Pie
Language Specification/RLO
Version 0.3

Jason Bell – 3078931

April 2, 2017

Comp 601

Professor: Dr. Huntrods

Foreword

For my undergraduate major project, I implemented a custom programming language and compiler: named Pie as a nod to Python as inspiration, and as whimsy. The first two versions of the language were built on the .NET runtime and were designed with unit testing and test coverage analysis as language features. I did this to learn about, and assess the efficacy of, test driven development.

While I am eager to continue this project over the course of my graduate degree, I am painfully aware that the graveyards are full of dead programming languages. What features can Pie have to make it stand out from the crowd? My previous two assignments for this course investigated this question.

The first assignment surveyed the history of popular programming languages and the features that made them so. My conclusions were as follows:

- Popular languages were always experimental.
- Popular languages were at times regressive: C appeared as a backlash against the English language syntaxes that preceded it.
- A new language must introduce something new: for example, although D is a beautiful language, I would argue that it has not seen wide adoption because while it does what it does well, it doesn't do anything new.

My second assignment continued this study by investigating current popular programming languages, and future needs. The two most important conclusions were:

- Fortran, despite being one of the oldest programming languages, is still widely used for scientific and numerical computing; including on supercomputers. This is because it is highly optimizable, easy to work with, and has excellent support for distributed data and computing.
- Any future languages MUST have excellent support for these as well.

This paper is a preliminary proposal for a more final version of the Pie language: versions 0.1 and 0.2 were essentially proofs of concept. I wrote it as a language specification, which is about as reusable learning object as one can get: this is a first version of the Pie RLO. However, it should be noted that for comparison, the C# language specification is 400 pages: there is a lot more work to do.

I appreciate any feedback, critical or otherwise. Nothing in this document is carved in stone yet!

Table of Contents

1.Introduction

Pie is a dynamic type, ahead-of-time compiled, programming language inspired by Python, but with syntax shared with the C family of languages. The language is designed to be general purpose, and with the demands of future technologies in mind. Pie supports multiple programming paradigms: procedural, functional, and object-oriented.

Pie is designed to be modular, customizable, and extensible. The ability to add new constructs to the language is provided, and the quality of this system is ensured by building the entire language compiler with that system. Three memory management options are provided: none, reference counted, and mark-and-sweep garbage collection. Ability to interoperate with the Java, Python, and C# languages is provided through plugins.

Pie supports unit testing and test coverage analysis as language features. Unit tests may exist alongside the code being tested, or in a separate project. The compiler is able to perform test coverage analysis by inserting sensors into a test build, that are activated if covered by a test. The percentage of activated sensors gives a percentage coverage, and the locations of inactive sensors indicate areas of code not currently tested.

Pie supports Fortran-style distributed code and data. Constructs are provided for transparent concurrent processing: the user should not care if the code is running on a single CPU or many. Distributed data is similarly transparent.

Pie is a sophisticated, but simpler to use, façade on top of the C++ programming language. The Pie compiler utilizes C++ as a bridging language: Pie is translated to C++ code conforming to the C++11 standard [1]. This has the advantages of leveraging the wide suite of existing C++ libraries, the performance and power of C++, and allows embedding of C++ code inline within Pie code. The latter advantage confers three further benefits: performance critical code can be offloaded from dynamic type Pie to static type C++, Pie code can invoke C++ libraries without having to write glue code, and it provides the ability to interoperate with any other language that is invokable from C++.

1.1 Hello, World

The “hello world” program traditionally provided for programming languages is the following in Pie:

```
// import the std.io namespace, wherein is found the cout class

using std.io;

main() {
        // Stream text to the console, ending with a newline
        cout <- “Hello, world” <- endl;
}
```

This program starts with a using directive that imports the std.io namespace of the Pie standard library. The entry point of the program, the main() procedure, is declared as in C++, without a return type specified: this specifies that there is no return value: equivalent to the void keyword in a C++ procedure.

The line of code in the procedure body outputs the string “Hello, world” to the console, followed by a newline character appropriate for the current platform. Those familiar with C++ will recognize the standard output method of C++, which the Pie standard library thinly wraps. Pie provides the “streaming operators” <- and ->, rather than obfuscating existing operators with functionality for which they were not intended.

1.2 Program Structure

The most basic component of a Pie program is the procedure, which performs operations on input parameters and may or may not produce a return result. Procedures that do not return a type are demonstrated in the previous “hello, world” example. A procedure that accepts inputs and returns a value is defined as:

```
using std.io;

// Return the square of x
sqr(x) = {
        return x * x;
}

main() {
        cout <- sqr(3) <- endl;
}
```

Note that the procedure declaration for sqr(x) ends with an equals sign: this indicates that the procedure returns a value. Parameters for the procedure do not have a type defined, as Pie is dynamic type.

A function is similar to a procedure, but has two constraints: a function must return a value, and can only perform operations on the input parameters and variables declared within its scope. This is a fundamental concept of functional programming, and minimizes the risk of side effects: if a function is not producing a correct result, the problem is limited to either the function, or the inputs. There is no risk of an outside variable affecting the function result. A function is declared similarly to a procedure, but with := instead of =.

```
using std.io;

// Declare a variable with the var keyword
var x = 3;

sqr1() := {
        // Compiler error: function is accessing a variable outside the function scope
        return x * x;
}

sqr2(y) := { } // Compiler error: function does not return a value

sqr3(y) := {
        return y * y;
}
```

In this example, x is declared as a global variable. The line of code in the sqr1 function generates a compiler error, as it attempts to use x despite it existing outside of the scope of the function. The sqr2 function generates a compiler error because it does not return a value. The sqr3 function does not generate an error because it only acts on the inputs, and returns a value.

Lambda expressions are closures: inline functions or procedures capable of capturing runtime data:

```
using std.io;

// Create a lambda procedure expression that returns the square of x
var sqr = (x) => { return (x) => { return x * x; } }

main() {
        // Generate a new procedure that returns the square of 3
        var sqr3 = sqr(3);
        cout <- sqr3(); << endl;
}
```

This example generates a lambda expression, sqr, and uses it to generate a procedure that produces the square of 3, without having to input a 3. Because the lambda expression is declared with the => operator, it acts as a procedure and is able to access variables outside of its scope.

If a lambda expression is declared with :=>, it generates a function, which can not access variables outside of its scope.

```
using std.io;

// Create a lambda function expression that returns the square of x
var sqr = (x) :=> { return (x) :=> { return x * x; } }

main() {
        // Create a new function that returns the square of 3
        var sqr3 = sqr(3);
        cout <- sqr3(); << endl;
}
```

Lambda expressions may be combined to form new functions or procedures:

```
using std.io;

var add = (x, y) :=> { return x + y; }

var sqr = (x) :=> ( return x * x; )

main() {
        var n = sqr(add);
        cout <- n(2, 3) <- endl; // output sqr(2 + 3)
}
```

While Pie is a dynamic type language, class declarations are still available for object oriented programming. In a dynamic type context, a class declaration defines an object’s initial state, and that initial state is immutable. New members may be added at runtime, but the members declared at compile time represent an immutable contract. A Pie class is a C++ class, but derived from a base class that provides the infrastructure necessary for dynamic types.

```
using std.io;

// Just a class that writes to the console
class DeliciousPie {
        eat() {
                cout <- “nom!” <- endl;
        }
}

main() {
        var pie = DeliciousPie();
        pie.eat();                                      // “nom!”
        var n = () => { cout <- “delicious!” <- endl; }
        pie.eat = n();                                  // Compiler error
        pie.n = n();
        pie.n();                                        // “delicious!”
}
```

In this example, a class is declared with a public member procedure: unlike in C++, class procedures and functions default to public accessibility. Class variables still default to private. Once an instance of the class is created, that member procedure can be invoked. Trying to reassign the member procedure with a new one, in this case from a lambda, generates a compiler error: the original contract as defined by the class is immutable. Instead, a new member can be added with a different name.

1.3 Numeric Types

All Pie types derive from the same base dynamic type class, which provides the infrastructure necessary for dynamic types. This includes numeric types, which are provided by the standard library. These numeric types can be instantiated by passing a literal or numeric variable to the constructor of that type, or through assignment of a literal:

```
using std;

var n1 = Int32(1234);
var n2 = 1234;
```

As it is built upon C++, Pie uses the same ranges and precisions for numeric types as defined by the C++ implementation [2]:

Numeric Type	Bits	Class Name	Approximate range/precision – actual values are platform specific	Literal Notation
Boolean	8	std.Boolean	0..1	true or false
Unsigned Byte	8	std.Byte	0..255	255b
Unsigned Short Integer	16	std.UInt16	0...65,535	1000us
Unsigned Integer	32	std.UInt32	0...4,294,967,295	1000u
Unsigned Long Integer	64	std.UInt64	0...18,446,744,073,709,551,615	1000ul
Signed Byte	8	std.SByte	-128 to 127	100sb
Signed Short Integer	16	std.Int16	–32,768...32,767	1000s
Signed Integer	32	std.Int32	–2,147,483,648...2,147,483,647	1000
Signed Long Integer	64	std.Int64	–9,223,372,036,854,775,808... 9,223,372,036,854,775,807	1000L
Float	32	std.Float	1.5×10^{-45} to 3.4×10^{38}, 7-digit precision	1000.0f
Double	64	std.Double	5.0×10^{-324} to 1.7×10^{308}, 15-digit precision	1000.0
Pointer	Platform word size	std.Pointer	Platform word range	100p

1.4 Variable Declaration and Pointers

As with C++, variables may be declared and allocated on the stack, inline in a class, or on the heap. Variables are declared with the var keyword, and declaring a variable name a second time in the same scope, or a nested scope, will generate a compiler error:

```
var n = 2;

n = 3;                  // OK
```

```
var n = 4;              // Compiler error
```

As with the C++ new keyword, variables to be allocated on the heap are declared with the ptr keyword, and the data type of the variable is a pointer. Unlike in C++, this is not a simple word primitive: it is of the std.Pointer type, and wraps the actual pointer. This is similar to how Fortran handles pointers: a variable is either a pointer, or it is not [3].

```
// n is a std.Pointer that points to a std.Int32 on the heap

var n = ptr std.Int32(1234);
```

Abstracting the actual pointer in this way is how memory management occurs: the std.Pointer class provides hooks necessary for the memory management system being used. In this way, Pie pointers behave similarly to C++ smart pointers [4].

Deletion of a variable on the heap is done as in C++, with the delete keyword:

```
var n = ptr std.Int32(1234);

delete n;
```

This executes the destructor for the data type and notifies the memory manager, if enabled. Variables will also be automatically deleted from the heap if a memory manager is enabled, and no more pointers to the variable exist.

A pointer to an existing variable can be declared similarly:

```
var n = 1234;

var p = std.Pointer(n);
```

The variable p is a pointer to the integer variable n. This also demonstrates declaration of a variable on a stack: both n and p exist in the stack of their scope, and will be automatically deallocated when that scope exits. A class definition also counts as a scope: a variable declared inline in a class will also be deallocated when the class is deleted. A pointer to an existing variable can also be declared as in C++, with the & operator. Be mindful, however, that this will not return a C++ pointer, but a Pie std.Pointer:

```
var n = 1234;

var p = &n;
```

The std.Pointer class allows access to the internal pointer in two ways: direct and locking. Both of these return the internal pointer for use in embedded C++ or external libraries. Direct access to the raw pointer is provided through a ptr() method:

```
var n = ptr std.Int32(1234);

var p = n.ptr();
```

This method will generate a runtime error if a memory manager is enabled: a memory manager can potentially alter the internal pointer at any time, so the consistency of the address can not be guaranteed. Thus, this method should only be used if memory management is disabled. If a memory manager is enabled, the internal pointer must be locked for the duration over which it is used, so that the memory manager does not alter it:

```
var n = ptr std.Int32(1234);
var p = n.lockPtr();

// Do things with the internal pointer
p.unlockPtr();
```

Members of the variable pointed to can be accessed through the pointer transparently. Normally, members of a dynamic type object are stored in a hash map, which is inspected when a member is invoked. The std.Pointer class still does this with the ptr, lockPtr, and unlockPtr methods, but overrides this behavior for any others. Instead of inspecting its own internal hashmap, it simply forwards the invocation request to the variable it points to.

```
var foo = Foo();

foo.bar();                 // Invokes the bar() member of foo, as normal

var p = &foo;

p.bar();                   // Delegates the invocation of bar() to the foo it points to
```

1.5 Operators and Expressions

Operators define expressions as combinations of operands and operators. Operands can be any data type: whether stored on the stack or the heap, so long as the data type supports the operator. The supported operators, in decreasing order of precedence, are:

Expression	Operation
foo.bar	Member access
bar()	Procedure, function, and lambda invocation, and inline or stack variable declaration
bar() { … }	Stack variable declaration with initializer
bar[]	Indexer access
n++	Post increment
n--	Post decrement
ptr foo()	Heap variable declaration
ptr foo() { … }	Heap variable declaration with initializer
typeof x	Return a std.Type describing the variable's class
-x	Negation
!x	Logical negation
~x	Bitwise negation
++x	Pre increment
--x	Post increment
X * y	Multiplicative
X / y	Division
X % y	Modulus
X + y	Additive
X – y	Division
X << y	Bitwise shift left
X >> y	Bitwise shift right
X < y	Less than conditional
X > y	Greater than conditional
X <= y	Less than or equal conditional
X >= y	Greater than or equal conditional
X == y	Equal conditional
X != y	Not equal conditional
X & y	Bitwise or logical AND
X ^ y	Bitwise or logical XOR
X | y	Bitwise or logical OR
X && y	Logical AND conditional
X || y	Logical OR conditional

X = y	Assignment
*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, !=	Compound assignment with operators
() => {}	Lambda procedure
() :=> {}	Lambda function
<- ->	Streaming operators

1.6 Statements

Statements use expressions, or several expressions, to define program actions. The simplest, variable declaration statements, make use of the var keyword and assignment operator. Use of a compound assignment operator generates a compiler error:

```
var n = 1234;            // Valid variable declaration statement

var n2 += 1234;          // Compiler error
```

Expression statements are the second simplest, simply being one or two operands and an operator:

```
foo.bar();               // Member invocation statement
```

Selection statements choose between different sets of statements, based on a conditional. This includes if and switch statements, both of which behave similarly to C++:

```
If(condition) {

        // Do things if true

}

Else {

        // Do things if false

}


switch(n) {

        case 0:

                // Do stuff if n is 0

                break;

        case 1:

                // Do stuff if n is 1

                break;

        default:

                // Do stuff if n is anything else

                break;

}
```

Iterative statements repeatedly execute one or more statements, based upon either a condition, or by members of a list. The do and while statements keep running depending on a condition, and behave the same as in C++, as does the for loop when using a condition. When iterating over the members of a list, the for statement has behavior not found in C++.

```
// A while loop

var n = 0;

while(n < 10) {

        n++

}


// A do/while loop

var n = 0;

do {

        N++;

} while (n < 10)


// A conditional for loop

for(n = 0; n < 10; n++) {

}


// An iterative for loop

for(var item : list) {

}
```

In the context of iterative statements, the continue and break keywords work identically to C++: break immediately exits the loop, while continue jumps to the next iteration of the loop.

The return statement immediately exits the current procedure, function, or lambda, and returns the variable that follows it, if appropriate:

```
n() = {

        return 1234;

}
```

Exception handling statements behave nearly identically to C++: they use that languages exception handling capability [5]. Exception handling differs in syntax: while Pie is dynamic type, it is useful to distinguish between the types of exceptions. This is done using a selection statement which behaves similarly to the switch statement:

```
try {

        // Generate a C++ exception

        throw “This is an exception!”;

}

catch ( e ) {

        // Handle differently depending on what kind of exception it is

        case std.String:

                cout << e1 << endl;
```

```
                        break;
                case SomeException:
                        throw e;
                        break;
                default:
                        cout << “Unknown exception.” << endl;
                        break;
        }
```

1.7 Lists and Dictionaries

Lists and dictionaries are deliberately similar to those found in both Javascript and Python: both languages are dynamic type and have the same syntax for both lists and dictionaries [6][7][8]. This greatly facilitates data passing between the languages: for example, inserting string representations of Python backend data into Javascript frontend. As a high-level language, Pie does not support arrays. If they are required, they should be declared in an embedded C++ block.

Lists are declared as:

```
        var list = [“One”, “Two”, 1234];
```

And dictionaries are declared as:

```
        var dictionary = {“One”: 1, “Two” : 2, “Three” : 3};
```

Both lists and dictionaries can be created on the heap with the ptr keyword.

2.Object-Oriented Paradigm

2.1 Member Functions and Procedures

The object-oriented paradigm supported by Pie is much the same as in C++, aside from being dynamic type. A Pie class has a type; an immutable one described by the class declaration, but new members may be added at runtime. Immutable members declared in the class declaration can not be replaced. Because Pie is not concerned with an object's type, it is also duck typed: if an object has the members needed for an operation, it can be used for that operation (if it looks like a duck, and quacks like a duck, it's a duck).

```
class Duck {
        quack() {
                cout <- “Quack!” <- endl;
        }
}
class Dog {
        bark() {
                cout <- “Bark!” <- endl;
        }
}

// This procedure doesn’t care what its input is, so long as it quacks like a duck
quackLikeADuck(duck) {
        duck.quack();
}
main() {
        var duck = Duck();
        quackLikeADuck(duck);                           // Works fine
        var dog = Dog();
        quackLikeADuck(dog);                            // Runtime error: dogs don’t quack
        dog.quack = () => { cout <- “Quack!” <- endl; }
        quackLikeADuck(dog);                            // Works fine: the dog can quack
}
```

As in C++, Pie supports multiple inheritance, as opposed to single inheritances and interfaces. This potentially leads to the “diamond problem”: if more than one base class implement a member, which implementation does the child class inherit? The solution used in Pie is to place precedence in the order of inheritance: the implementation first found in the list of inherited classes is the one used, and is not overwritten by following implementations. For simplicity, only public inheritance, as defined by C++, is used.

```
// Declare two parent classes that both have a doSomething() procedure

class Parent1 {

        doSomething() {

                cout <- “Parent1” <- endl;

        }

}

class Parent2 {

        doSomething() {

                cout <- “Parent2” <- endl;

        }

}


// Create a child class that inherits both, but only the doSomeThing() from Parent1

class Child : Parent1, Parent2 {

}

main() {

        var child = Child()

        child.doSomething();    // Outputs “Parent1”

}
```

Pie does not support virtual, abstract, or override keywords, as these concepts are meaningless in the context of dynamic and duck typing. In static type languages, “virtual” functions are stored in hash maps and are late bound. In dynamic type languages, this is the case for all class members.

Unlike Python, Pie supports overloading of procedures and functions in a class: class members are stored using their entire signature, not just their name. However, overloaded members must have different numbers of arguments, as in the absence of type information they can not be distinguished by argument types.

```
class Foo {

        bar(x) {}

        bar(y) {}               // Compiler error: indistinguishable from bar(x)

        bar(x, y) {}            // OK

};
```

As in C++, the object instance is accessed via the this keyword. Unlike in Python, this does not need to be in the argument list.

```
class Foo {

        var x;
        bar() {
                // Set the x variable in this instance
                this.x = 1234;
        }
}
```

Static members, which are shared by all instances of a class, are declared with the static keyword and accessed through the class name:

```
class Foo {
        static var x;
        static setX(x) {
                Foo.x = x;
        }
}
```

2.2 Constructors and Destructors

Constructors and destructors are declared with the same syntax as C++:

```
class Foo {
        Foo() {         // Constructor
        }
        ~Foo() {        // Destructor
        }
}
```

Constructors of superclasses are invoked in the same manner as in C++. Base destructors do not need to be invoked, as they are automatically invoked in reverse order in C++:

```
class Child : Parent1, Parent2 {
        Child() : Parent1(), Parent2() {
        }
}
```

2.3 Access Modifiers

Member accessibility can be public, private, protected, or internal, and combined with the static keyword. For conciseness, member functions and procedures default to public accessibility, while variables default to private. Public members can be accessed from anywhere, whereas private members can only be accessed by the class it is within; it can not be accessed by child classes. Protected members can be accessed by the class it is within and by child classes. Internal members can be accessed from anywhere within the same project.

```
class Foo {

        var A;

        public var B;

        protected var C;

        private var D;

        internal var E;

}
```

2.4 Polymorphism

In static type languages, polymorphism is the ability of an object to be cast to other classes. By this strict a definition, Pie does not support polymorphism. However, as a dynamic and duck typed language, it is inherently polymorphic: if an object meets the requirements, it can be treated as another object type without casting.

3.Functional Programming Paradigm

3.1 Pure Functions

Pie functions are pure: they are only allowed to perform operations on variables passed them as arguments. This eliminates possible "side effects":

- Invoking a function with the same input arguments will always produce the same result, which allows for memoization: storing of previously calculated values for quick retrieval rather than recalculation.
- As long as the inputs are not used by other functions at the same time, the function is thread-safe.

3.2 First Class and Higher-Order Functions

Pie functions are first-class: they can be used are arguments in other functions. This allows them to be treated as variables, and passing of different functions to alter the behavior of another function. This also means that a function may return another function.

A higher-order function is the result of using at least one function as an argument in another. A classic example is a list sorting function that accepts a list and sorting function as arguments: the list contains the items to be sorted, and the function determines how they are to be sorted.

```
var list = [1, 3, 2, 9, 5];

list.sort((x, y) :=> { x > y });         // Sort the list by comparing the two numbers
```

This technically does not meet the second requirement of higher order functions: that they return a new function. Pie does not require this, but does allow it:

```
var addop = (x, y) :=> { x + y };        // A simple addition lambda

var subop = (x, y) :=> { x - y };        // A simple subtraction lambda

// A high level function capable of incorporating another

var math = (op, x, y) :=> { return (x, y) :=> { op(x, y); } };

// Generate an add function by combining the math and addop lambdas

var add = math(addop, x, y);

cout <- add(3, 4) <- endl;

// Generate a subtract function by combining the math and subop lambdas

var sub = math(subop, x, y);

cout <- sub(3, 4) <- endl;
```

4.Parallelization

4.1 Nodes

Pie borrows heavily from Fortran in allowing definition of an arbitrary number of processors [9], although in Pie the are referred to as nodes to represent their ownership of memory as well. The precise nature of these nodes is platform specific: on an embedded device with a single CPU any nodes beyond the first will be entirely virtual, and therefore will not provide any performance advantage. If the device has 4 CPUs, and 8 nodes are created, 4 will map to real CPUs, and the other 4 will be virtual. In a cloud, the nodes will map to remote CPUs. A list of nodes is generated with the node keyword and an indexer:

```
var nodes = node[20];           // Generate 20 nodes
```

4.2 Threads

The thread keyword executes a function, procedure, or lambda in its own thread. Whether this thread is virtual, on another CPU on the same computer, or on a remote node in a cloud or supercomputer, is transparent to the user and platform dependent. If a node is not specified, the thread is run on the least busy node. How this node is chosen is determined by platform specific load balancing algorithms.

```
var out = (x) => { cout <- x <- endl; }

for(i = 0; i < 100; i++)

        thread out(i);                  // Execute out() in its own thread
```

This will output the numbers from 0 to 99, each in its own thread: this example code will almost certainly generate race conditions. As well, note that although each loop spawns a thread, the loops are still executed in serial order rather than in parallel.

Note that if a function is executed in a thread, greater thread safety is ensured than would be the case with a procedure, as a function may only act upon its inputs.

Multiple statements can follow the thread keyword. If combined with a for loop, this is similar to Fortran's DO CONCURRENT statement [10]:

```
for(i = 0; i < 100; i++) thread {

        // Multiple statements

}
```

The thread keyword returns a std.Thread object that provides more advanced capabilities. The join keyword halts execution of the current code until the thread being joined completes:

```
var t = thread doThings();

join t;
```

Alternatively:

```
join thread doThings();
```

The abort keyword will trigger an exception in the running thread, and force it to end its operation:

```
var t = thread doThings();

abort t;
```

The stop keyword stops, but does not terminate, a thread. The start keyword resumes execution for a stopped thread:

```
var t = thread doThings();

stop t;

start t;
```

A thread object can be used as a conditional: if it evaluates to true, it is currently running. If it evaluates to false, it is not running:

```
while(thread doThings()) {

        cout <- “Thread is running.” <- endl;

}

cout <- “Thread is not running.” <- endl;
```

A specific node can be selected for a thread by using the threadto keyword instead. This forces the thread to run on the requested node, rather than allowing the node to be chosen by the platform specific load balancing algorithm.

```
var ns = node[20]                         // Generate 20 nodes

var t = threadto ns[15] doThings();       // Run doThings in a thread on node 15
```

4.3 Concurrency Control

The simplest form of concurrency control is with the lock keyword that provides a mutex. Threads are mutually excluded from entering the critical section bound by the block at the same time. The lock keyword may be used alone to create a single lock that applies to all threads:

```
doSomething() {

        // Unlocked section: all threads may access in parallel

        lock {

                // Locked critical section: only one thread may access at once

        }

        // Unlocked section: all threads may access in parallel

}
```

Alternatively, the lock may be used on an object to allow more refined control:

```
doSomething(x) {

        // Unlocked section: all threads may access in parallel

        lock x {

                // Locked critical section: only one thread may use x at once

        }
        // Unlocked section: all threads may access in parallel
    }
```

More advanced forms of concurrency control such as semaphores are provided by the standard library than as language features.

4.4 Distributed Data

Pie data distribution is heavily inspired by that of Fortran [9].

Variables can be declared as distributed using the dist keyword; distributed variables are treated as on a heap rather than a stack. The dist returns a std.DistPointer that behaves similarly to a std.Pointer, except that it points to and gives access to data which may or may not be on a remote node. If a node is not specified, an appropriate node is chosen with a platform specific load balancing algorithm:

```
    var data = dist Foo();                  // Create a Foo on a node
```

If the data is to be distributed to a specific node, use the distto keyword:

```
    var ns = node[20];

    var data = distto ns[15] Foo();         // Create a Foo on node 15
```

As a general case, when threads and data are distributed on nodes, a thread will only see and operate on data on the same node. For example, when iterating over a list, the current thread will only operate on those elements of the list that exist on the same node as the thread:

```
    var ns = node[20];

    var list = [];

    for(i = 0; i < 20; i++)

        list.add(distto ns[i] Int32(i));        // Add numbers 0-19, each on a different node

    for(n : ns) threadto n                      // Run each loop on one of the nodes

        for(i : list)

            // Output the number on each node: each thread will only output one number

            cout <- i <- endl;
```

5.Unit Testing

5.1 Unit Tests

Pie supports unit tests as a language feature and are deliberately simple for ease of use. Unit tests can be defined anywhere a procedure or function can be: whether to place the test alongside the tested code, or in a separate project, is entirely user choice. A unit test is declared with the test keyword, and if it completes without an exception being thrown is considered to have passed:

```
test passingTest {
        var n = 3 / 2;          // 3 is divisible by 2: no problem
}
test failingTest {
        var n = 1 / 0;          // Divide by zero exception
}
```

The assert keyword tests a condition. If the condition resolves to true, the unit test continues. If the condition resolves to false, an exception is thrown and the test fails:

```
test passingTest {
        var n = 3 + 2;
        assert(n == 5);         // Succeeds: n == 5
}
test failingTest {
        var n = 3 + 2;
        assert(n == 6);         // Fails: n == 5
}
```

5.2 Test Coverage Analysis

Compiler settings allow generation of a test build of the project, upon which test coverage analysis can be performed. This test build has sensors inserted at every fork in the code, including:

- If and else statements
- Switch statements
- Loops

If a unit test covers a block of code, sensors found within that block will be activated. Once all tests have run, the percentage test coverage is simply the number of activated tests divided by the total number of tests: 0.0 indicates that the tests do not cover any code, while 1.0 indicates that it covers all of it.

```
testedFn(x) {
        // Create an either/or scenario to demonstrate test coverage analysis
        if(x) {
                return true;
        } else {
                return false;
        }
}
test t1 {
        // Only the if block is executed, not the else: 50% coverage
        assert(testedFn(true));
}
test t2 {
        // Both the if and else blocks are executed: 100% coverage
        assert(testedFn(true));
        assert(!testedFun(false));
}
```

6.Embedded C++

6.1 C++ Code blocks

Because Pie is built on top of C++, this allows one to embed C++ code within Pie code. This contrasts with other languages, such as Cython, that can compile C++, but only if it's in a separate file and bridging code is implemented [11]. A block of C++ code is declared with the cpp keyword, which can be inserted anywhere in Pie code:

```
cpp {
        #include <iostream>
}

main() {
        cpp {
                std::cout<< "Hello, world!" << std::endl;
        }
}
```

Pie is a dynamic type language, which means that virtually every action is late bound. This has serious performance consequences as, for example, invoking a member of a class always involves inspection of a hash table. Allowing embedded C++ code enables offloading of performance critical code from Pie to C++. Consider for example the recursive Fibonacci algorithm, implemented in a C++ code block:

```
cpp {
        int fib(int x) {
                if( x == 0 || x == 1)
                        return x;
                else return fib(x – 1) + fib(x – 2);
        }
}

main() {
        // Use the C++ Fibonacci function to get the values more speedily
        for(i = 0; i < 20; i++)
                cout <- Int32(fib(i)) <- endl;
}
```

Since Pie classes are just C++ classes, with extra infrastructure added to support dynamic typing, they are usable within C++ blocks. Unfortunately, because the . and -> operators of C++ can not be overridden, there is extra syntax required. For example, to invoke a member of a Pie class requires looking up the member with the appropriate hash table lookup function:

```
class Foo {

        bar() { cout <- "bar" <- endl; }

}

main() {

        var f = Foo();
        cpp {
                // Invoke the late-bound procedure, bar()
                f.invokeMethod(“bar”, NULL);
        }
}
```

The invoke method is provided by the base class for all Pie classes and invokes the method of the given name with the given arguments, in this case NULL indicates zero arguments.

6.2 Native Libraries

The second advantage of C++ blocks is enabling use of native libraries without using bridging code which is often buggy or poorly maintained. For example, in languages such as Java if one wishes to use OpenGL, one must either rely upon third party bindings or implement ones own. In a C++ block one can use the library directly:

```
cpp {
        #include <GL/glut.h>
        void draw() {
                // Clear the display and draw a box
                glClear(GL_COLOR_BUFFER_BIT);
                glBegin(GL_POLYGON);
                        glVertex3f(0.0, 0.0, 0.0);
                        glVertex3f(0.5, 0.0, 0.0);
                        glVertex3f(0.5, 0.5, 0.0);
                        glVertex3f(0.0, 0.5, 0.0);
                glEnd();
                glFlush();
        }
}
main() {
        cpp {
                // Setup and run GLUT
                glutInit();
                glutInitDisplayMode(GLUT_SINGLE);
                glutInitWindowSize(300, 300);
                glutInitWindowPosition(100, 100);
                glutCreateWindow("Hello world");
                glutDisplayFunc(draw);
                glutMainLoop();
        }
}
```

6.3 Assembly

The third advantage is the simplification of device driver development: if the C++ compiler supports Assembly, then so does Pie. Therefore, Pie provides the ability to embed assembly in a high level language. The version of Assembly is highly platform and compiler specific, but here is an example of a hello world application using Assembly and a Visual C++ compiler:

```
cpp {
        #include “stdio.h”
        char hello[] = “Hello world”;
}
main() {
        cpp {
                __asm {
                        mov eax, offset hello
                        push eax
                        call printf
                        pop ebx
                }
        }
}
```

7.Language Interoperability

7.1 Language Plugins

In the spirit of Pie's support for embedded C++, Pie also supports a plugin architecture that allows use of libraries from other high level languages. Any language that can be invoked from C++ can potentially have a plugin for Pie. For example, Python has excellent interoperability with C++ [12], Java can be invoked with JNI [13], and C# can be invoked by embedding the Mono runtime [14]. The syntax to import something from a language plugin is:

```
import random from Python;
```

This will import the random class from the Python plugin. Use of the import keyword, rather than the using keyword, makes it more explicit that it is being imported from an external source. Once the entity has been imported, it can be used as Pie object, insofar as their functionality and syntax is supportable.

```
import random from Python;

main() {

        random.seed(1234);

        for(i = 0; I < 20; i++)

                cout <- random.random() <- endl;         // Output 20 random numbers

}
```

This functionality is possible because Python is invokable from C++: when a member of a plugin object is invoked, rather than its hash table being inspected, the invocation is passed along to the bridging code for that language. Importing the random class provides access to a dynamic type project that wraps the Python object, and the invocations for seed() and random() are passed along to the Python object.

While the importation and use of these external objects is supported as a language level feature, the compiler arguments must still specify information about where to find the Python runtime and libraries; though this feature will be presented as clean and friendly project settings in an IDE.

7.2 Example (Hypothetical) Plugin Implementation

Follows a hypothetical, entirely unproven, example of implementation of a rudimentary Python plugin. The final product would be vastly more complicated than this. This example merely loads a brief Python script, and allows it to be invoked from Pie: in fact, this script is the one used for the simplest embedding example in the Python documentation [12].

The first step is a class that manages the Python bindings:

```
cpp { #include “Python.h” }

class Python {

        Python() {

                cpp {
                        Py_Initialize();
                }
        }
        ~Python() {
                cpp {
                        Py_Finalize();
                }
        }
        import (name) = {
                // Return a DatePrinter if requested, otherwise throw and exception
                if(name == “DatePrinter”)
                        return DatePrinter();
                else throw name + “ entity not found”;
        }
}

class DatePrinter {
        print() {
                cpp {
                        // Compile and run a Python script
                        PyRun_SimpleString(“from time import time, ctime\n”
                                “print ‘Today is ‘, ctime(time())\n”);
                }
        }
}
```

This is of course a rudimentary example that only runs a predefined script, rather than allowing access to any Python object. In the final product, the Python class would need to be able to see and load any requested Python object, and there would need to be a generic PythonObject class to wrap any given object. The code to invoke this Python script from Pie would be:

```
import DatePrinter from Python;
main() {
        var dp = DatePrinter();         // Create a DatePrinter
        dp.print();                     // Run its Python script
}
```

8.Project Management

Pie’s IDE extensions provide optional support for project managers and general users. These include:

- Requiring a minimum level of test coverage
- Requiring adherence to a paradigm: procedural, object oriented, or functional
- Requiring a memory manager

9.Citations

[1] International Organization for Standardization. (2011). *ISO/IEC 14882:201*1 [Online]. Available: https://www.iso.org/standard/50372.html

[2] C++ Reference. (2017). *Fundamental Types* [Online]. Available: http://en.cppreference.com/w/cpp/language/types

[3] Penn State. (2017). *Fortran 90 Pointers* [Online]. Available: http://www.personal.psu.edu/jhm/f90/lectures/42.html

[4] C++ Reference. (2017). *Dynamic Memory Managem*ent [Online]. Available: http://en.cppreference.com/w/cpp/memory

[5] C++ Reference. (2017). *Exceptions* [Online]. Available: http://en.cppreference.com/w/cpp/language/exceptions

[6] Mozilla Developer Network. (2017). *Array* [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

[7] Mozilla Developer Network. (2017). *Map* [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

[8] Python Software Foundation. (2017). *Data Structures* [Online]. Available: https://docs.python.org/2/tutorial/datastructures.html

[9] Ian Foster. (2017). *Designing and Building Parallel Programs* – Data Distribution [Online]. Available: http://www.mcs.anl.gov/~itf/dbpp/text/node85.html

[10] IBM Knowledge Center. (2017). *DO CONCURRENT Construct (Fortran 2008)* [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSAT4T_15.1.5/com.ibm.xlf1515.lelinux.doc/language_ref/do_concurrent_construct.html

[11] Cython. (2017). *Using C++ in Cython* [Online]. Available: http://cython.readthedocs.io/en/latest/src/userguide/wrapping_CPlusPlus.html

[12] Python Software Foundation. (2017). *Embedding Python in Another Application* [Online]. Available: https://docs.python.org/2/extending/embedding.html

[13] Code Project. (2015). *Calling Java from C++ with JNI* [Online]. Available: https://www.codeproject.com/Articles/993067/Calling-Java-from-Cplusplus-with-JNI

[14] Mono Project. (2017). *Embedding Mono* [Online]. Available: http://www.monoproject.com/docs/advanced/embedding/