Introduction (Include your language white paper) (Edward)   
Language Tutorial (A short explanation telling a novice how to use your language.) (James)  
Language Manual (Include your language reference manual) (James, do you have the page file?)   
Language Manual

**Introduction**

Graph algorithms are an extremely ripe domain for networks and relationships of data. Graph algorithms can be very useful in a wide range of applications, including databases, network flow problems, and even language parsing using finite automata. Grape is a language that is designed to make the assembly and manipulation of graphs much more visually comprehensible and syntactically convenient. It should allow its user to implement programs like the Deterministic Finite Automata and the shortest path algorithms such as Dijkstra’s Algorithm easily and concisely. Quick graph descriptions and pattern searching are among the optimizations of the language.

**Language Manual**

1. **Data Types**

**1.1 Primitive Types**

**Integer**: 32-bit signed integer designated by a series of digits

**Boolean**: 1-bit boolean designated by *True* or *False*

**Float**: Signed double precision flowing number designated by a

sign, a decimal and an exponent.

**String**: Series of characters that are enclosed in double quotes

**1.1.1 Examples of Primitives**

Int i = 1;

Bool b = False;

Float pi = 3.14;

String name = "Stephen";

**1.2 Reference Types**

**List** - A collection which is ordered and mutable. It is designated by a

series comma-delimited expressions enclosed in square brackets. List has to be declared using the keyword **List** followed by **<Type of the element>**.

**1.1.2 Examples of Reference Types**

List<Int> a = [1,2,3];

List<String> b = ["Hello", "World"];

**1.3 Graph Types**

**1.3.1 Node**

Node is a container representing a vertex in a graph,

designated by an expression enclosed in single quotes. Node has to be declared using the keyword **Node** followed by **<Type of the node data>**.

**1.3.1.1 Examples of Node**

Node<Int> a = '3';

Node<String> b = '"Hello"';

**1.3.2 Edge**

Edge is an object that represents a directed relationship between two nodes, designated by an expression enclosed in hyphens with a closing bracket representing its directionality. As with a node, the expression contained in an edge can be of any type, for instance an integer containing a cost of traversing that edge. Edge has to be declared using the keyword **Edge** followed by **<Type of the edge data>**.

**1.3.2.1 Example of Edge**

Edge<Int> a = <<-3->>;

**1.3.3 Graph**

Graph is a collection of Nodes and Edges that can be interconnected or disjoint. Graph initialization is designated by a space-delimited path of nodes and edges, enclosed in double angle brackets. Graph has to be declared using the keyword **Graph** followed by **<Type of node data, Type of edge data>**.

**1.3.3.1 Example of Graph I**

Graph<Int, Int> x;

x = <<'3' -3- '4'>>;

Graphs can contain any Nodes of any type and any Edges of any type. More complicated graphs can be described using a comma-delimited series of paths. Reference names can be passed into the graph.

**1.3.3.2 Example of Graph II**

Node<String> a = '"Atlanta"';

Graph<String, Int> cities =

<<a -5- '"Charleston"', a -30- '"New York"', a -100- '"San Francisco"'>>;

Graph initialization can be used to describe paths wherein two edges share a common node between them, for instance:

**1.3.3.3. Example of Graph III**

Graph path = <<'1' -30- '2' -40- '3'>>;

These paths are evaluated from left to right, where the *from*of each subsequent Edge is the same as the *to* of the Edge preceding it. In the above example, the Nodes containing Integers 1 and 3 are both connected to the node containing '2' via the Edges containing 30 and 40 respectively.

Undirected edges can be expressed in the context of a graph using the “- data -” shorthand. They are evaluated as a pair of directed edges pointing to both of the nodes, like so:

<<'1' -3- '2'>>

1. **Operators and Expressions**

**2.1 Variable Assignment**

The ***=*** operator is used for a variable assignment. The right-hand expression is evaluated and its value is assigned to the left-hand typed ID. LHS and RHS must have the same type. This operator will be evaluated right-to-left.

**3.2 Arithmetic Operators**

The arithmetic operators are ***%*** (Modulo), ***\*\**** (Exponent), ***\**** (Multiplication), ***/*** (Division), ***+*** (Addition) and ***-*** (Subtraction). They are all binomial operators. The minus sign can also be used as a unary operation to invert a number's sign (Negation).

**3.3 Relational Operators**

The relational operators are ***<*** (Less than) ***>*** (Greater than) ***<=*** (Less than or equal to) ***=>*** (Greater than or equal to) ***==*** (Equal to) ***!=*** (Not equal to). They are evaluated from left to right. They each require two values which are to be compared and will return *True* if the comparison is truthful and *False*otherwise.

**3.4 Boolean Operators**

The logical operators are *not*, *and*, and *or*. Not negates the subsequent boolean, while and, or, and both return the logical comparison of the values on either side of them.

**3.4.1 Example of Boolean Operators**

Bool T = True

Bool T = False

Bool Yes = t or f

Bool No = not t

* 1. **Precedence and Order of Operations**

Parentheses have the highest priority in the evaluation of expressions. Logical and relational operators have lower precedence than the arithmetic operators, so statements including that include logical or relational operators alongside arithmetic operations will evaluate the arithmetic statement first and then apply relational and logical operators to them in that order. For instance this statement evaluates to ***True***:

**Bool yes = 3 > 5 - 2 and 2 + 2 <= 4**

1. **Programming Structure**

Grape programs are described as a single source file which contains a series of global statements or function declarations which are evaluated from top to bottom.

* 1. **Blocks and Statements**

Grape is an imperative programming language and is designed to be written in blocks, a series of statements which are executed top to bottom. Statements within a block are delimited by semicolons, and can span an arbitrary number of lines.

* 1. **Comments**

Single-line comments are designated by a double forward slash, and are terminated by a new line. Multi-line comments are designated by three forward slashes, and are terminated by another three forward slashes.

**Int a = 5; // Look 'ma a comment!**

**///**

**Welcome to the COMMENT ZONE!**

**///**

* 1. **Functions**

Functions act as a way to compartmentalize segments of your program. Functions are defined by a return type, an ID, and zero or more comma-delimited parameters enclosed in parentheses. The function body consists of a series of statements that must contain a return statement specifying the value to be returned. A function declaration is designated as follows:

**fun return-type function-name(param, param) {body}**

Here is a Grape implementation of Euclid's Algorithm using a recursive functions

**fun Int gcd(Int a, Int b) {**

**Int r = a % b;**

**if (r == 0) {**

**return b;**

**}**

**return gcd(b, r);**

**}**

Functions can be called anywhere in the program body or in any function body, including its own. Grape supports recursion. A function call is designated by the function ID and a series of parameters enclosed in parentheses:

**Int a = gcd(10, 15);**

1. **Control Flow**
   1. **Conditionals**

Grape supports if statements that may contain an optional else condition to execute if the given condition is false.

**if (r == 0) { return 3; }**

**else { return 2; }**

* 1. **Loops**

***while*** loops are designated by a looping condition and a block to be executed as long as the condition is truthful. They are designated as follows:

**while (x < 5) {**

**x = x + 1;**

**}**

***each*** loops allow the user to iterate over Lists. Within the body of an each loop, there is special local variable called ***this*** which stores the current item in the List.

**List a = [3,1,4,1,5,9,2,6];**

**each (a) {**

**this = this + 1;**

**}**

1. **Standard Library**

The Grape standard library provides useful built-in methods for manipulating the List, Dict, and Graph types, as well as for standard I/O:

***print(a)*** - writes a to stdout.

* 1. **List Methods**

***l.append(x)*** - append x to the end of the List and return the modified List.

***l.clear()*** - remove all items from the List and return the modified List

***l.copy()*** - return a deep copy of the List

***l.insert(i,x)*** - insert variable x at index i and return the modified List

***l.pop()*** - remove and return the last element from the List

***l.remove(x)*** - remove the first instance of x and return the modified List

***l.reverse()*** - reverse the entire list, and return the reversed List

***l.size()*** - return the number of elements in the List

* 1. **Dict Methods**

***d.size()*** - return the number of keys in the Dict

***d.key(x)*** - return the key of the value x, if it exists in the Dict.

***d.remove(x)*** - remove the key x and its value from the Dict

* 1. **Graph Methods**

***g.size()*** - return the number of Nodes in the Graph

***g.root()*** - return source Node of the Graph

***g.leaves()*** - return a list of Nodes with only one incoming Edge

***g.neighbors(x)*** - return a List of Nodes adjacent to Node x

***g.find(x)*** - return the list of all Nodes that contain the value x

***g.empty()*** - return ***True*** of the Graph is empty, otherwise return ***False***

***g.switch(x,y)*** - switch Nodes x and y in a Graph

**Project Plan** (Timmy)

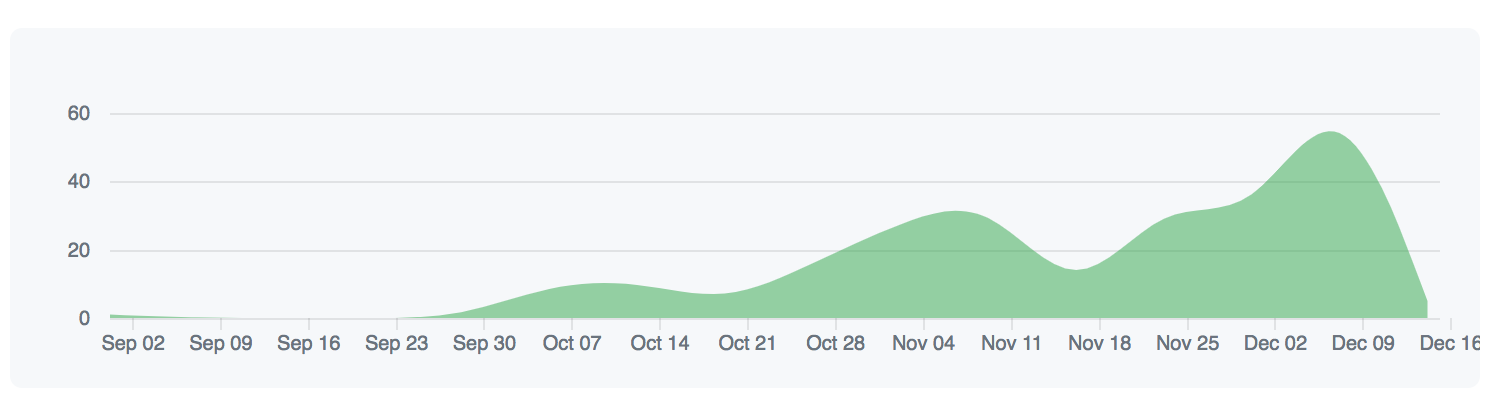
**Weekly meetups:**

In order for the group to slowly establish the codebase, we scheduled a meeting once a week with our TA, Dean Deng. We also met at least two more times a week to discuss the project goals. At the start, these meeting were primarily discussions of broader concept, of the syntax, and the functionality of the language and throughout the semester, the team incrementally met more and more often, shifting from discussion to delegating technical tasks to each individual.

**Workflow:**

All of the project is located within Docker, because it allows us to streamline to development among teammates. However, while programming, members can either choose work on local repository that is mounted onto the docker or the docker container itself. As we make changes throughout the project, we uses Git version control and Github, and at times, worked from new branches to test new features. To ensure that there weren’t erroneous code being pushed onto the shared repository, every member can only push compilable code.

The graph below plots the project’s commits over the course of the semester. As one can tell, there are incrementally more and more commits at the second half of the semester, as we finalize the language and make necessary changes for



**Programming Style Guide:**

1. Always use tabs when we need to indent.
2. For the C library, we strive to make reasonable function names to ensure clarity
3. At the start of a git commit message, denotes whether the commit is a “feat” or “fix”.
4. Comment on particularly intricate code to convey meaning clearly.

**Project timeline:**

During the semester, we scheduled milestones to achieve throughout the semester adn strived to actualize them. In reality, the process turned out to be more of a back-and-forth trajectory as a lot of times, we realize that there are previous mistakes or designs that we wanted to alter.

|  |  |
| --- | --- |
| Milestones | Time |
| LRM, Scanner done, elementary Parser | 10/15 |
| Parser, AST, SAST, “Hello World” | 11/18 |
| Semantically checked types (edges, nodes) | 11/25 |
| Edge, Node, List typing in codegen.ml | 12/2 |
| Graph type in codegen.ml | 12/10 |
| Writing C library, Linking C library | 12/11 |
| List indexing, Dot notation, Overloading functions | 12/12 |

**Roles and Responsibility:**

**Language Guru - James Kolsby**

The language guru, a true visionary, dictates the utility and aesthetics of the language, Grape. As he caressed the purple marbles bursting of nature’s nectar, he gasped in Archimedean fashion: “A grape cluster is basically a tree, a graph!”

**Project Manager - Po Yu (Timmy) Wu**

The project manager primary deals with humans, the only part in the project that are, unfortunately, not made of booleans. He strives to keep the project afloat by keeping track of the schedule, coaxing the hearts and minds of self-absorbed Columbia students, and, oftentimes, making announcements in a messaging group that sometimes seems to be constituted of only he, himself and “hi”.

**System Architect - HyunBin (Edward) Yoo**

The system architect decides the software development environment that the group operates in and just like Meryl Streep in the Sophie’s choice, the architect knows all too well that no matter what he chooses, he and his team would carry the burden, forever haunted.

**Tester - Nick Krasnoff**

The tester ensures that the code that we so diligently farmed truly comes to fruition and works the way we wanted it to. His responsibility includes building test suites, writing test scripts, and having inhumanely amount of patience for his trial and error was full of disappointments and, well, errors.

**Development Tools and Languages:**

* + Communication: SMS messages, Slack
  + Version Control: Git, Github
  + Programming Editor: Vim, Sublime Text
  + Languages: Ocaml, C Language, Shell Script
  + Documentation: Github Markdown

**Project Log:**

**Architectural Design**

Grape’s compiler is constituted of several files, with functionalities of their own. There are a primary repository, src, that holds the translator of scanner, parser, ast, sast, semant, codegen and a c library that is linked to codegen for functions.

|  |
| --- |
| example.grp |

|  |
| --- |
| scanner |

|  |
| --- |
| parser |

|  |
| --- |
| ast |

|  |
| --- |
| sast |

|  |
| --- |
| semant |

|  |
| --- |
| codegen |

|  |
| --- |
| executable |

* Give block diagram showing the major components of your translator

Components

**Scanner**

The scanner takes a stream of ASCI texts and turned them into tokens. Whitespaces are discarded and texts in the comment syntax are not transformed. At this stage, if a stream of text is not syntactically correct, the executable would return a parsing error.

**Parser, Ast and Sast**

The parser takes in the tokens produced by the scanner, and convert into a Abstract Syntax Tree.

**Semant**

**Codegen**

**C Library**

* State who implemented each component

Scanner: James Kolsby, Nick Krasnoff, Po Yu (Timmy) Wu

Parser, Ast and Sast: Edward Yoo, James Kolsby, Nick Krasnoff, Po Yu (Timmy) Wu  
Semant and Codegen: James Kolsby, Nick Krasnoff, Po Yu (Timmy) Wu

C Library: Edward Yoo, Nick Krasnoff, Po Yu (Timmy) Wu, Edward Yoo

Test Plan (Nick)

* Show two or three representative source language programs along with the target language program generated for each: dfa, dijkstra?
* Show the test suites used to test your translator
* Explain why and how these test cases were chosen
* What kind of automation was used in testing
* State who did what

Lessons Learned (Everyone)

* Each team member should explain his or her most important learning

Edward (HyunBin) :

James:

Nick:

Timmy (Po Yu) :

* Include any advice the team has for future teams (Timmy)

Appendix

* Attach a complete code listing of your translator with each module signed by its author  
  (Do not include any automatically generated files, only the sources.)