allium

Language Summary
And Example Programs
By Daniel Yost

Version 0.0.2

Table of Contents

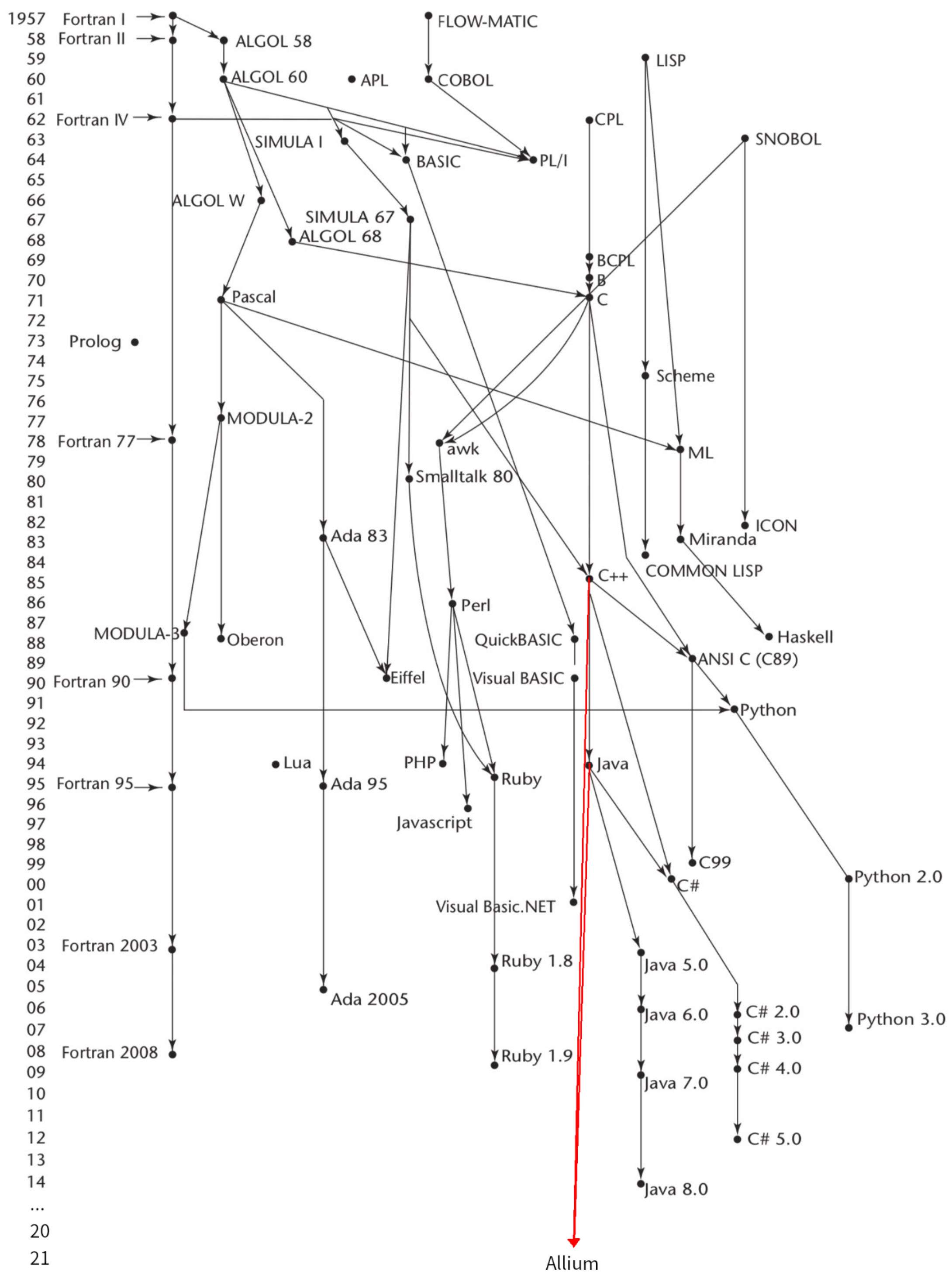
1. Introduction

Allium is a lightweight functional language inspired by Java and built in C++. Like Java, Allium is designed to compile into bytecode, which is then run on the Allium Virtual Machine. This ensures cross-compatibility, security, and ease of installation. However, Allium differs from Java and C++ in the following ways:

1. Allium's primary feature of note is its flexible type declarations, which provide the benefits of a type-safe language while still allowing a lot of customizations.
2. Allium is a functional language, meaning data objects and functions are kept entirely separate. This was done to allow data to be more structured and organized, and to enable some of the flexible type features.
3. Along with flexible type declarations, Allium has a custom type conversion feature. Type conversions allow a programmer to predefine how variables of different types convert into one another, which makes typecasting quick and easy.

For more information on the Allium Project beyond the Allium Language, please visit github.com/danstuff/allium.

1.1 Genealogy



1.2 Hello World

```
tydef Str Char[+];

Str hw = “Hello World”;
print(hw);
```

1.3 Program Structure

There are several elements that give an allium program structure:

1. Custom data types - In lieu of classes, Allium provides a set of rules for defining data objects that can be accessed with the `tydef` keyword.

2. Functions - Like most languages, Allium allows the programmer to abstract and reuse code with functions. Functions are declared with the `func` keyword.

3. Imports - Imports allow multiple files to be linked together into a larger allium program. Using the `import(“filename”)` keyword will cause the program to treat any declarations or code in the provided file as part of the larger program.

4. Namespaces - Accessed with the `nspace` keyword, namespaces are the simplest form of structure a programmer can add to a program. Like in most languages, namespaces provide a wrapper around functions and objects, restricting their access so that a function `foo` in the namespace `bar` is called via `bar::foo()`.

See the next page for a structure example.

```
import("other-file.am");

nspace bar {
	tydef Month Int 1 to 12;
	tydef Day Int 1 to 31;
	tydef Year Int 0 to 9999;

	tydef Date (Month m, Day d, Year y);

	func sameDay(Date d1, Date d2, out Bool o) {
		o = (d1.d == d2.d && d1.m == d2.m);
	}

	func foo() {
		Date today = (5,17,2021);
		Date bday = (9,23,1998);

		Bool o;
		sameDay(today, bday, o);

		if(o) {
			print("Happy birthday!");
		}
	}
}

bar::foo();
```

This file first imports a file called "`other-file.am`", so any code or definitions used there can also be used here. Next, a namespace `bar` is declared, which contains four type definitions and two functions. First, three types are declared: a `Month` type, which can be any integer from 1 to 12, a `Day` type that takes a range of 1 to 31, and finally a `Year` type, which specifies a 4-digit year. These type definitions combine to create a `Date` tuple, which is used in the two functions to check if two dates contain the same month and day. If they do, a birthday message is displayed.

1.4 Types and Variables

Allium has a strong type system, so all type errors are caught and processed during compilation. Since allium has custom types, this means the compiler must keep a list of all user-defined types and valid conversions between them, so that code using those types can be checked for errors. There are five base types: Booleans, Decimals, Integers, Characters, and References. Along with these base types, Tuples and static/dynamic Arrays are also supported. See section 3 for more information on types.

1.5 Statements Differing from Java and C++

Statement Type	Example

Type/Object declaration	`tydef Num Int -9999 to 9999;`
	`tydef Position (Int, Int);`
	
	`Position p = (20, 50);`

Function declaration	`func foo(in Num a, out Num b, inout Num c) {`
	`    //a is pass by value, b by result`
	`    b = a*2;`
	
	`    //c is pass by value and result`
	`    c = c*2;`
	
	`    //there’s no return keyword in allium`
	`}`

Source inclusion	`import(“source-file-name”);`

Pointer/reference declaration	`tydef Num Int -9999 to 9999;`
	`tydef NumRef Ref Num;`
	
	`Num foo = 9;`
	`NumRef bar = foo;`
	
	`bar => 0; //will set foo to 0`
	`bar = null; //will set foo to an empty address`

Namespace declaration	`nspace foo {`
	`    //any code here`
	`}`

Type conversion definition	`tydef Celsius Dec;`
	`tydef Fahrenheit Dec;`
	`tycon Fahrenheit : (Celsius*1.8 + 32);`
	
	`Celsius c = 20;`
	
	`//automatically convert Celsius to Fahrenheit`
	`Fahrenheit f = c;`

2. Lexical Structure

2.1 Programs

An Allium program is constructed of one or more Unicode source files. Source files are always compiled into AVM bytecode. The AVM, or Allium Virtual Machine, is a lightweight virtual machine that I wrote in C++. More information is available on the AVM and the Allium Project at github.com/danstuff/allium.

A program is compiled in five steps:

1. Translation: if the source file is not using the Unicode encoding scheme, convert it to Unicode.
2. Lexical Analysis: translate the input stream from the source file into a stream of tokens.
3. Syntactic Analysis: examine the token stream to validate syntax.
4. Type compilation: Translate each custom data type and their conversions (`tydef` and `tycon` statements) into an AVM-executable bytecode program.
5. Code compilation: Translate the remaining code into an AVM-executable bytecode program.

2.2 Grammars

This specification presents the Allium language syntax where it differs from Java and C++. All strings in this BNF are assumed to be regular expressions evaluated in C++.

2.2.1 Lexical Grammar

```
<Keyword>        ::=   “tydef”          |

                       ”tycon”          |

                       ”func”           |

                       ”nspace”         |

                       “import”         |

                       “print”          |

                       <IOTerm>         |

                       <RangeTerm>      |

<BaseType>       ::=   “Char”     |

                       “Int”      |

                       “Dec”      |

                       “Bool”     |

                       “Ref”

<Operator>       ::=   “\+” | “\-” | “\*” | “\/” | “\%” | “\=\>”

<Container>      ::=   “\(“ | “\)” | “\[“ | “\]” | “\{“ | “\}”

<Letter>         ::=   “[A-z]”

<Digit>          ::=   “[0-9]”

<AlphaNum>       ::=   <Letter> | <Digit> | “_”

<IntLit>         ::=   <Digit> | <IntLit><Digit>

<DecLit>         ::=   <IntLit> | <IntLit> “\.” <IntLit>
```

```
<CharLit>        ::=  “\”(\s|\S)*\””

<Literal>        ::=  <IntLit> | <DecLit> | <CharLit>

<TypeName>       ::=  <BaseType> | <VarName>

<VarName>        ::=  <Letter> | <VarName> <AlphaNum>

<IOTerm>         ::=  “” | “in” | “out” | “inout”

<RangeTerm>      ::=  “to” | “or”
```

2.2.2 Syntactic Grammar

```
<TypeDecl>            ::=  “tydef” <VarName> <TypeStatement>

<TypeStatement>       ::=  <TypeName>      |

                           <TypeBounds>    |

                           <TypeArray>     |

                           <TypeTuple>

<TypeBounds>          ::=  <ToRange> | <OrRange>

<ToRange>             ::=  <Literal> “to” <Literal>

<OrTerm>              ::=  <ToRange> | <Literal>

<OrRange>             ::=  <OrTerm> | <OrRange> “or” <OrTerm>

<TypeArray>           ::=  <TypeName> “\[“ <ArrayTerm> “\]”

<ArrayTerm>           ::=  <IntLit> | “\+”

<TypeTuple>           ::=  <TypeTuple> “\,” <TypeName> <VarName> |

                           <TypeName> <VarName> |

                           “\(“ <TypeTuple> “\)“

<TypeConversion>      ::=  “tycon” <VarEquation> “\:” <VarEquation>

<VarEquation>         ::=  <VarName> |

                           <VarEquation> <Operator> <VarName> |
                                “\(“ <VarEquation> ”\)”
<VarDeclaration>          ::=   <TypeName> <VarName> “=” <DeclarationTerm>
<DeclarationTerm>         ::=   <VarEquation> | <DeclarationTuple>
<DeclarationTuple>        ::=   <DeclarationTuple> “\,” <VarName> |
                                <VarName> |
                                “\(“ <DeclarationTuple> “\)“
<VarAssignment>           ::=   <VarName> “\=” <DeclarationTerm>
<RefAssignment>           ::=   <VarName>  “\=\>” <DeclarationTerm>
<Import>                  ::=   “import\(“ <CharLit> “\)”
<Namespace>               ::=   “nspace” <VarName> “\{“ <Block> “\}”
<Function>                ::=   “func” <VarName> <ArgList> “\{“
                                     <Block>
                                “\}”
<ArgList>                 ::=   <IOTerm> <TypeName> <VarName>    |
                                <ArgList> “\,”
                                <IOTerm> <TypeName> <Varname>    |
                                “\(“ <ArgList> “\)”
<Statement>               ::=   <Namespace>          |
                                <Function>           |
                                <VarDeclaration>     |
                                <VarAssignment>      |
                                <TypeDeclaration>    |
                                <TypeConversion>
                                <Import>
```

```
<Block>              ::=  <Statement> “\;” |

                          <Block> <Statement> “\;”

<Program>            ::=  <Block>
```

2.3 Lexical Analysis

2.3.1 Comments

Allium uses C-style single and multi-line comments. Single-line comments always start with a double slash `//` and extend to the end of the line. Multi-line comments start with a `/*` and end with a `*/`, and, as the name would suggest, can span multiple lines. Comments are discarded and are not turned into tokens.

2.3.2 Whitespace and Unknown Characters

All whitespace is discarded and not turned into tokens, meaning whitespace is not required for a runnable program. Characters or symbols that are not in the symbol table cause the lexer to throw an unrecognized symbol error.

2.3.3 Analysis Method

The Allium lexer will use a greedy algorithm to try and match the longest possible valid input. The algorithm starts at the beginning of the file with a search window of size 0. The size of the search window is gradually increased until the longest possible valid symbol is found. Once this symbol is located, it is added to the symbol stream, and the search window is moved forward and returned to size 0.

2.4 Tokens

Allium has several different kinds of tokens. Here is a list of the different kinds of tokens and what they represent. For a detailed BNF describing the token syntax see section 2.2.1.

Token	**Description**
`<IntLit>`	An integer literal, i.e. 1024 or -256.
`<DecLit>`	A literal number with a decimal place, i.e. 10.1 or 100.0.
`<CharLit>`	Any string of characters, i.e. “Hello.”
`<Operator>`	Add, subtract, multiply, divide, or modulo. Also includes all boolean operations, the assignment operator =, and the reference assignment operator `=>`.
`<Container>`	Parenthesis or square or curly brackets.
`<VarName>`	Any variable or custom type name. Must start with a letter, but can contain numbers and underscores.
`<BaseType>`	Any Allium base type, such as Char or Int.
`<Keyword>`	Any reserved word in the Allium language.

2.4.1 Keywords different from Java and C++

Keywords in Allium are a sequence of characters like a variable name, but they can never be used as variable names because they serve a specific function in the language.

New Keyword	Function
`tydef`	Define a custom type
`tycon`	Define a type conversion
`nspace`	Define a namespace
`func`	Define a function
`import`	Import a code file
`print`	Print to console
`to`	Specify a type range
`or`	Specify a list of type options
`in`	Designate a function parameter as an input
`out`	Designate a function parameter as an output
`inout`	Designate a function parameter as both an input and output.

Removed Keywords
`typedef`
`namespace`
`void`
`nullptr`
All base type names

3. Type System

At its core, Allium has five base data types: Int(eger), Char(acter), Dec(imal), Ref(erence), and Bool(ean). Arrays, and Tuples can also be declared via a `tydef`. Integers, characters, and decimals store actual values. References are addresses that point to an object, but contain no data themselves. Arrays and Tuples are two ways to build types that are composites of other types. Using these core data types, the programmer can build their own custom object types using the `tydef` keyword.

Allium is strongly typed, meaning unless a conversion between two types of variable is explicitly specified, attempting to combine two objects of different types will fail. However, conversions between types can be defined via the `tycon` keyword, thereby adding some flexibility to the traditional strong-type model. The type system is also static, meaning types are bound and checked at compile time. Also, Allium uses name-equivalence, so two variables can only be compared if they have the same type name.

3.1 Type Inference Rules

The type rules for Allium are as follows:

$S \vdash e_1$: T
$S \vdash e_2$: U
T is any type
U is any type with a valid conversion to T

$S \vdash e_1 = e_2$: T

$S \vdash e_1$: T
$S \vdash e_2$: U
T is any type
U is any type with a valid conversion to T

$S \vdash e_1 == e_2$: Boolean

$S \vdash e_1$: T
$S \vdash e_2$: U
T is any type
U is any type with a valid conversion to T

$S \vdash e_1 != e_2$: Boolean

```
S ⊦ e₁: T
S ⊦ e₂: T
T is a base type or an alias/range of a base type
--------------------------------------------------
S ⊦ e₁ <= e₂ : T


S ⊦ e₁: T
S ⊦ e₂: T
T is a base type or an alias/range of a base type
--------------------------------------------------
S ⊦ e₁ >= e₂ : Boolean

S ⊦ e₁: T
S ⊦ e₂: T
T is a base type or an alias/range of a base type
--------------------------------------------------
S ⊦ e₁ < e₂ : Boolean


S ⊦ e₁: T
S ⊦ e₂: T
T is a base type or an alias/range of a base type
--------------------------------------------------
S ⊦ e₁ > e₂ : Boolean

S ⊦ e₁: T
S ⊦ e₂: T
T is a base type or an alias/range of a base type
--------------------------------------------------
S ⊦ e₁ + e₂ : T

S ⊦ e₁: T
S ⊦ e₂: T
T is a base type or an alias/range of a base type
--------------------------------------------------
S ⊦ e₁ - e₂ : T
```

```
S ⊦ $e_1$: T
S ⊦ $e_2$: T
T is a base type or an alias/range of a base type
-------------------------------------------------
S ⊦ $e_1$ * $e_2$ : T
```

```
S ⊦ $e_1$: T
S ⊦ $e_2$: T
T is a base type or an alias/range of a base type
-------------------------------------------------
S ⊦ $e_1$ / $e_2$ : T
```

```
S ⊦ $e_1$: T
S ⊦ $e_2$: T
T is a base type or an alias/range of a base type
-------------------------------------------------
S ⊦ $e_1$ % $e_2$ : T
```

```
S ⊦ $e_1$: T
S ⊦ $e_2$: U
T is a Reference
U is the type that T points to
------------------------------
S ⊦ $e_1$ => $e_2$ : T
```

3.2 Characters, Decimals, and Integers

In Allium, decimal numbers, integers, and characters are all designed to take on different sizes based on their range restrictions. Larger ranges of values cause these value types to claim more bits. The following formula is used to determine how many bits each value should reserve:

$$\mathrm{ceil}(\log_2(o)/4)*4 = b$$

Where o is the number of options a variable can represent and b is the number of bits it stores, which for decimals and integers can be up to 64 bits. This means there are 16 possible options for bit count, because there is one possible option every 4 bits and 64/4 = 16.

Number of Options	Number of Bits
0 to 16	4
17 to 256	8
257 to 4096	12
...	...

For characters, the maximum size is 8 bits, because a character must be in the ASCII format.

3.3 Booleans and References

Booleans and references in Allium are similar in that they both occupy a static number of bits. A boolean represents a single bit value that can be true or false, whereas references occupy 32 bits of data and represent a memory address. References also have a reference assignment operator, `=>`, which is used to assign a value to the variable a reference points to.

3.4 Custom Type Definitions

3.4.1 Variations

Custom type definitions for variables can be as simple as a single character, but are flexible enough to become very complex as well. To define a type, the following format is used:

```
tydef name options;
```

Where `name` can be any alphanumeric non-keyword string. The options are where the possible values that a data type can store are defined. The options can take several forms:

- **Alias**: simply define `name` as an alias for whatever type name you put in `options`. For example, the following will make `Foo` an alias for `Int`:

  ```
  tydef Foo Int;
  ```

- **Bounds**: by specifying a range of possible values in the options after a type name, the new type can be restricted in size. This is done using the `to` and `or` operators.
  - `or` can be used to specify a list of possible options. For example, the following creates a cardinal direction that can have one of four possible values.

    ```
    tydef Direction Char “N” or “S” or “E” or “W”;
    ```

  - `to` is used to specify a range of possible options. For example, the following creates a character that can only be a capital letter:

```
tydef Capital Char “A” to “Z”;
```

- Note that `to` and `or` can be used in tandem to specify multiple possible ranges of values. The `to` operator always has precedence over `or`. For example, the following defines a character that can be either a capital letter or a digit:

```
tydef CapNum Char “A” to “Z” or “0” to “9”;
```

- **Array**: using brackets in tandem with a type name allows the programmer to define an array of that type. These arrays can be either static or dynamic.
    - To define a static array type, simply specify the static size of the array inside the brackets. For example, the following defines a static array of four integers:

```
tydef FourInt Int[4];
```

    - To define a dynamic array, use the + operator inside the brackets instead of a number. For example, a String type of dynamic length can be defined like so:

```
tydef String Char[+];
```

- **Tuple**: using parentheses and a comma-separated list of types allows the definition of a tuple, or a list of varying types. For example, the following specifies a tuple that can store a character (with the name c), an integer, and a decimal number:

```
tydef CharIntDec (Char c, Int i, Dec d);
```

3.4.2 Compilation

Each custom type is compiled into a series of bytecode programs that are designed to instantiate, index, reassign, or convert values to a variable of that type. Each `tydef` statement generates its own instantiation program, along with other programs depending on the variation of `tydef` used:

- **Alias:** The instantiation, indexing, reassignment, and conversion programs simply call the corresponding functions of the type being aliased.
- **Bounds:**
    - Bounds instantiation reserves a certain number of bytes on the stack. The number of bytes reserved is based on the number of possible options specified in the bounds. then assigns a value to them.
    - Bounded base types have no index program.
    - Reassignment simply replaces the previously assigned value.

  - Conversion stores a series of calculations between one value and another.

- **Array:**
  - Instantiation of a static array calls the sub-object's instantiation function N times, where N is the size of the static array. Dynamic arrays are reserved on the heap instead of the stack.
  - Indexing returns the address of the sub-object at the designated position.
  - Reassignment replaces every value in the array.
  - Conversion calls the sub-object's conversion program on every value.
- **Tuple:**
  - Instantiation of a tuple calls each sub-object's instantiation function.
  - Indexing returns the address of the named sub-object.
  - Reassignment replaces every value in the tuple.
  - Conversion only works if all sub-objects are either equivalent or can convert into one another.

4. Example Programs

4.1. Caesar Cipher

```
tydef Alpha Char “a” to “z” or “ “;
tydef Shift Int -25 to 25;

tydef String Alpha[+];

String input = “the quick brown fox jumps over the lazy dog”;

func doShift(in Shift sv) {
	for(Int i = 0; i < input.size; i++) {
		if(input[i] != “ “) {
			input[i] = input[i] + sv;
		}
	}
}

Shift sv = 5;

print(input);
doShift(sv);	//encrypt
print(input);
doShift(-sv);	//decrypt
print(input);
```

4.2. Factorial

```
func factorial(in Int x, out Int y) {
      Int z = 1;
      if(x > 1) factorial(x-1, z);
      y = x*z;
}
```

4.3. Swap Sort

```
tydef IntArr Int[+];

func swapSort(inout IntArr arr) {
      Bool swapped = false;
      do {
            for(Int i = 0; i < arr.size-1; i++) {
                  if(arr[i] > arr[i+1]) {
                        Int buff = arr[i];
                        arr[i] = arr[i+1];
                        arr[i+1] = buff;

                        swapped = true;
                  }
            }
      } while(swapped == false);
}
```

4.4. Binary Tree

```
tydef BinNode (Int value, Ref l, Ref r);

BinNode treeLL = (3, null, null);
BinNode treeLR = (3, null, null);
BinNode treeL = (2, treeLL, treeLR);

BinNode treeRL = (3, null, null);
BinNode treeRR = (3, null, null);
BinNode treeR = (2, treeLL, treeLR);

BinNode treeHead = (1, treeL, treeR);
```

4.5. Doubly Linked List

```
tydef LLNode (Int value, Ref prev, Ref next);

LLNode head = (0, null, null);

func pushNode(LLNode newnode) {
      LLNode current = head;
      Bool atEnd = false;

      while(!atEnd) {
            if(current.next != null) {
                  current = current.next;
            } else {
                  atEnd = true;
            }
      }

      newnode.next = null;
      newnode.prev = current;

      current.next = newnode;
}

pushNode((1, null, null));
pushNode((2, null, null));
pushNode((3, null, null));
pushNode((4, null, null));
```