Tangent Language Specification 0.1

# Contents

[Contents 2](#_Toc323886686)

[Introduction 6](#_Toc323886687)

[Language Overview 7](#_Toc323886688)

[Getting Started 7](#_Toc323886689)

[Types 7](#_Toc323886690)

[Nominative Typing vs Structural Typing 7](#_Toc323886691)

[Predefined Types 9](#_Toc323886692)

[Phrases 9](#_Toc323886693)

[Symbol Types 10](#_Toc323886694)

[Phrase Parameters 10](#_Toc323886695)

[this Parameter 11](#_Toc323886696)

[Type Operations 13](#_Toc323886697)

[Kinds 13](#_Toc323886698)

[Type Variables vs Constant Types 14](#_Toc323886699)

[Type Operators 14](#_Toc323886700)

[Built-in statements 15](#_Toc323886701)

[try/catch/finally 15](#_Toc323886702)

[return/yield 16](#_Toc323886703)

[repeat 16](#_Toc323886704)

[throw exception 16](#_Toc323886705)

[Built in Expressions 16](#_Toc323886706)

[Equality/Assignment/Initialization 17](#_Toc323886707)

[Block 17](#_Toc323886708)

[Anonymous Methods 17](#_Toc323886709)

[Method Dispatch 17](#_Toc323886710)

[Dynamic Dispatch 18](#_Toc323886711)

[Phrase Disambiguation 19](#_Toc323886712)

[Classes/Namespaces 22](#_Toc323886713)

[Inheritance 24](#_Toc323886714)

[Enumerations 25](#_Toc323886715)

[Where did X go? 26](#_Toc323886716)

[Arrays 26](#_Toc323886717)

[Labels/Goto/break/continue 27](#_Toc323886718)

[if/foreach/while 27](#_Toc323886719)

[using/lock 27](#_Toc323886720)

[Constants 27](#_Toc323886721)

[Properties 27](#_Toc323886722)

[Events 28](#_Toc323886723)

[Operators 28](#_Toc323886724)

[Indexers 28](#_Toc323886725)

[Constructors 29](#_Toc323886726)

[Value Types 29](#_Toc323886727)

[Interfaces 29](#_Toc323886728)

[Basic Concepts 29](#_Toc323886729)

[Programs 29](#_Toc323886730)

[Scopes 30](#_Toc323886731)

[Name Hiding 31](#_Toc323886732)

[Automatic Memory Management 31](#_Toc323886733)

[Lexical Analysis 31](#_Toc323886734)

[Input 31](#_Toc323886735)

[Whitespace 31](#_Toc323886736)

[Comments 32](#_Toc323886737)

[Tokens 32](#_Toc323886738)

[Literals 32](#_Toc323886739)

[Built-In Punctuation 34](#_Toc323886740)

[Symbols 34](#_Toc323886741)

[Identifiers 34](#_Toc323886742)

[Syntactic Analysis 34](#_Toc323886743)

[Declarations 35](#_Toc323886744)

[Field Declaration 35](#_Toc323886745)

[Method Declaration 36](#_Toc323886746)

[Type Definition Declaration 37](#_Toc323886747)

[Expressions 38](#_Toc323886748)

[Type Analysis 39](#_Toc323886749)

[Type Variables vs. Constant Types 39](#_Toc323886750)

[Type Definition Declarations, generic types and constant-ness 40](#_Toc323886751)

[Phrase Building 40](#_Toc323886752)

[Partial Application 42](#_Toc323886753)

[Phrase Parsing 42](#_Toc323886754)

[Phrase Parsing Rules 42](#_Toc323886755)

[Predefined Types 44](#_Toc323886756)

[exists 44](#_Toc323886757)

[void 44](#_Toc323886758)

[any 44](#_Toc323886759)

[type of null 44](#_Toc323886760)

[vardecl 44](#_Toc323886761)

[vardecl T 45](#_Toc323886762)

[Sub-Typing Rules 45](#_Toc323886763)

[Type Operation Behavior 46](#_Toc323886764)

[kind of 46](#_Toc323886765)

[declared type of 46](#_Toc323886766)

[with 46](#_Toc323886767)

[intersect 47](#_Toc323886768)

[implicit 48](#_Toc323886769)

[-> 48](#_Toc323886770)

[<-> 48](#_Toc323886771)

[<- 48](#_Toc323886772)

[~> 48](#_Toc323886773)

[Class Literals 48](#_Toc323886774)

[Class Modifiers 49](#_Toc323886775)

[Private Blocks 49](#_Toc323886776)

[Expression Analysis 49](#_Toc323886777)

[Initializers 49](#_Toc323886778)

[Unspecified Initializers 50](#_Toc323886779)

[Methods 50](#_Toc323886780)

[Method Parameters 50](#_Toc323886781)

[Blocks 51](#_Toc323886782)

[Block Variable Declarations 55](#_Toc323886783)

[Anonymous Methods 55](#_Toc323886784)

[Variable Declaration Expressions 55](#_Toc323886785)

# Introduction

The Tangent programming language is intended to be a platform for testing experimental programming language features. The language is intended to be a general purpose programming language with support for both object-oriented and functional programming. Program readability is a key goal. The language is intended to be a very high level language with flexible syntax that allows programmers to create domain specific languages or dialects.

The initial implementation assumes that Tangent will make use of Microsoft’s Common Language Infrastructure (CLI) for library support and execution environment. Other language implementations need not make those assumptions, provided they support an alternate method of providing equivalent features. The CLI or alternate implementation of features will be referred to as the base framework in this document. This document assumes the base framework is CLI, and that common types like bool, int, double, and string are provided.

In general, this document will use the notational conventions as used by the fourth edition of the C# Language Specification. One exception is a slight difference in grammar notation. In this document a superscript asterisk shall be used to mean 'zero or more' and a superscript plus sign shall mean 'one or more'.

Any reference to C# in the document refers to the language as of the fourth edition of its specification.

# Language Overview

This overview will describe the core features of Tangent from the perspective of someone familiar with C#. Following documentation will describe the rules that define the language, while this overview focuses on clearly defining concepts.

## Getting Started

“hello, world” in Tangent:

using .Net.System;

entrypoint => void {  
 Console.WriteLine("hello, world");  
};

Tangent source code is typically stored in one or more text files with a file extension of .tan.

Tangent has a few changes compared to the equivalent C# program:

* CLI namespaces are accessed via the .Net namespace prefix.
* The default entry point method for Tangent programs is simply entrypoint in the global namespace.
* Tangent methods may exist outside of classes. Such methods are static.
* The syntax for method definition looks more like C# lambda syntax. The name and parameters of the method exist before the => and the return type is specified after.
* Method definitions are followed by semi-colons.

## Types

Under the covers, Tangent uses CLI types to represent and store all data. Variables that are .NET value types remain value types. Variables that are .NET reference types remain reference types. All user defined types within Tangent are reference types.

### Nominative Typing vs Structural Typing

For a type in C# to be considered a subtype of another, it must explicitly inherit from that other type (or implicitly inherit from object). This called nominative typing.

User defined types in Tangent default to  *not* using nominative typing. Instead, a type in Tangent is considered to be a subtype of another if it contains all of the other's members. This is called structural typing.

using .Net.System;

pirate => class {  
 name: string;  
};

hasName => class {  
 name: string;  
};

PrintName( argument: hasName ) => void {  
 Console.WriteLine( argument.name );  
};

entrypoint => void {  
 Blackbeard: pirate = default pirate;  
 Blackbeard.name = "Blackbeard";  
 PrintName(Blackbeard);  
};

In C#, there would be an error generated at the PrintName call since pirate does not inherit somethingWithName. Due to structural typing, this program will print Blackbeard to the screen.

This program also contains other differences from C#:

* Field, parameter, and local variable definitions have the variable name first, followed by a colon and then the variable type.
* User defined types are defined using syntax similar to methods. => separates the type-name and the class definition.
* User defined type definitions are followed by semi-colons.
* Instantiating a default type uses the default keyword and does not require parenthesis

#### Specifying Nominative Typing

There are a few problems with the program above. pirate is a subtype of hasName, but *the opposite is also true*. Since hasName supplies all of the members that pirate requires, it will satisfy the structural subtyping check. Types that use nominative typing in Tangent are called goose-types:

pirate => goose class {  
 name: string;  
}

By using the goose keyword (it might walk like a duck and talk like a duck, but it is not), the only things that can be considered subtypes of pirate are types that explicitly inherit from pirate. Imported .NET types are automatically goose types to enforce nominative typing in the Tangent type system.

### Predefined Types

Not *every* Tangent type uses structural or nominative typing for subtyping checks.

* void represents nothing, as it does in other languages and despite requiring no members, nothing can be a subtype of it and it only is a subtype of exists.
* exists is a catch-all type that is a super-type for all Tangent types, including symbol types, vardecl types and void.
* null has its own anonymous type that has no members, and is a sub-type of all Tangent types except void and .
* vardecl T is a type that matches only variable declarations of at least type T.

## Phrases

When specifying a name for a method, class, field, variable, enumeration, or enumeration value Tangent allows a *series* of identifiers rather than one. Collectively, the series of tokens in these definitions are called a phrase.

In the example above hasName could be replaced with has name and remain a correct Tangent program:

using .Net.System;

pirate => class {  
 name: string;  
};

has name => class {  
 name: string;  
};

PrintName( argument: has name ) => void {  
 Console.WriteLine( argument.name );  
};

entrypoint => void {  
 Blackbeard: pirate = default pirate;  
 Blackbeard.name = "Blackbeard";  
 PrintName(Blackbeard);  
};

### Symbol Types

When the programmer specifies a phrase, Tangent breaks down each part of the phrase into an anonymous sub-method. When an identifier or symbol is encountered, a symbol type is used to represent that. In the example:

has name => class {  
 name: string;  
};

Tangent creates a method named has that takes a single parameter with the symbol type for the identifier name and returns the proper class definition. Single quotes (') are used to define these types:

has (symbol: 'name') => class {  
 name: string;  
};

Symbol types may only be used as types for parameters. (TODO: true?)

### Phrase Parameters

When declaring methods and classes, Tangent allows phrases to include parameters as long as there is at least one identifier or symbol in the phrase.

using .Net.System;

add (a: int) and (b: int) => int {  
 return a + b;  
};

entrypoint => void {  
 Console.WriteLine( add 2 and 2 );  
};

The identifier or symbol does not need to be at the start of the phrase, as long as one exists:

using .Net.System;

(a: int) plus (b: int) => int {  
 return a + b;  
};

entrypoint => void {  
 Console.WriteLine( 2 plus 2 );  
};

And symbols can be used within phrases to allow operator overloading:

using .Net.System;

(a: int) | (b: int) => string {  
 return a.ToString() + b.ToString();  
};

entrypoint => void {  
 Console.WriteLine( 2 | 2 ); // 22  
};

Tangent does not require parenthesis around parameters of a method call. Having the parameter in the correct position relative to the method is sufficient.

### this Parameter

In addition to normal parameters, Tangent uses this to be used in a parameter context to specify the instance a phrase operates on:

foo => class {  
 (this).ToString() => string {  
 return "foo";  
 };  
  
 (this) to string => string {  
 return this.ToString();  
 };  
};

Because the location of the instance parameter cannot simply be inferred in a phrase, Tangent requires its specification.

#### Conversion Methods

It is possible to define a method for a class that takes no parameters, but returns a different type. Methods of this sort are called conversion methods and have (this) as the only phrase element:

error states => enum {  
 values {  
 success,  
 read error,  
 write error  
 }  
   
 (this) => bool {  
 return this == success;  
 };  
};

Trying to declare a conversion method to the same type, or any super type of the declaring class results in an error.

#### Assignment Methods

It is possible to define a method for a class that imitates assignment to a variable of that class but from a different type. Methods of this sort are called assignment methods and look like (this) = (*paramname*: *T*) => void.

// <include some library that defines conditionals>  
error states => enum {  
 values {  
 success,  
 unknown error,  
 read error,  
 write error  
 }

(this) = (is errored: bool) => {  
 if ( is errored ) {  
 this = unknown error;  
 } else {  
 this = success;  
 };  
 };  
};

Trying to declare an assignment method where the parameter type is the same as the declaring type results in an error.

#### Property Types

It is possible to define a type that has a conversion to another type and an assignment from that same type. These types are called property types:

// <include some library that defines conditionals>  
number between (min: int) and (max: int) => class {  
 private {  
 value: int = min;  
 }

(this) => int { return value; };

(this) = (new value: int) => void {  
  
 if ( new value < min ) {   
 value = min;   
 } else if ( new value > max ) {  
 value = max;  
 } else {  
 value = new value;  
 };  
 };  
};

It is important to note that fields in Tangent are considered property types. They simply pass through to and from a private storage area.

## Type Operations

User defined type declarations in Tangent need not declare a new class. Aliasing a type to a new name can be done by changing what is on the right side of the arrow:

using .Net.System;

Integer => int;

(a: Integer) plus (b: Integer) => Integer {  
 return a + b;  
};

entrypoint => void {  
 Console.WriteLine( 2 plus 2 );  
};

### Kinds

Tangent also allows types to be used as parameters or stored as variables. The type of a stored type is called a kind. Kinds satisfy C#'s use of generic constraints. In Tangent, every type variable has a kind that restricts what sort of types may be stored in that variable:

using .Net.System;

has name => class {  
 name: string;  
};

pirate => class {  
 name: string = "Blackbeard";  
};

dog => class {  
 name: string = "Fido";  
 (this).speak => void {  
 Console.WriteLine( "woof." );  
 };  
};

entrypoint => void {  
 speaker type: kind of has name = pirate;  
 Console.WriteLine( default speaker type.name ); // BlackBeard  
 speaker type = dog;  
 Console.WriteLine( default speaker type.name ); // Fido  
};

Here if the program tries to call speak on a default speaker type there will be a compile time error since the kind does not guarantee that speak will exist.

### Type Variables vs Constant Types

Type variables are only allowed to be used in runtime code. They cannot be used as part of user defined types. Constant types on the other hand can be used in both runtime and other user defined types.

### Type Operators

In addition to existing types, Tangent provides built-in operators that work on types. If all inputs to the operators are constant types, the result will be a constant type. TODO: precedence/associativity.

#### kind of T

The built-in kind of unary operator takes a type and returns the kind for that type.

#### Mixing operators

The mixing operators create new types based on type input:

* A with B – Returns an anonymous type which inherits from A and then from B (preferring B's members that collide with A; see [TODO]).
* A intersect B – Returns an anonymous type with all of the shared members between A and B.

#### implicit T

implicit is a modifier used for specifying an implied scope to callers of a method (see [TODO]).

#### Arrow operators

The arrow operators define method signatures based on input type. They are <-,<->,->, and ~>

int -> int defines a method that takes a number and returns a new number.  
int <-> int defines a method that takes a number from the left of the method and returns a new number. Only one <-> operator may be used within each signature since it effectively marks the anchor of the phrase. For more about anchors, see .  
The <- arrow is used to specify additional parameters to the left of the method. The left arrows are placed to the left of the <-> arrow, so that the expression

A <- B <-> C

Means the signature takes type B from the left, then type A from the left, and returns C.

Combined, int <- string <-> int -> bool defines a method that takes a string from the left, returns a method that takes a number from the left, returns a method that takes a number from the right and returns true or false. The signature of a method defined like:

(x: int) (label: string) example (y: int) => bool {   
 // do stuff  
};

~> int defines a lazy reference to a number. This type does not provide a number until it is needed, but instead stores how to get that number; either from a field or from a method with all of its parameters supplied. (see [TODO])

## Built-in statements

The Tangent language itself defines far fewer built-in statements compared to other languages. The normal flow control statements exist, but are defined in terms of the language rather than as built-ins.

### try/catch/finally

Except for the new variable declaration syntax, and requiring a semi-colon after the statement, Tangent supports the same try/catch/finally syntax as C#.

using .Net.System;

entypoint => void {  
 try {  
 x: int = 42/0;  
 } catch ( ex: DivideByZeroException ) {  
 // exception handling  
 } finally {  
 // finally behavior  
 };  
};

### return/yield

Tangent uses return similarly to other languages in order to exit a method and return the specified value. Tangent also uses return in IEnumerable methods:

using .Net.System;

Fibonacci until (limit: int) => IEnumerable<int> {  
 if( limit < 0 ) {  
 return;  
 };  
  
 a: int = 0;  
 b: int = 1;  
 yield a;  
 while( b <= limit ) {  
 yield b;  
 a = b;  
 b = a + b;  
 };  
};

Instead of the C# yield return T and yield break; Tangent uses the syntax yield T and an empty return respectively.

### repeat

The repeat statement returns execution to the beginning of the current function. This is how Tangent defines many of the flow control statements without having them built in:

while (condition: ~> bool) (body: ~> void) => void {  
 if( !condition ) {  
 return;  
 }  
  
 body;  
 repeat;  
};

### throw exception

The throw exception statement raises the specified exception, just as throw in C#.

## Built in Expressions

Tangent provides a few expression elements beyond built-in statements and type operators.

### Equality/Assignment/Initialization

The equal sign (=) is used both for assignment and equality depending on context (see ). It is also used for initialization of local variables.

### Block

A block is a series of statements within curly brackets (the {} symbols). Tangent implicitly converts the block into an anonymous method of type ~> void.

### Anonymous Methods

Anonymous methods may be declared in Tangent using the => operator:

using .Net.System;  
using .Net.System.Collections.Generic;

entrypoint => void {  
 numbers: List<int> = default List<int>;  
 numbers.Add(1);  
 numbers.Add(4);  
 numbers.Add(11);  
  
 odd numbers: IEnumerable<int> = numbers.Where(   
 (i: int) => bool { return i % 2 = 1; }  
 );  
};

## Method Dispatch

Tangent allows overloading methods based on parameter count and type, similarly to C#:

using .Net.System;

example (x: int) => void {  
 Console.WriteLine("number: {0}", x);  
};

example (x: string) => void {  
 Console.WriteLine("text: '{0}'", x);  
};

entrypoint => void {  
 example 42;  
 example "hello";  
};

Here, the program would print:

number: 42  
text: 'hello'

### Dynamic Dispatch

Tangent will also call the 'most specific' method based on runtime information (see [TODO]):

using .Net.System;

A => goose class {};  
B => A with goose class {};

example (arg: A) => void {  
 Console.WriteLine("example A");  
};

example (arg: B) => void {  
 Console.WriteLine("example B");  
};

entrypoint => void {  
 instance: A = default B;  
 example instance;  
};

Here instance is a variable of type A but contains an object of type B. When example is invoked, it sees that the object is of type B so will invoke the second method. If the most specific method cannot be determined at runtime, a AmbiguousInvocationException is raised:

using .Net.System;

example (arg: exists) => void {  
 Console.WriteLine("unknown object: {0}", arg);  
};

A => goose class {};  
B => goose class {};

example (arg: A) => void {  
 Console.WriteLine("example A");  
};

example (arg: B) => void {  
 Console.WriteLine("example B");  
};

entrypoint => void {  
 variable: exists = null;  
 example variable; // Exception thrown!  
};

Here example has three overloads:

exists -> void  
A -> void  
B -> void

When example is called with a variable of type exists but a null object (or any other object that is a subtype of both A and B), an exception is thrown. Here both A -> void and B -> void are better than exists -> void but not better than each other.

If the type of variable is changed:

using .Net.System;

example (arg: exists) => void {  
 Console.WriteLine("unknown object: {0}", arg);  
};

A => goose class {};  
B => goose class {};

example (arg: A) => void {  
 Console.WriteLine("example A");  
};

example (arg: B) => void {  
 Console.WriteLine("example B");  
};

entrypoint => void {  
 variable: A = null;  
 example variable;   
};

Then an exception is *not* thrown. Since B -> void is not more specific than the compile-time signature of the method (A -> void), there is no ambiguity.

### Phrase Disambiguation

The exception is only thrown in the case of runtime ambiguity. Tangent provides other mechanisms to combat compile time ambiguity (see [TODO]). If the ambiguity is apparent at compile time, an error will be reported during compilation:

using .Net.System;

A => goose class {};  
B => A with goose class {};

example (arg: A) => void {  
 Console.WriteLine("example A");  
};

example (arg: B) => void {  
 Console.WriteLine("example B");  
};

entrypoint => void {  
 example null; // error!  
};

#### Variable vs Symbol

Tangent programs can be constructed in such a way that different method overloads, or different order of operations are valid for a given statement:

using .Net.System;

foo (x: int) => void {  
 Console.WriteLine("number: {0}", x);  
};

foo bar => void {  
 Console.WriteLine("foobar.");  
};

entrypoint => void {  
 bar: int = 42;  
 foo bar;  
};

In this program, either overload of foo could be valid. Tangent always prefers the overload that takes a symbol type. This program will then output:

foobar.

#### Prefix vs Postfix

Tangent parses expressions from left to right:

using .Net.System;

foo (x: int) => void {  
 Console.WriteLine("foo: {0}", x);  
};

(x: int) bar => void {  
 Console.WriteLine("bar: {0}", x);  
};

entrypoint => void {  
 foo: int = 42;  
 bar: int = 6;  
  
 foo bar;  
};

Since Tangent reads the expression from left to right, the first identifier matches foo: int -> void so the output of this program is:

foo: 6

Even though Tangent reads from left to right, the language prefers methods that consume the token on the left (postfix and infix) to ones that consume the token on the right (prefix):

using .Net.System;

foo (x: string) => int {  
 return x.Length;  
};

(a: string) foo (b: string) => void {  
 return a + " foo " + b;  
};

bar (x: int) => string {  
 return "bar" + x.ToString();  
};

entrypoint => void {  
 bar: string = "bar";  
 bar foo "baz";  
};

As Tangent reads the last statement, it looks at bar but there is no appropriate parameter for the method, so it continues to foo. Both of the overloads for foo have the proper parameters. Because of the preference for left-consuming methods, the foo: string <-> string -> void overload is the one chosen and the program outputs:

bar foo baz

#### Type matching

In addition to these preference checks, Tangent adds a constraint to further remove possible ambiguities:

**Statements must be of type void**.

In addition to reducing possible ambiguities, this condition allows Tangent to overload methods on *return type* as well as parameters:

using .Net.System;

pirate => class {   
 name: string;  
};

(subject: pirate) is (name: string) => bool {  
 return subject.name = name;  
};

(subject: pirate) is (name: string) => void {  
 subject.name = name;  
};

entrypoint => void {  
 Blackbeard: pirate = default pirate;  
 Blackbeard is "Blackbeard";  
 if Blackbeard is "Blackbeard" {  
 Console.WriteLine("We found 'em!");  
 };  
};

Here Blackbeard is "Blackbeard" is used in two different contexts. Since statements must result in void the first use picks the is: pirate <-> string -> void overload. Since if consumes a bool the is: pirate <-> string -> bool overload is the only one which can satisfy the constraint.

A similar constraint exists in type expressions used in method, type, and variable declarations. Type expressions must result in kind of exists or a subtype of kind of exists.

## Classes/Namespaces

Tangent uses the class keyword to define new types. Classes can include fields and methods. The class keyword may be modified with an abstract modifier that prevents instantiation. Namespaces may include fields, methods, classes, and other namespaces. The fields and methods defined within namespaces are static.

Fields that do not specify a (this) parameter are static:

using .Net.System;

configuration => class {  
 file location: string = "example.config";  
};

Classes can declare a private block to modify fields and methods as private:

using .Net.System;  
using .Net.System.Collections.Generic;

configuration => class {  
 private {  
 contents: Dictionary<string, string>;  
 }

number of items in (target: configuration) => int {  
 return target.contents.Count;  
 };  
};

Methods may be declared abstract. When abstract, the methods must be supplied before the type may be instantiated:

using .Net.System;

coordinate => class {  
 x: double;  
 y: double;  
};

movable component => class {  
 location: coordinate;  
 velocity => coordinate;  
 process frame (time: TimeSpan) for (this) => void {  
 location.x = location.x + (velocity.x \* time.TotalSeconds);  
 location.y = location.y + (velocity.y \* time.TotalSeconds);  
 }  
};

physics component => class {  
 velocity: coordinate;  
 // allow forces, etc.  
};

game entity => class {  
 process frame (time: TimeSpan) for (this) => void; // abstract method  
};

ball => game entity with physics component with movable component;

Here movable component declares velocity abstract so that it can use the method to process movement. The class itself cannot be instantiated until it is combined with a class that provides the velocity. The game entity class cannot be instantiated until it is combined with a class that provides a process frame method. Since all of the abstract members are satisfied, ball can be instantiated.

### Inheritance

Tangent provides inheritance via the with operator. The with operator combines the two parameters into a single class and returns the result. Non-private members in the class on the right hand side of the operator (the preferred class) will override non-private members in the class on the left hand side (the base class). If a member of the preferred class is abstract, it will not override a member of the base class that is a sub type of it. Using with on a class marked as abstract and one that is not will result in a type that is not abstract.

using .Net.System;

A => class {  
 private {  
 pr: string = "private A";  
 }  
  
 pub: string = "public A";  
 (this).print A => void {  
 Console.WriteLine(pr);  
 }  
};

B => class {  
 pr: string = "private B";  
 pub: string = "public B";  
 (this).print => void {  
 Console.WriteLine(pr);  
 }  
};

AB => A with B;  
BA => B with A;

entrypoint => void {  
 ab:AB = default AB;  
 ba:BA = default BA;

Console.WriteLine(ab.pub);  
 ab.print A;  
 ab.print;  
 Console.WriteLine("---");  
 Console.WriteLine(ba.pub);  
 ba.print A;  
 ba.print;  
};

Since pr is private in A there is no unification of fields when inheritance is done. pr within the scope of A is used when methods are referring to A. This program outputs:

public B  
private A  
private B  
---  
public A  
private A  
private B

## Enumerations

Tangent supports enumerations, though they differ from C#:

using .Net.System;

pets => enum {  
 values {  
 dog,  
 cat,  
 bird  
 }  
  
 (this).number of legs => int {  
 if( this = bird ) {  
 return 2;  
 }else{  
 return 4;  
 };  
 };  
};

entrypoint => void {  
 Fido: pet = dog;  
  
 Console.WriteLine("Fido has {0} legs.", Fido.number of legs);  
};

Enumerations must contain a values block, which defines the legal values of the enumeration. Enumerations may specify methods like classes do, but not fields. Enumerations may inherit from other types, but may not be inherited from. The only instances of the enumeration type are the symbols in the values block. Values may exist in multiple enumerations. In those cases, the unqualified value (like the use of dog in the entrypoint above) will be overloaded:

using .Net.System;

A => enum {  
 values {  
 value  
 }  
};

B => enum {  
 values {  
 value  
 }  
};

test (a: A) => void {};  
test (b: B) => void {};

entrypoint => void {  
 test A.value; // okay  
 test B.value; // okay  
 test value; // AmbiguousInvocationException  
};

## Where did X go?

Some features present in C# do not exist in Tangent explicitly. This section describes how to get the same behavior with Tangent, and occasionally why the same behavior is unavailable.

### Arrays

Tangent will allow interaction with CLI (or other base framework) arrays, but does not allow declaration of arrays. User defined types or base framework collections can be used in place of arrays.

### Labels/Goto/break/continue

Tangent does not provide many of the goto style statements that modify the execution point of a method. repeat and return provide those changes to the beginning and end of the method. Flow control constructs can be used to change the execution path within a method.

### if/foreach/while

Tangent does not provide flow control mechanisms other than repeat, return, and method dispatch in the language. The common flow control statements should be provided in a library that is outside of the scope of the language. (TODO: standard library requirements)

### using/lock

using and lock statements are not included as part of the language, and should be provided in a library that is outside of the scope of the language.

### Constants

Constants in Tangent are provided as static methods:

namespace math {  
 pi => double { return 3.14159265; };  
};

### Properties

Properties in Tangent are created by using or by having two distinct methods with the same name:

using .Net.System;

pirate => class {  
 private {  
 gold pieces: int;  
 }  
  
 (this).gold => int { return gold pieces; };  
 (this).gold = (value: int) => void {  
 if( value >= 0 ) {  
 gold pieces = value;  
 };  
 };  
};

gold here has a getter and a setter. The various property variations can be achieved by implementing or not implementing the getter and setter in a private and/or static context.

### Events

Tangent allows manipulation of base framework events, but provides no syntax for adding them to user defined types. They will have to be made manually:

using .Net.System;  
using .Net.System.Collections.Generic;  
  
print with event => class {  
 private {  
 onPrint: List< string -> void > = default List< string -> void >;  
  
 (this).onPrint (argument: string) => void {  
 foreach( action: string -> void in onPrint ) {  
 action(argument);  
 };  
 };  
 }  
  
 (this).OnPrint += (action: string -> void) => void {  
 onPrint.Add(action);  
 };  
  
 (this).OnPrint -= (action: string -> void) => void {  
 onPrint.Remove(action);  
 };  
  
 (this).print (text: string) => void {  
 Console.WriteLine(text);  
 onPrint(text);  
 };  
};

### Operators

Tangent provides operator overloading via symbols in method definitions. (TODO: implicit conversions via this -> type?)

### Indexers

Tangent provides indexers by simply using the square brackets in a phrase definition:

using .Net.System;  
using .Net.System.Collections.Generic;

configuration => class {  
 private {  
 storage: Dictionary<string, string> =   
 default Dictionary<string, string>;  
 }  
  
 // …  
};  
  
(config instance: configuration) [(key: string)] => string {  
 if( config instance.storage.ContainsKey(key) ) {  
 return config instance.storage[key];  
 }  
  
 return string.Empty;  
};

### Constructors

Tangent provides constructors via phrase definition in the class declaration:

using .Net.System;

web server (port: int) => class {  
 listening port: int = port;  
 // …  
};

web server => web server 80;

### Value Types

Tangent does not allow user defined value types.

### Interfaces

Tangent does not allow user defined interfaces. Abstract classes and Tangent's inheritance model provide similar behavior.

# Basic Concepts

## Programs

A Tangent program consists of one or more source files. A source file is an ordered sequence of Unicode characters. Tangent compilation has an atypical compilation path, used to support phrase definition:

1. Lexical Analysis, which translates the stream of Unicode characters into a stream of tokens.
2. Syntactic Analysis, which verifies that the stream is well formed and identifies tokens (or token groups) that have a known role in the program.
3. Type Analysis, which determines what types the program has defined, and how they are inter-related.
4. Expression Analysis, which determines the role for tokens with unknown or ambiguous roles. This step is responsible for determining how phrases need to be executed in order to result in void.
5. Code Generation, which verifies that none of the expressions cause semantic errors and generates executable code from the defined phrases.

When a Tangent program is run, the method entrypoint in the global namespace with the signature ~> void is executed.

## Scopes

The scope of a phrase is the region of program text within with is it possible to refer to the phrase. Scopes can be nested. If a phrase is declared in an inner scope, it gains priority over the phrase in the outer scope. The phrase in the inner scope must match the signature of the phrase in the outer scope to gain priority. [TODO: allow alias/disambiguation]

* Phrases declared outside of a namespace declaration are in the global scope.
* The scope of phrases defined in a namespace declaration is within the namespace defined by the namespace's name and all namespaces declared within that namespace.
* The scope of phrases included via a using statement is the source file that the using statement exists in.
* The scope of parameter declarations are anything after the => sign in their owning declaration. For methods, this means the return type and block. For types, this means the type definition (including any anonymous class/enum declarations).
* The scope of elements declared within anonymous classes or enums is the elements between the curly brackets {} for each declaration.
* Elements to the left of the with operator are included in the scope of the elements to the right of the operator, as defined in TODO and .
* The scope of a block variable declaration is the block that it is declared in.
* The scope of a variable declaration expression is the sequence of expressions that it exists in.
* When passing a parameter to a method marked with an implicit parameter, the expression elements are within the scope defined by the implicit parameter. (See TODO)

It is possible to refer to a phrase before it is declared, as is possible in C#.

### Name Hiding

Tangent supports name hiding via private declaration. Normally, a phrase included in a class via inheritance is part of the scope of the new class. Phrases that are part of a private block declared in the base class are not visible in the inheriting class' scope.

## Automatic Memory Management

Tangent is designed to utilize automatic memory management provided by the base framework. At the very least, the base framework needs to provide:

1. Automatic memory allocation for newly created objects.
2. Automatic freeing of the allocated memory once no part of the object can be accessed by any possible continuation of execution.

# Lexical Analysis

## Input

The input production defines the lexical structure of a Tangent source file. Each Tangent source file must conform to that production.

input::  
 input-section\*

input-section::  
 whitespace  
 comment  
 token

Tangent does not distinguish between line terminators and other whitespace. Tangent does not provide any pre-processing capabilities. Whitespace and comments separate tokens. If multiple lexical grammar productions match a sequence of characters, the longest possible element is preferred.

## Whitespace

Whitespace encompasses the non-visible characters that help format the source file.

whitepace::  
 Any character that matches .NET's char.IsWhiteSpace(char)

At time of writing [this document](http://msdn.microsoft.com/en-us/library/t809ektx.aspx) provides the complete list of Unicode characters that are considered whitespace.

## Comments

Tangent supports C++ style comments.

comment::  
 single-line-comment  
 multi-line-comment

single-line-comment::  
 // Any Unicode character except a newline single-line-comment-terminator

single-line-comment-terminator::  
 newline  
 <eof>

newline::  
 Carriage return (\u000D)  
 Line feed (\u000A)  
 Next line (\u0085)  
 Line separator (\u2028)  
 Paragraph separator (\u2029)

multi-line-comment::  
 /\* Any Unicode character except \* \*/

## Tokens

Tokens represent the individual elements that comprise a functional Tangent program. They are broken into four general categories: literals, punctuation, symbols, and identifiers.

token::  
 any-literal  
 built-in-punctuation  
 identifier  
 symbol

### Literals

A literal is a source code representation of a value.

any-literal::  
 int-literal  
 real-literal  
 string-literal

#### Integer Literals

Integer literals define constant whole number values. Tangent implementations must define conversion rules for how the number is translated into the base framework. The production is simply any sequence of decimal digits:

int-literal::  
 decimal-digit+

decimal-digit:: one of  
 0 1 2 3 4 5 6 7 8 9

It is expected that implementations will provide a unary negation operator for negative values.

#### Real Literals

Real literals define constant fractional number values. Tangent implementations must define conversion rules for how the number is translated into the base framework. The production is simply any sequence of decimal digits, a decimal point, and another sequence of decimal digits:

real-literal::  
 decimal-digit+ . decimal-digit+

It is expected that implementations will provide a unary negation operator for negative values.

#### String Literals

Tangent supports two forms of string literals similar to C#: regular string literals and verbatim string literals. Regular string literals support simple escape sequences.

string-literal::  
 regular-string-literal  
 verbatim-string-literal

regular-string-literal::  
 " regular-string-literal-character\* "

regular-string-literal-character::  
 Any character except for " \ and newline  
 simple-escape-sequence

simple-escape-sequence::  
 \ simple-escape-character

simple-escape-character:: one of  
 " \ 0 a b f n r t v

verbatim-string-literal::  
 @ " "" or Any character except " "

Tangent does not provide an escape sequence for the single quote. Simple escape sequences except for \' translate identically to C#.

### Built-In Punctuation

Even with the flexibility in user defined operators, Tangent requires some reserved punctuation for doing the defining.

built-in-punctuation:: one of  
 { } ; ( ) " => ~> :

### Symbols

Symbols represent the not built-in, not alphanumeric characters that Tangent can use for declaring operators in phrases.

symbol::  
 Any Unicode symbol in categories Po, Ps, Pe, Pd, Sm, Sc, or So except for built-in-punctuation

Each Unicode character represents an individual token in the stream. Note that the multiple character punctuation (=> and ~>) are single tokens and are whitespace sensitive. = > is interpreted as two symbol tokens due to the space between the characters.

### Identifiers

Tangent has similar identifier definition as C# and other languages except that it allows any identifier character to be used anywhere in the identifier instead of just at the beginning of the identifier, and digits are not allowed:

Identifier::  
 Any number of Unicde symbols in categories Lu, Ll, Lt, Lm, Lo, Nl, Mn, Mc, or Pc

Tangent does not support C#'s @identifier style escaping for keywords since there are no explicit keywords.

# Syntactic Analysis

Each Tangent source file has a series of declarations that act as the constituent pieces of that file:

compilation-unit::  
 using-statement\* namespace-element\*

using-statement::  
 using delimited-namespace-identifier ;

namespace-element::  
 namespace-decl  
 type-definition-decl  
 concrete-method-decl  
 field-decl

namespace-decl::  
 namespace delimited-namespace-identifier { namespace-element\* }

delimited-namespace-identifier::  
 .? identifier  
 delimited-namespace-identifier . identifier

A source file is composed of any number of using statements, followed by any number of items legal in a namespace. This anonymous namespace is the global namespace. Different namespace declarations are aggregated to form a single namespace with all of the members from each of the namespace declarations. Namespaces declared within other namespaces are appended to the containing namespace with a period:

namespace Foo {  
 namespace Bar { // effectively Foo.Bar  
 }  
}

## Declarations

Outside of namespaces, Tangent provides 3 core declarations: variables, methods, and types.

### Field Declaration

Fields represent actual storage space for variables in Tangent.

field-decl::  
 vardecl initializer? ;

vardecl::  
 vardecl-element+ : vardecl-type

vardecl-element::  
 symbol  
 identifier  
 ( this )

vardecl-type::  
 type-decl except =

initializer::  
 = block ;  
 = initializer-expression ;

initializer-expresion::  
 expression-element+ that does not start with a block

The phrase here defines the field name and the type declaration defines the field type. An initializer may be specified to populate the field before first usage. The block must take no parameters and return at least the type of the field type. Omission of an initializer will cause the initializer to be the result of the default method called with the field type as its only parameter. Further, omission of an initializer changes the inheritance behavior of that field if a name collision occurs (see ). If no acceptable overload exists for a given type (as would happen if the field type is itself declared abstract) a compiler error occurs. Initialization behavior is discussed further in [TODO]

Fields automatically declare a method for access and a method for assignment. These have the signatures (field name) -> (field type) and (field name) = (field type) -> void respectively.

It is an error to declare two fields with the same name and types where (at least) one of the field types is a subtype of the other.

### Method Declaration

Methods provide the executable commands of a Tangent program.

method-decl::  
 abstract-method-decl  
 concrete-method-decl

abstract-method-decl::  
 method-phrase-element+ => type-decl ;

concrete-method-decl::  
 method-phrase-element+ => type-decl block ;

method-phrase-element::  
 ( this )  
 phrase-element

phrase-element::  
 symbol  
 identifier  
 parameter-decl

parameter-decl::  
 ( vardecl )

For methods, the list of phrase elements define the method name. The type declaration defines the method return type. Methods cannot contain only parameter declarations; doing so results in a compiler error. The use of a (this) element in a field or method declared within a namespace results in a compiler error. Not using a (this) element in a method name declared within a class/enum results in a compiler error. Using a (this) element in a parameter delcaration results in a compiler error. Variables defined within parameter declarations cannot be assigned to. A parameter of type vardecl T must be preceded and succeeded by a symbol or identifier.

### Type Definition Declaration

Types provide the structure of Tangent data, as well as providing Tangent a way to understand if things 'make sense'.

type-definition-decl::  
 phrase-element+ => type-decl ;

anonymous-class-decl::  
 class-modifier\* class { class-decl-element\* }

anonymous-enum-decl::  
 enum { enum-decl-element\* }

class-modifier::  
 goose  
 abstract

class-decl-element::  
 field-decl  
 method-decl  
 private-class-block

enum-decl-element::  
 concrete-method-decl  
 values-decl  
 private-enum-block

private-class-block::  
 private { 0 or more class-decl-element that is not private-class-block}

private-enum-block::  
 private { concrete-method-decl\* }

values-decl::  
 values { comma-delimited-value-list}

comma-delimited-value-list::  
 value-entry  
 comma-delimited-value-list , value-entry

value-entry::  
 One or more vardecl-element except for ,

An enum must have one and only one values block, and that block must contain at least one value. It is a compiler error to have duplicate value entries for the same enum declaration. Classes and enums may have multiple private blocks. Classes and enums may have multiple fields and/or methods with the same name as long as they have different types/signatures. Having multiple fields with the same type in a class declaration is a compiler error. Having multiple methods with the same signature in a class or enum declaration is a compiler error. Note that unlike C# differing return type is sufficient to avoid the compiler error. Having multiple type definition declarations with the same signature in the same namespace is a compiler error. Note that how the type is defined does not impact this signature.

## Expressions

When Tangent parses expressions, the actual interpretation of those expressions into operations does not occur via grammar. Figuring out how each element fits into an expression occurs in the expression analysis step.

expression-element::  
 identifier  
 symbol  
 any-literal  
 paren-expression  
 block  
 anonymous-method  
 anonymous-class-decl  
 anonymous-enum-decl  
 built-in-typeop

paren-expression::  
 ( expression-element\* )

block::  
 { statement\* }

anonymous-method::  
 parameter-decl+ => anonymous-method-return-type block

anonymous-method-return-type::  
 type-decl unless the expression-element is block

statement::  
 expression-element+ that is not block-variable-decl ;  
 block-variable-decl ;

block-variable-decl::  
 vardecl = initializer-expression  
 vardecl = block

type-decl::  
 expression-element+

built-in-typeop::  
 ~>  
 ->  
 <-  
 <->  
 :  
 ' symbol-type-value '

symbol-type-value::  
 identifier  
 symbol

Use of a (this) parameter in the vardecl in block-variable-decl results in an error.

#### *A note on Abstract/Concrete Method syntax ambiguity*

There is a bit of ambiguity between concrete and abstract method declarations since a block can be part of the type declaration. The language will always prefer the concrete method interpretation when ambiguity occurs.

# Type Analysis

Types are extremely important to Tangent programs. They define structures for data, but they also implicitly define the relations between those structures as well as how expressions may be built by the compiler into functional code. After parsing Tangent source code, the language determines the types declared.

## Type Variables vs. Constant Types

Tangent allows types to exist in two general forms; as a type variable and as a constant type. Constant types are the types defined by the user type definition declaration syntax:

foo => class {  
 (this).x : int = 42;  
};

foo cannot be modified, but foo can be used anywhere that requires constant types, like other type definitions, method declarations, and variable declarations. Rules for determining if a type definition declaration is constant is defined in Type Definition Declarations and constantness

Type variables are defined like other variables in Tangent:

bar : kind of any = class {   
 (this).x : int = 42;  
};

bar can be modifed, but only a type that satisfies the kind of bar may be assigned to it. All type variables have a kind. Trying to assign a type that does not satisfy that kind results in a compiler error, just like if the programmer tried to assign a number to a string variable. Type variables are much more limited in what they may do since it is unknown what type the variable might reference at any given time. [TODO: enumerate these] [TODO: for more info, see Kind rules]

### Type Definition Declarations, generic types and constant-ness

If a type definition is a constant type depends largely on how its parameters (and the parameters of its compositional types [if any]) are used. How that behaves depends on if the type parameters are generic or not. In Tangent, a parameter used in a type definition declaration is considered generic if:

* The parameter is used to define any type information for the declaration.
* The parameter is used with built-in type operators to define any type information for the declaration.
* The parameter is used as a parameter to a type definition generic parameter that defines any type information for the declaration.
* The parameter is used in a base framework type as a parameter used to construct a type.

A type is considered constant if it has no generic parameters, or if all of its generic parameters are specified with constant values.

Constant values are:

* Constant types
* any-literal
* Methods
* Anonymous methods
* The type being considered itself

## Phrase Building

All Tangent declarations allow some sort of phrase definition. This section defines how phrases are handled uniformly.

At the most elemental level, all Tangent methods are unary methods; they take only one parameter. When a phrase is declared, it is built into a series of unary methods that take a parameter and returns another unary method that takes the next element of the phrase, and so on until all parts of the phrase are dealt with.

The order that the methods are generated is:

* Start with the first symbol or identifier in the phrase. This is called the phrase's anchor. This acts as the first parameter to the first method. Tangent uses to specify the type of the parameter for both symbols and identifiers.
  + If there are no symbols or identifiers, a (this) parameter may be used as an anchor. If a phrase contains no anchor, this results in a compiler error.
* From that starting point, other symbols or identifiers to the right of that anchor are processed next.
* After the contiguous symbols and identifiers, parameters to the left of the anchor (if any) are processed in order from the anchor to the start of the phrase.
* Then the rest of the phrase is processed from left to right.
* After all of the phrase elements are processed, what the phrase refers to is used as the last return type of the last method. For methods, this means the return type declared in the method. For parameters, this means the type declared for the parameter. For type declarations, this means the kind of the declaration.

So, for example:

dot product (v1: vector) and (v2: vector) => vector

For this, dot serves as the anchor since it is the first symbol or identifier. product is adjacent to the anchor to the right so is consumed next. Next Tangent looks for parameters to the left of the anchor. Since there are none, that step is skipped. Then the last step of consuming the remaining phrase elements. The final result looks like:

A method that takes 'dot' and returns  
 A method that takes 'product' and returns  
 A method that takes a vector and returns  
 A method that takes 'and' and returns  
 A method that takes a vector and returns  
 A vector

Tangent does not require the generation of methods in this way, but requires that and [TODO: expression construction] rules are kept.

### Partial Application

Tangent phrases support specifying part of the phrase in order to get a sub-phrase. This is called partial application. For example:

Double: int -> int = 2\*;  
x: int = Double 4; // 8

Only partial application along the lines of the phrase construction is supported. Assuming only the default integer multiply method is used for the symbol \* then partial application will not apply to \* 2.

## Phrase Parsing

To take a type-decl and make sense of it, Tangent applies a set of rules to determine the order of operations. These rules are identical for type-decl, initializer-expression, statement, and paren-expression. The input to the rules is the sequence of expression elements, and a target type. The result of the rules is one of:

1. How to arrange the expression elements such that they will result in the target type.
2. An error indicating that the expression is ambiguous.
3. An error indicating that the expression cannot be formed in such a way to result in the target type.

The target type for type-decl is kind of exists. The target type for initializer-expression is the type of the declaration that is being initialized. The target type of a statement is void. The target type of paren-expression is specified by the parameter type of whatever method is trying to consume the paren-expression.

As the rules work through the expression list, expressions will be combined or translated into new entries in the list that represent a method call or other operation. In the rules defined below, this list is referred to as the node list. Each element in the list has an effective type (except for paren-expression elements), that represents the result of whatever expression or operation the element represents.

### Phrase Parsing Rules

If the node list matches one that was already tested for the expression list, return a 'no answer' error.

If the node list contains only one element, and that element's effective type is compatible with the target type, return that element as a valid answer about what order of operations will result in the target type. Otherwise, continue.

For each element in the list:  
 If the effective type of the current element is not a method, skip it and proceed to the next.  
 Test each method overload using the .   
 If no overload results in an answer, continue.  
 If one overload results in an answer, return that answer.  
 If multiple overloads result in answers, see if the eliminate possible answers. If only one answer remains, return that answer. If not, return an expression ambiguity error.

If the end of the list is reached without finding an answer, return a 'no answer' error.

#### Method Reduction Process

Take the current method and see if a parameter exists that is compatible with the method's parameter type. For postfix methods, that means the previous element in the list. For prefix methods, that means the next element in the list. For methods with symbol type parameters, that means that the element has the equivalent symbol or identifier. For methods without symbol type parameters, that means that the element's effective type is compatible with the method's parameter type. If the element has no effective type because it is a paren-expression element, follow the *Paren Expression Compatibility Rules*.

[TODO: this.arg and implicit scoping]

If a suitable parameter exists, repeat the with a list that replaces the method's element and the parameter's element with an element that represents binding that parameter to the method.

When a method has all of its parameters bound (or is a conversion method) it has the type ~> R where R is the return type of the method. Methods of this sort need no suitable parameter to test. For these, repeat the with a list that replaces the method's element with an element that represents executing the fully bound method.

#### Candidate Preference Rules

Candidate preference rules work on a set of possible methods called the candidate pool. The rules eliminate less-preferred candidates. If at any point 1 or 0 candidates remain in the pool, the result is returned. The rules are specified in order of importance. If two rules could apply, the first rule is applied.

* If a candidate has a symbol type as a parameter, the other candidates are eliminated.
* If a candidate is a binding operation, executing operations are eliminated.
* If a candidate is a postfix method, prefix methods are eliminated.
* If a candidate's parameter type (if any) is a subtype of (but not equivalent to) another candidate's parameter type, the other candidate is eliminated.
* If two candidates' parameter types are equivalent, but a candidate's return type is a subtype of (but not equivalent to) another candidate's return type, the other candidate is eliminated.

#### Paren Expression Compatibility Rules

If the method's parameter type is the appropriate symbol type ('(' for prefix methods, ')' for postfix methods) repeat the with the paren expression element replaced with a list of elements representing the expressions within the parenthesis and literal ( and ) at the start and end of the new list of elements respectively.

Otherwise, repeat the with the node list being the list of elements representing the expressions within the parenthesis and the target type being the parameter type of the method that is checking for compatibility.

## Predefined Types

Tangent has a set of types that are required by implementations to provide. They are treated specially in the sub-typing rules.

### exists

exists is a placeholder type that super-types all other types in the system. It is abstract. It contains no members, and is a normal type.

### void

void is a placeholder type that represents nothing. It is abstract. It contains no members and is a normal type.

### any

any is a type that super-types all normal objects. It is abstract. It contains no members, and is a normal type.

### type of null

The phrase type of null provides the type of the null literal. It is abstract. It contains no members, but type-checks as though it does. For more information about null's behavior, see [TODO]

### vardecl

vardecl is a simple alias for vardecl exists.

### vardecl T

vardecl T is a type that represents a variable declaration (of at least type T), which allows access to a shared variable by a different name.

## Sub-Typing Rules

Throughout the document, there is the concept of sub-typing or a type being compatible with another. This section defines how that relation is determined. To determine if a type *B* is a subtype of type *A*, these checks are performed in order (or out of order while maintaining their ordered behavior):

* If *B* is the same type as *B* then *B* is a subtype of *A*.
* Check if *A* is exists. If it is, then *B* is a subtype of *A*.
* Check if *A* is void. If it is, then *B* is **not** a subtype of *A*.
* Check if *A* or *B* are symbol types. If they are, then *B* is **not** a subtype of *A*.
* [TODO: vardecl]
* Check if *B* is type of null. If it is, then *B* is a subtype of *A*.
* Check if *A* is any and *B* is not a method (unless *B* has a method anchored via (this)). If it is, then *B* is a subtype of *A*.
* If *A* is a kind:
  + If *B*is a kind, *B*is a subtype of *A* if and only if the type *B* is a kind of is a subtype of the type *A* is a kind of.
  + If *B* is a constant type, *B* is a subtype of *A* if and only if *B* is a subtype of the type *A* is a kind of.
  + Otherwise, *B* is **not** a subtype of *A*.
* Otherwise (*A* is a common type):
  + If *B* is a kind or a constant type, *B* is **not** a subtype of *A*.
  + If *B* is a generic type, *B* is a subtype of *A* if and only if the type constraint of *B* is a subtype of *A*.
  + If *B* is abstract and *A* is not abstract, *B* is **not** a subtype of *A*.
  + If *B* does not contain all of the goose type references that *A* contains, *B* is **not** a subtype of *A*.
  + For each non-private, non-static method defined in *A* of type *Ap -> Ar* (also considering the methods (this).*fieldname* => *fieldtype* and (this).*fieldname* = (value:*fieldtype*) => void for each non-private, non-static field):
    - If there does not exist a method *Mp -> Mr* in the scope where *B* is defined with the same anchor where A*p* is a subtype of M*p* and *Mr* is a subtype of *Ar* then *B* is **not** a subtype of *A*. [TODO: review once module behavior is done]

## Type Operation Behavior

Tangent provides a number of built in operators that work with types. They are defined as part of the language so that other operations can be defined in terms of them. Unless otherwise specified, all type operators will return a constant type if their parameters are constant types.

*Note:* Types imported from the base framework might have limitations that prevent them from working as specified when working with those types. Tangent implementations must specify how imported types deviate from the specified behavior.

If the operator defines an error state, that error can occur different depending on the context that it occurs. If the error state occurs at compile time, a compiler error is generated. If the error state occurs at runtime due to a type variable assignment, an implementation defined exception is thrown.

### kind of

The kind of operator is a prefix operator that takes one type as its parameter. It returns a type that defines a variable or parameter that accepts any other type that is a subtype of the specified parameter. Calling kind of on a kind results in an error.

### declared type of

The declared type of operator is a prefix operator that takes a vardecl T as a parameter. It returns the value (of type kind of T) that is the declared type of the variable declaration.

### with

The with operator is an infix operator that takes two parameters with types kind of exists and returns a type that is a subtype of both specified types (if no exceptions occur). For an expression *A* with *B*:

* If *B* is a subtype of *A*, then *B* is returned.
* Otherwise, if *A* is a subtype of *B*, then *A* is returned.
* Otherwise if either parameter is void, or a symbol type an error occurs.
* Otherwise, a type is returned that follows these rules:
  + The parameters are considered to be inherited by the result type, so that if either parameter (or the types *they* inherit) are goose types, the result type can be considered subtype of the goose types.
  + If both parameters are declared abstract, the result type will be declared abstract. Otherwise it will not.
  + The result type contains storage for each private field.
  + No private fields or methods are visible for the result type.
  + An error occurs if two fields exist in either parameter with the same phrase, but different non-equivalent types where one of the types is a subtype of the other.
  + If two fields exist in either parameter with the same phrase with equivalent types, a single field is included in the result type. Its field type is equivalent to the types of the parameters' fields. The initializer used for the field is picked using these rules:
    - If the initializer for the field declared in *B* is omitted, an initializer equivalent to the initializer for the field declared in *A* is used (even if that is also omitted).
    - Otherwise, an initializer equivalent to the initializer for the field declared in *B* is used.
  + Any other fields are included in the result type with the initializers they are declared with.
  + All methods from both parameters are included in the result type except where:
    - The method is abstract and there exists another method that is a subtype of the abstract method.
    - The method is equivalent to an auto-generated method for a field.
    - The method is declared in *A* and a method exists in *B* with the same signature.

### intersect

The intersect operator is an infix operator that takes two parameters with types kind of exists and returns a type that is a supertype of both parameters. For an expression *A* intersect *B*:

* If *A* is a subtype of *B* then *B* is returned.
* If *B* is a subtype of *A* then *A*is returned.
* If *A* or *B* is a symbol type, then exists is returned.
* If *A* or *B* is void then void is returned.
* Otherwise a type is returned that follows these rules:
  + The type is abstract.
  + The type is considered to inherit from all goose types that both *A* and *B* inherit from.
  + The type includes abstract methods with the same signatures as all non-private, non-static methods (including auto-generated non-private field methods) in *A* if and only a method with the same signature exists in *B*. [TODO subtype sigs?]

### implicit

The implicit operator is a prefix operator that modifies a parameter type, if that parameter type is a single method type. It provides an implicit scope for the parameter of the method type at call-sites when defining the parameter where the implicit operator exists. [TODO: make this coherent and better defined]

### ->

The -> operator is an infix operator that takes two parameters of type kind of exists and returns a prefix method signature where the left hand side of the operator is the parameter type and the right hand side of the operator is the return type.

### <->

The <-> operator is an infix operator that takes two parameters of the type kind of exists and returns a postfix method signature where the left hand side of the operator is the parameter type and the right hand side of the operator is the return type.

### <-

The <- operator is an infix operator that takes a parameter of the type kind of exists from the left hand side and a postfix method signature from the right hand side. The result of the operation is a postfix method signature where the left hand side of the operator is the last parameter in the series of postfix method signatures.

For example:

A <- B <-> C is interpreted as a postfix method that takes type B as a parameter, and returns a method that takes type A as a parameter which returns type C.

### ~>

The ~> operator is a prefix operator that takes one parameter of type kind of exists and returns a type signature that represents a fully bound method that when executed results in the type specified by the parameter of ~>.

## Class Literals

A class literal is formed using the class keyword.

anonymous-class-decl::  
 class-modifier\* class { class-decl-element\* }

The class declaration has access to variables within the scope of its declaration. Use of variables that exist outside of the declaration in the type declaration of its members can cause the class literal to be generic. Referencing variables outside of the declaration can cause them to be captured, similarly to anonymous methods as described in (TODO).

Non-generic classes that do not capture variables must share instances. (TODO: example for and against)

For the behavior of field and method declarations, see (TODO) and (TODO).

### Class Modifiers

Classes may optionally include modifiers goose and/or abstract. A class may have both modifiers; in this case, both are applied. A class may have a modifier specified more than once; in this case the modifier is simply applied as if it were only specified once.

#### abstract

The abstract modifier prevents instantiation of the specified class, and modifies what classes it subtypes and is subtyped by.

#### goose

The goose modifier requires that classes explicitly inherit the modified class in order to be considered a subtype of it.

### Private Blocks

In class literals, fields and methods behave as they normally do in a non-class context except when they exist within private blocks. A private block limits the visibility of the fields and methods declared within the block to the class literal where they are defined.

# Expression Analysis

Expression Analysis represents a step in compilation that Tangent includes unlike many other languages. Within this step, the actual instructions of the program (method blocks and initializers) are transformed from a flat sequence of expression elements into the more common syntax tree that includes what operations to perform and in what order.

## Initializers

Fields and variables have initializers to provide a default value. Initializers come in two forms: Initializer Expressions and Initializer Blocks. An initializer expression is a simple expression that represents a value for the field or variable. An initializer block is a block that must return a value for the field or variable. An expression that cannot be formed into a value of the correct type, or a block that does not include a return statement of the correct type is an error.

Instance fields must be initialized on instantiation of the type. The order is implementation dependent. Static fields must be initialized before first access at an implementation dependent time before first access. Variables in a block are instantiated in textual order at an implementation dependent time so long that rules about operation reordering (TODO) are met.

Field initializers may not get or set the value of another field within the same declaration space as the field the initializer is declared for.

Initializer expressions are transformed into a syntax tree using the standard where the specified target type is the type of the field or variable being initialized. Initializer blocks are transformed into a syntax tree by executing an anonymous method of the type ~> T where T is the type of the field or variable being initialized and the body of the anonymous method is defined by the initializer block.

### Unspecified Initializers

Initializers are optional. If unspecified, the initializer will be the result of the expression default *T* where *T* is the phrase specified for the field or variable type that the initializer is declared for.

## Methods

Methods represent a series of operations, usually resulting in a value of a particular return type. Methods usually will have one or more method parameters that allow values to act as input to the operations.

concrete-method-decl::  
 method-phrase-element+ => type-decl block ;

method-phrase-element::  
 ( this )  
 phrase-element

phrase-element::  
 symbol  
 identifier  
 parameter-decl

parameter-decl::  
 ( vardecl )

### Method Parameters

As the method is supplied parameters, each (not-vardecl) parameter is copied and assigned to storage represented by the parameter phrase. Parameters of type vardecl T instead use the storage provided by the referred to variable declaration. Method parameters can be referred to via the parameter phrase, and upon execution of the method, the parameter value will be supplied. The storage represented by the parameter phrase may not be written to after the initial population.

The special this parameter exists for instance methods. Referring to it will get the current instance of the class that was specified when invoking the method. this in a static method uses the normal name resolution rules.

### Blocks

A block is a series of statements between curly brackets {}. Blocks are used to define the operations that are executed when a method is invoked, the operations that are executed when an anonymous method is invoked, the operations that are executed when a field with an initializer block is initialized and as sequence of operations passed into another method. Each block has a return context. The return context defines a type that must be supplied by a return statement, as well as what point of the program to jump to when the return statement is encountered.

The return context of a method block or anonymous method block is the return type of the method it is declared for, and will exit method invocation to the point at which the method was invoked (except in the ). The return context of an initializer block is the type of the field that the initializer is declared for, and will exit the initializer to continue other execution. The return context of block expressions are described in the sub-section .

Each component of a block is transformed into a syntax tree using the standard where the target type for each is void and taking the Built In Methods into account.

#### Enumerable Return Context

Tangent requires the base framework to provide one or more Sequence Types. In .NET, the sequence type is IEnumerable. The sequence type represents an ordered sequence of items. The type must have the ability to get the next item in the sequence (if any). If the return type of a method or anonymous method is a sequence type, an enumerable return context is used.

In an enumerable return context, three different return methods are available:

1. return; - Calling return with no parameters will exit the method, and not supply any value.
2. return *T*; - Calling return *T* requires that T be a subtype of the sequence type specified as the return type of the method. All items in the value specified will be returned in sequence, and then the method will be exited.
3. yield *T*'; - Calling yield T' requires that T' be a subtype of the items that can be supplied by the sequence type specified as the return type of the method. The specified value will be returned, and method execution will 'pause' until the next item in the sequence is requested similarly to C#'s yield return.

When a block (or any block expressions it contains) contains either return; or yield T;, the method will behave differently upon execution. Upon execution, an object of the specified sequence type will be returned immediately. As the method to get the next item is invoked, the block will be executed until one of the three return methods are encountered and the next item(s) will be supplied as described above. Further invocations of the get next item method will provide no item if a return statement was called. Otherwise, the block will continue execution from the next statement after the yield statement it stopped at.

#### Block Expressions

Blocks may be used as an expression element within another block. When this occurs, the block expression has the same return context as the block that it is declared within. If the block expression is not declared within a block (in a return type expression for example), then it has no return context.

Block expressions have the type ~> void and behave as anonymous methods of the same type would behave with regards to variable capture. They can be assigned to variables, and passed as arguments to methods. When the ~> void variable is executed the statements in the supplied block are executed.

If the block expression encounters a return statement, the value will be set for the return context for where the block expression was created (if it still exists), and the execution will exit (or pause for a yield) for execution of the block. If the path of execution returns to the method whose return context has been set, it will then behave as though the return statement had been called after the statement that was last executed.

For example:

using .Net.System;

execute and log (operation: ~> void) => void {  
 Console.WriteLine("Before operation.");  
 operation;

// printed.   
 // The return stops execution of the block and the method it is  
 // declared it if it is executed within that context, but   
 // \*not\* any intermediary methods.  
 Console.WriteLine("After operation.");   
}

entrypoint => void {  
 execute and log {   
 Console.WriteLine("Hello World.");   
 return;

// not printed, return stops execution of the block.  
 Console.WriteLine("Hello?");   
};

// not printed, return in the block stops execution of the   
 // method it is declared in when execution returns here.  
 Console.WriteLine("We're done!");   
}

But:

using .Net.System;

operation: ~> void;

operation prints (i: int) => void {  
 operation = { Console.WriteLine(i); return; };  
 Console.WriteLine("Operation set.");  
}

entrypoint => void {  
 operation prints 42;  
 operation;  
}

In this example, the return statement in the block is redundant. The operation prints method has already exited before the return statement is called.

The repeat statement also effects the method it is declared in rather than the block it is executed in. If execution returns to the method that the block expression was created in, then it will proceed from the beginning of the method rather than after the statement that led to the block expression's execution. The repeat statement will also exit the block expression execution immediately.

#### Built In Methods

In addition to the usual scoping rules, there are a number of phrases that are always available and always take priority. They are not fixed keywords, because the various identifiers used for the built in methods may be used by the user in phrases unrelated to the built in methods.

##### *return*

Methods that do not use an and do not return void will have a built in method with a signature of return (value: T) => void where T is the return type of the method. If the method returns void then a built in method with a signature of 'return' -> void. When called, this method exits the given method (see for behavior when return is called in a block expression) returning the specified value (if any) to the invoker of the method.

##### *try/catch/finally*

Exception handling methods always exist. They have the form:

try (try block: ~> void) (catch series: catch consumer) finally (finally block: ~> void) => void;  
try (try block: ~> void) finally (finally block: ~> void) => void;  
try (try block: ~> void) (catch series: catch consumer) => void;

// internal anonymous class.  
catch consumer => class {  
 (this) (exception: exists) => void;  
 (this) catch (exception variable: vardecl) (catch block: ~> void) => catch consumer;  
}

catch (exception variable: vardecl) (catch block: ~> void) => catch consumer;

Which boils down to try <block> zero or more catch (vardecl) <block> and an optional finally <block> but there needs to be at least one catch or finally.

Catch blocks form a method group with an implicit method that takes exists and rethrows the exception, which is invoked using the standard when an exception occurs in the try block. The variable declared in a catch block unlike other vardecls may not be assigned to.

The finally block is run when the try block exits, after the catch methods are executed when an exception is thrown in the try block, if a return statement (but not yield statement) occurs within the try block, and if a repeat statement occurs within the try block and this is not the first statement of the method. (TODO: always?)

##### *throw exception*

The exception throwing phrase is always available and has the signature throw exception (exception: exists) => void. The propagation of the exception is handled as in C#, though the exception handler behaves as described in .

##### *default*

The default method is always defined and has the form default (type: kind of any) => type. For abstract types, it returns null. For non-abstract types, it returns a new instance of the specified type.

##### *repeat*

The repeat method is defined in all methods and has the form repeat => void. When invoked, execution of the method where it is declared is returned to the beginning of the method. Initializers will not be run for block variables after a repeat, and those variables will keep the values they had when the repeat statement was invoked.

### Block Variable Declarations

A block variable declaration represents a local variable for each invocation of the block that it is declared in.

block-variable-decl::  
 vardecl = initializer-expression  
 vardecl = block

The variable is within the scope of the block that it is declared in (possibly hiding other variables that are visible, but in different scopes). Block variables must be initialized, either via an initialization expression, or an initializer block. Rules about block variable initialization can be found at Initializers.

Block variables follow C#'s rules for garbage collection, including guarantees regarding variables captured by anonymous methods.

### Anonymous Methods

Anonymous methods in Tangent have a slightly different syntax but behave like those in C# and other languages.

anonymous-method::  
 parameter-decl+ => type-decl block

They may capture variables from the scope that they are declared in. In other ways, anonymous method declarations behave as named method declarations do.

### Variable Declaration Expressions

Variable declaration expressions represent a statement scoped variable that may be supplied to variables of type vardecl.

vardecl::  
 vardecl-element+ : vardecl-type

vardecl-element::  
 symbol  
 identifier

vardecl-type::  
 type-decl except =

They allow the statement to refer to the variable by one name, while supplying the variable to a method which can manipulate the variable alongside the rest of the statement using the parameter name. Consider an implementation of the C# using statement:

using .Net.System;

using (resource: vardecl IDisposable) = (initializer: ~> declared type of resource) (block: ~> void) => void {  
 try {  
 resource = initializer;  
 block;  
 } finally {  
 resource.Dispose();  
 };  
}

Unlike block variable declarations, the variable declaration expression cannot be supplied with an initializer. If accessed before it has been assigned, it will be seen as null.