' value. In Perl, all values except for `0`, `"0"`, `""`, `()`, and `undef` are considered 'true'; in Ruby, only `nil` and `false` are considered to be 'false'; in Haskell, only the values `True` and `False` have any meaning when used as booleans.
Angle follows the latter path, with `true` and `false` being the only values that can be used in a boolean context. The main reason for this is that it makes code easier to understand[9] - writing `if nonzero(num_users) then...` is a lot easier to understand than `if num_users then...`.

15.2 Grammar

```
stmt_condition = cond_if | cond_unless                    ;

cond_if        = 'if'     expr 'then' stmt [ 'else' stmt ] ;
cond_unless    = 'unless' expr        stmt [ 'else' stmt ] ;
```

Part III
Implementation

This section provides an overview of the implementation of Angle, some important design choices and their ramifications, and the overall layout of the project.

16 Project structure

The Angle implementation is split into four sections:

16.1 Language representation

This is the internal representation of Angle's language structures; it describes how the various types relate to each other and the general structure the abstract syntax tree (AST) will take.
The actual implementation is not described further, but the documentation for `Angle.Types.Lang` covers this in great detail. Additionally, Part II provides a higher-level overview.

[9] It also avoids having to come up with potentially controversial ideas of what really is 'true'.

16.2 Executable

The executable is the tool that users will make use of in order to run software written in Angle.
The software provided, called 'angle' is command-line based and self-contained. Options provided by the executable can be found by running `angle --help`.

16.3 Parser

The parser has the job of compiling the textual source code into an AST representation of Angle within Haskell.
Within the module structure, the collection of modules `Angle.Parse.*` and `Angle.Scanner` define the implementation of the parser.
Within these modules, the `Parser a` monad is used to define the computational ability of the parser itself.[10]

16.4 Interpreter

The role of the interpreter is to execute the program according to the structure of the AST produced by the parser.
Within the implementation, the interpreter is the first stage that is able to interact via IO, all previous steps are pure-monadic.
The advantage of having the parser execute purely in terms of pure functions means that any two source-texts that are the same produce the exact same abstract syntax tree - as they should.

17 Process

The four main components of Angle mentioned in Section 16 interact to form the following execution method:

1. *User:* The programmer writes a program using Angle syntax.
2. *Executable:* The source file is run using the 'angle' program.
   (a) *Parser:* An attempt is made to compile the source to the abstract syntax tree based on the *language representation*.
      i. If the syntax does not coincide with that expected, the program will halt and alert the user.
   (b) *Interpreter:* The AST is executed.
      i. Run-time errors are raised as exceptions that may be caught by the program. If an exception makes it to the top-level, the program will halt and alert the user.

[10] Described in more detail in Section 18.3.2.

3. *Executable:* The program exits and memory is freed.

18 Parser implementation

18.1 Relevant Modules

`Angle.Scanner` defines the parser type. `Angle.Parse.Parser`, `Angle.Parse.Helpers` and `Angle.Parse.Token` implement the parser functions. `Angle.Types.Lang` defines language structures in terms of Haskell types.

18.2 Overview

Angle builds its parser on top of the `Parser a` monad - a custom monad that supports a combinatory parsing style.

The parser-library components:

- `Angle.Scanner` defines the `Parser a` monad and the fundamental functionality of the parser.
- `Angle.Parse.Helpers` defines the functions to support combinatory parsing.

Parser implementation:

- `Angle.Parse.Token` uses the previously defined combinators to build parsers for the basic structures in Angle (strings, keywords, numerics).
- `Angle.Parse.Parser` uses combinators defined in `Angle.Parse.Helpers`, along with the parsers defined in `Angle.Parse.Token` to define the parsers for each of the language constructs, and the main parser that combines these in order to parse an entire Angle program.

18.3 The Parser Monad

18.3.1 What is a monad?

In the context of a purely functional language such as Haskell, a monad is a structure that represents a certain type of computation and the rules associated with it. Monads are particularly useful because they allow the combination of effects whilst following the accompanying rules. [27]

The Maybe monad An example of a monad in Haskell is the `Maybe a` monad. `Maybe a` is often used to represent some computation that may fail.

A `Maybe a` *data declaration*

```
data Maybe a = Nothing | Just a
```

There are two operations associated with monads: 'return' and 'bind'. 'return' allows a non-monadic value to be lifted into the monad, and 'bind' allows the combination of monadic computations.
For the `maybe a` monad, 'return' wraps the value in the `Just` constructor, and 'bind' unwraps the value from the 'Just' constructor and passes it to the next function, or produces 'Nothing' if the first computation produces 'Nothing' as well.

```
return a == Just a

Nothing >>= _ == Nothing

Just a >>= f == f a
```

18.3.2 The Parser Monad

With the definition of a monad out of the way, it can be understood that the `Parser a` monad should have the operations 'bind' and 'return'.
The example of the `Maybe a` monad given above is very simplistic, it does one thing and one thing only. The `Parser a` monad will have to provide a lot more functionality if it is to be used for language parsing.

Monad Transformers Monad transformers are special structures that allow the combination of monads. [22, 30]
Monad transformers must satisfy the standard monad laws, but possess an additional operation 'lift' that promotes monadic computations to the combined monad of the transformer. [1]
This effectively allows the stacking of monadic effects, for example, if a monad was required that could both keep state and fail, the `maybe a` monad could be combined with the `state s a` monad via the `MaybeT m a` monad transformer.

```
type FailAndState s a = MaybeT (State s) a
```

What it should do With monads and monad transformers in mind, all that needs to be done is to state the desired computational abilities, pick the correct monads and combine them to form the final monad.

State The `Parser a` monad will obviously need to be able to keep track of its internal state. The parser will be running through a source file, making requests to the scanner and collecting characters to form the result.
The `State s a` monad was chosen to satisfy this as it provides a simple interface and all the required functionality without side-effects. [8]

Environment The `Parser a` monad will need access to a source string throughout its lifetime. In Haskell, the `Reader e a` monad is used when a static 'environment' of type `e` should be passed along with computations without being altered. [9]

Failure Although it would be nice to think that all input to the parser would be well-formed, it is likely that a string could not be parsed due to being syntactically incorrect. The `Parser a` monadic should be able to fail, and describe *why* it failed.
The `Maybe a` monad mentioned earlier can be used to represent computations that can fail, but it is limited in that it can only indicate that the computation failed, not *why*. For failure with additional information, the `Except e` monad can be used.[11]

The monad stack To combine these monads the use of monad transformers is required, as mentioned earlier.
An important point to note is that combining monads is, in general, not commutative - meaning that the order of operations *is* important.
An example of this would be `ExceptT e (State s) a` versus `StateT s (Except e) a`. The former describes a monad in which it is possible for the result of each computation to be a failure, whereas the latter describes a monad in which the entire computation can fail.

```
type ES a = ExceptT String (State Int) a
type SE a = StateT Int (Except String) a

message1 = return "hello" :: ES String
message2 = return "hello" :: SE String

runState (runExceptT message1) 1
> (Right "hello", 1)
# The result (first part of the tuple) is wrapped in the
# Either e a monad.

runExcept (runStateT message2 1)
> Right ("hello", 1)
# The entire computation is wrapped in the Either e a monad.
```

As the above example shows, the former monad is wrapping the inner value in exceptions, whereas the latter is wrapping the whole computation with potential failure.

Ordering With this in mind, the ordering of the stack can be decided.
Clearly an `ExceptT e m a` transformer will have to sit on the top of the stack, as either the whole file can be parsed, or none of the file.
Regarding the `Reader e a` and `State s a` monads, the ordering isn't particularly important; they both produce the same results (a reader that produces a stateful computation of type `State s a` is the same as a stateful computation that produces a reader of type `Reader e (s, a)` when fully evaluated - as the environment has no effect on the

[11] Initially 'ErrorT' was used, but due to depreciation 'ExceptT' was used instead - [10]

final result).
Out of personal preference I put the `Reader e a` monad on the bottom of the stack.[12]

The Parser a monad

```
type Parser a = ExceptT SyntaxError (StateT Position (Reader Source)) a
```

18.4 Scanner

18.4.1 Relevant modules

`Angle.Scanner` defines the scanner.

18.4.2 What is a Scanner?

The scanner reads in individual characters from source and passes them to other components (namely the parser and/or lexer) to be converted to tokens. The scanner has to keep track of its position in source in order to be able to backtrack and/or provide contextual syntax errors.

18.4.3 The implementation

I decided to implement the scanner as two parts: a type representing the information that the scanner would require to run, which would be embedded into the `Parser a` monad; and the base-most function for the parser-combinator functionality.

As a function As a function, the scanner is represented by `scanChar`, defined in *Angle.Scanner*. `scanChar` has the simple type `Parser Char`, it is a parser that either produces a character as the result, or fails. Another description of `scanChar` might be: *'the parser for a grammar in which any character is valid, with the only invalid token being the empty string'*. This is very useful as it provides the most simple grammar on top of which all the other parsing functions can be built, by refining the grammar from 'any character' to a set of characters in sequence.
The `scanChar` function does have a couple of other duties, such as explicitly updating the state (See below), and checking for special characters such as newlines.

As a type The `ScanState` datatype defined in *Angle.Scanner* represents the information required for the scanner to look ahead, backtrack and present positional information. a `ScanState` consists of three attributes: `sourcePos`, which is the current position in source;[13] `sourceRemaining`, which is the source text that has not yet been traversed; and `sourceScanned`, which represents the previously traversed source text.

[12] `SyntaxError`, `Position` and `Source` are not the actual names of the types used. See *Angle.Scanner* for the actual definition.
[13] See the `SourcePos` type in *Angle.Scanner*.

This information is then used as the state for the `Parser a` monad, and is updated by the scanner function `scanChar` during parsing.

18.5 Parsing Angle

The `Parser a` monad forms the basis of parsing in Angle, but with just the definition of the monad and the scanner, the only supported grammar is the most general one.

18.5.1 Layers

As mentioned in Section 18.2, the parser is split over several modules.

- `Angle.Scanner` defines the `Parser a` monad and the scanner functionality.
- `Angle.Parse.Helpers` builds upon the functionality provided by `Angle.Scanner` to create the set of parser-combinator functions.
- `Angle.Parse.Token` defines primitive parsers for Angle's basic structures.
- `Angle.Parse.Parser` defines the Angle parser.

Having a 'Parser' type alone doesn't automatically allow the parsing of Angle syntax. Angle must be described in Haskell's type system and functions must be defined to parse each of the constructs.

18.5.2 Implementing strings - an example

As an example of the process of defining a new Angle feature, I will use string literals.

Functions from `Angle.Parse.Helpers`:

`char :: Char -> Parser Char` parses the specified character.

`manyTill :: Parser b -> Parser a -> Parser [a]` a higher-order parser that parses until the first parser is satisfied.

`anyChar :: Parser Char` essentially the same as the base scanner function - parses any character.

Using these functions means that a parser could be defined in `Angle.Parse.Token` for parsing a basic string.[14]

```
string :: Parser String
string = char '"' *> manyTill (char '"') anyChar
```

[14]There are a lot of edge cases when parsing strings, see `Angle.Parse.Token.tokString` for a better representation.

Which would parse some text surrounded by double quotes.
Then, assuming the type

```
data LangLit = ...
             | LitStr String
             | ...
```

defined in `Angle.Types.Lang`, a parser can be implemented in `Angle.Parse.Parser` for wrapping a Haskell string in an Angle string.

```
litStr :: Parser LangLit
litStr = liftM LitStr tokString
```

Then this process is repeated for any other structures that need to be parsed.

19 Design choices

This section outlines certain design choices, why they were made and their impact upon the language.

19.1 Lists as both expressions and literals

19.1.1 The problem

Angle has few types Angle provides relatively few data-types compared to many other languages, thus each one has to take on the roles that would potentially be carried out by several data-types in other languages.

Storing data Oftentimes a programmer may wish to dump large amounts of data in a file, then read this back at a later date. Angle's lists are the obvious candidate for this scenario.

Lists must be expressions Angle's lists must be expressions on some level, otherwise it would not be possible to have lists containing values that are unknown before runtime (e.g. in `[1, x, 4]`, the value of `x` is unknown until it is evaluated).
Angle's method of dealing with this is to wrap every element of a list in an expression during the AST generation. The main issue here is that every time a list needs to be evaluated, each of the elements needs to be evaluated first.

19.1.2 The solution

The solution is fairly simple - have lists as both expressions and literals.

Literals Lists have a literal read syntax - when parsing a program, every time a list is encountered an attempt is made to read it as a literal - this can only succeed if all the contained values are literals as well.
If a list can be read as a literal, then the whole list can be directly evaluated without needing to first evaluate each of the elements, thus reducing computational complexity especially when dealing with very large lists.
If the list cannot be read as a literal then each of the elements is wrapped in an expression, as before.

19.2 Variadic operators

Angle's prefix variadic operators are quite unconventional. They are based upon a Lisp style, but are very different from operators in most languages.

19.2.1 The problem

Initially I intended to implement binary infix operators, the same as many other languages. I soon encountered the issue of precedence with my parsing library and all the workarounds for this were ugly and verbose.

19.2.2 The solution

The solution was to have the programmer explicitly state the order in which operations would be executed.
For example: with binary operators, `1 + 2 * 5 / 7` becomes `1 + (2 * 5 / 7)`, which then becomes `(1 + (2 * (5 / 7)))` because even though multiplication and division generally have the same precedence, they are often defined to have left or right associativity that determine how equal precedence is resolved (left in this case). However, in Angle, an equal statement would be `(+ 1 (* 2 (/ 5 7)))`, thus there cannot be precedence clashes due to each layer needing to be evaluated before the higher layers.

19.2.3 Variadic operators are versatile

Although not my initial choice for operators, the variadic operators have proved to be very powerful.[15]

Flat operations Conventional binary operators only have two operands, thus when an operation needs to be applied to a set of values, one might have to do the following: `a + b + c + ... + x + y + z`, this is inconvenient, and due to most of Angle's operators being variadic, all these operations could be combined into one: `(+ a b c ... x y z)`.

[15] Section 14.1.5 outlines some prominent features.

Part IV
Conclusion

Angle satisfies its initial design requirement as a general-purpose programming language. It supports most major language features, including subroutines, variables, looping and conditional structures, exception handling and file IO.
For small projects, the standard structures, along with the support for input and output to files, as well as direct calls to shell commands, means that most use-cases should be handled directly.
For larger projects, the ability to embed code from other files through built-in functions should encourage the use of multiple source files in a single project, and the creation of libraries to reduce code duplication.
Regarding the internal structure: the design of Angle allows for new syntax, built-in functions, types and language structures to be defined relatively easily. This extensibility means that creating new language features in the future is a definite possibility.
The documentation for the source code is satisfactory, and the use of Haddock means that this documentation may be displayed in a user-friendly manner.
The use of Cabal has greatly sped up development of Angle, as changes to directory structure need only be updated in the cabal file, and cabal's support for benchmarks and testing aided in the development work-flow.

20 Implementation

As is often the case, with hindsight I am aware of areas of the implementation that may have been improved if certain knowledge was available at the beginning of the project. For example, Haskell supports Generalized Algebraic Datatypes [25] - a system that allows you to explicitly state type signatures for type constructors. Knowledge of this when starting the project may have made it easier and cleaner to come up with a representation for the language structures.

In the early stages of the project there were some issues with the testing frameworks - the sizes of the test-cases that QuickCheck was generating were too large to be completed in a reasonable time. With `test-framework`, the initial testing framework, I had great difficulty controlling the size of individual test-cases. After switching to `tasty` these issues were more easily dealt with.
For future projects I shall have to do more research into testing libraries to ensure they can cope with all my projects' needs before commencing on the actual project itself.

20.1 Errors and debugging

An important part of any language is its ability to convey error messages to the programmer, and provide support for debugging the software.

This was a design consideration from quite early in the project, and I feel that Angle's error reporting system is satisfactory for the project level. Angle provides no separate debugging utilities, bar the interactive mode accessible through the software. In future projects I would encourage a more ingrained error system, with much more detailed messages - especially at the parsing stage.

20.2 Type system

My choice was to have a dynamic type system for Angle - this was done intentionally to reduce reliance on Haskell's type system in order to improve my understanding of type systems.
In the future, when this is not a goal in mind, I believe it would be wise to allow Haskell to enforce type-correctness in any small language implementations. This would not only reduce the required effort, but also perhaps provide some speed increases and allow me to better understand how to use existing type-systems to my advantage.

21 Final comments

After having now created a basic programming language, I feel I have gained enough knowledge to study the topic more in-depth and implement more small languages. I will perhaps revisit Angle in the future to implement additional features and review how my methods have changed.

References

[1] Ross Paterson Andy Gill. transformers: Concrete functor and monad transformers. `https://hackage.haskell.org/package/transformers`.

[2] base: Basic libraries. `https://hackage.haskell.org/package/base-4.8.1.0/docs/Data-Char.html#t:Char`. Data.Char.

[3] base: Basic libraries. `http://hackage.haskell.org/package/base-4.8.1.0/docs/Prelude.html#t:Double`. Prelude.

[4] base: Basic libraries. `http://hackage.haskell.org/package/base-4.8.1.0/docs/System-IO-Error.html`. System.IO.Error.

[5] Roman Cheplyaka. tasty: Modern and extensible testing framework. `https://documentup.com/feuerbach/tasty`.

[6] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.

[7] Lars Marius Garshol. BNF and EBNF: What are they and how do they work? `http://www.garshol.priv.no/download/text/bnf.html#id1.2`.

[8] Andy Gill. mtl: Monad classes, using functional dependencies. https://hackage.haskell.org/package/mtl-1.1.0.2/docs/Control-Monad-State-Lazy.html.

[9] Andy Gill. mtl: Monad classes, using functional dependencies. https://hackage.haskell.org/package/mtl-1.1.0.2/docs/Control-Monad-Reader.html.

[10] Andy Gill. mtl: Monad classes, using functional dependencies. https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Except.html#t:ExceptT.

[11] The Haskell programming language. https://www.haskell.org/.

[12] Haskell platform. https://www.haskell.org/platform/contents.html.

[13] HaskellWiki. Functional programming. https://wiki.haskell.org/Functional_programming.

[14] Isaac Jones. The Haskell Cabal, a common architecture for building applications and libraries.

[15] Keunwoo Lee. Dynamic typing vs. static typing. http://courses.cs.washington.edu/courses/cse341/04wi/lectures/13-dynamic-vs-static-types.html, 2004.

[16] Simon Marlow. Haddock, a Haskell documentation tool. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Pittsburgh Pennsylvania, USA, October 2002. ACM Press.

[17] Simon Marlow and Simon Peyton Jones. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume 2*. Lulu, 2012.

[18] Dave Marshall. Literal values. https://www.cs.cf.ac.uk/Dave/Multimedia/node71.html.

[19] Matt Might. The language of languages. http://matt.might.net/articles/grammars-bnf-ebnf/.

[20] Microsoft Developer Network. Functional programming vs. imperative programming. https://msdn.microsoft.com/en-gb/library/bb669144.aspx.

[21] Bryan O'Sullivan. criterion: Robust, reliable performance measurement and analysis. http://www.serpentine.com/criterion/.

[22] Bryan O'Sullivan, John Goerzen, and Don Stewart. Monad transformers. In *Real World Haskell*, chapter 8. O'Reilly Media, Inc., 1st edition, 2008.

[23] Loyola Marymount University Ray Toal. Programming paradigms. http://cs.lmu.edu/~ray/notes/paradigms/.

[24] Vanguard Software. Compiled vs. interpreted languages. http://www.vanguardsw.com/dphelp4/dph00296.htm.

[25] The GHC Team. GHC language features. In *The Glorious Glasgow Haskell Compilation System User's Guide*, chapter 7.

[26] Linus Torvalds. Git - the stupid content tracker. https://git-scm.com/.

[27] Eric Walkingshaw. Monads. http://web.engr.oregonstate.edu/~walkiner/teaching/cs583-fa14/slides/6.Monads.pdf.

[28] C2 Wiki. Dynamic typing. http://c2.com/cgi/wiki?DynamicTyping.

[29] C2 Wiki. Weakly typed. http://c2.com/cgi/wiki?WeaklyTyped.

[30] Wikibooks. Haskell/monad transformers. https://en.wikibooks.org/wiki/Haskell/Monad_transformers.