The solution

To bad code

Lucas Standen

7949

ABSTRACT

1. Reading this document

This document is writen in roff and can be found online at:

https://github.com/standenboy/school/tree/master/comp/lucas-standen-NEA/writeup

It is using the ms macro of troff. It can be compiled using the Makefile, or make.sh. A table of contents has been generated using pdftocgen, it is embedded into the pdf, most pdf readers have a button to open it (firefox has it in the top left, in zathura press tab to view it).

A note on formating of the roff, the text is limited to 100 characters per line and is writen in plain ascii, no utf8 emojis and the like. Code snippets are left in plain text, while full files are converted to a ps file via https://carbon.now.sh/ they should be 150mm ^ 2 (as ps is a vector format this wont lower quality, you might need to zoom in though) and then have there source linked above them; assuming they are from a file and not a small example.

2. Analysis

2.1. The current problem

For general small and simple projects, I write in C. However this leads to hours of debugging due to segfaults, and memory leaks. Due to the languages manual memory management the programmer is required to know so much information about the hardware they write for, and the second anything goes wrong, it is vague on how to fix things.

I need a language that stops me from shooting myself in the foot

C has been standard for many decades now and its age is showing, it lacks many modern features like OOP, or higher level functional abstractions, that have become common in modern years due to there helpfulness. This is not to fault C's achievements either, the language is my personal choice for most projects for a reason, it's fast and powerful; any solution I make should not cut that away.

2.2. A solution

Zippy LANG

A next generation language, for general use. Designed for keeping code simple, neat and readable. It will be similar to functional languages, known for there strict ability to keep code safe and practical. The language should be interpreted like python, perl and lisp, to allow for easy debugging tools.

The goal of Zippy is to make codding easier, while remaining fast, with a interpreter writen in C.

2.3. What is a programming language

2.3.1. A very simple explanation

At its lowest definition a PL is a set of specific words, that when given to a computer in the right order have a reproducible behaviour. A more human way of saying that, would be its how we control computers.

2.3.2. Why are there so many

When someone is looking at code it can often be seen as just that, however there are hundreds of languages that all take the idea of "code" in very different ways. Some are designed for specific hardware, some are designed for making general use programs while others are highly specialized. It is important to see "code", as more than just one overarching term and instead see where the code is being used, and evaluate it from that.

2.4. Researching, and getting a scope of the project

Before I start to design a language i should first find examples of others and find what i want my language to be like.

I'd like my language to feel modern so i should take inspiration from what other modern languages do, however on the backed i want my language to be stable and fast, for that i should look at older projects.

2.4.1. Examples of older similar projects, that are a good base for my language

2.4.1.1. Python

Python is a high level OOP language that was designed in 1991. It was made to make programming easy while still being able to use some of C's functions. Although it has become standard for many use cases, it is slow and inefficient, and very bloated.

https://www.python.org/

Zippy should take pythons high level abstractions, as they make programming very easy and it should try and take notes from its libraries as they are mostly well written, and well documented.

2.4.1.2. Lisp

Lisp is the second ever programming language, developed at MiT, it is the first functional language, creating many common features like higher order functions, recursion, and garbage collection. It is generally not used any more as it feels old compared to other functional languages, like Ocaml or Haskell.

https://lisp-lang.org/

Zippy should try to take alot from the syntax of lisp, () make it easy to see what parts of code will effect what, and make things easy to parse.

2.4.1.3. Perl

Perl is scripting language designed for use in linux, when bash is too slow, or not suited for the job. Perl is often described as the glue of the universe (see xkcd https://3d.xkcd.com/224/). Its syntax is quite strange however and it is slow. Making it poorly suited towards general use.

https://www.perl.org/

Zippy should take from perls minimalism, it is a small language that is of a similar size to bash or zsh, while feeling closer to python. If Zippy can achieve a similar small size, while remaining powerful I will be happy.

2.4.2. Examples of new similar projects that are also a good base

2.4.2.1. Gleam

Gleam is a modern language releasing in the past 5 years. It is highly functional, with no mutable data, no traditional loops. Instead recursion can be used to replace alot of these features. Gleam compiles to erlang/Beam bytecode, much like java to the jvm, and doing this has made Gleam a highly scalable language with good library support out the box.

https://gleam.run/

Zippy should take from the functional elements of Gleam, as they keep programs safer, however Zippy should not remove all procedural elements, as for loops are very helpful

2.4.2.2. Haskell

Haskell is another modern functional language known for being very complicated, however incredibly powerful. Its syntax feels very mathematical, and incredibly terse.

https://www.haskell.org/

Perhaps Zippy could learn from Haskell, as it provides functional and procedural elements, making it a well rounded language

2.4.2.3. Hare

Hare was designed to be a 100 year language, and thus stability is its main goal, it is not set to have a syntax change any time soon, and it has strong emphasis on memory safety. It fits into the same part of the tech stack as C, and thus it can be used for some very low level work.

https://harelang.org/

I think Zippy should have a strong emphasis on stability, much like Hare, to many times have I segfaulted due to a tiny mistake. Zippy should also look to Hare's small size, you can buy a copy of Hare on a

SINGLE 3 1/2" FLOLPY

This is something I too should try to achieve.

2.4.3. What should be taken away from these languages?

I was already leaning towards functional programming when I started this project however now I believe it's the only option for producing safe applications. Zippy will be a functional language with a strong emphasis on recursion.

I also believe that I should take size of the interpreter into account, as this is important for keeping the project manageable and consistent.

And finally I think that syntax should be inspired by Lisp, although Lisp itself can be a messy language, with the right changes I am confident that I can make a attractive language for the 21st century.

2.5. Clients

In a project of this nature, the Client is every programmer alive; which is a pretty large scope. To narrow this down as much as possible, I will interview a small handful of people throughout the project, of different skill levels.

2.5.1. Client 1, Amy C

My first client is a friend of mine, Amy C, she is a confident programmer who has completed many complicated projects. I am choosing her as a client as she can give me technical feed back on my project and its function/utility.

2.5.2. Client 2, Rayn M

Another friend of mine, Rayn M, is a technical computer user, however he does not know how to program at a high level. He will be a good client as he can show me how my language looks to some one who doesn't understand the inside workings, helping me design the structure of the code.

2.5.3. Client 3, a normie

some stuff about how the normie finds the completed project.

2.5.4. Client 4, myself

I've wanted to take out a project like this for a long long time, and this is the perfect opportunity to do so, I will be assessing myself along the way of this, building the project to my personal specification.

2.6. Questionnaires

It is important to get feedback from end users, so I will take multiple questionnaires throughout the project. I will then use them to slightly edit the requirements of my project this should make the final outcome more helpful and what people want.

In the section bellow you will find questionnaires from the analyses stage of my project.

2.6.1. Questionnaire 1 for Amy C

[30th April 2024] answered by Amy, see pull request she left

2.6.1.1. What do you find the most important in a language? (eg: speed, readability)

Speed, readability, debugging ease and disk space efficiency.

2.6.1.2. What tools are important for a language to have? (eg: pkg-manager, IDE integration)

IDE integration (things like tab complete and debugging tools), a package manager, and the ability to interact with the user through the command line easily.

2.6.1.3. What features do you like from other languages (eg: C's advanced memory management, haskell's terse syntax)

The ability to pass the memory reference of an object or function and a collection of built-in or standard functions like "print", "split", or "sort".

2.6.1.4. What do you want to program in this language (eg: websites, low level systems)

Lightweight command line tools and web back ends.

2.6.1.5. Do you intend to use graphics in the programs you write?

No.

2.6.1.6. Would you prefer a language that focuses on ease of use, or power of the code?

I like a good balance between the two.

2.6.1.7. What were your last 3 projects? (could they have been written in Zippy?)

A website, a small command-line tool and a midi keyboard (program runs on a Raspberry Pi Pico).

2.6.1.8. How many languages would you use on a single project? (could Zippy be used in your codebase?)

I try to use as little languages in a project as possible, so likely not in an existing project.

2.6.1.9. Do you care for low level control, or would you prefer high level abstractions?

I think low-level control is very important, but high-level abstractions are convenient, so a good balance between the two is best.

2.6.1.10. Would you be happy to develop libraries for things that aren't already implemented (eg: an SQL library)

Potentially if it is simple enough to implement new things.

2.6.2. Notes from questionnaire 1

Some of the key things that I'm taking away from this first questionnaire, are my client/users initial needs and use cases. I think it's clear my language can be of assistance to my client, Zippy will be a good language for web back ends and small command line tools, which my client expressed interested in.

I find the fact my client is worried by executable size interesting, however I doubt it will be an issue; a ballooning code-base is unlikely as only one person is writing the project.

I am also taking on the fact that my client wants good command line tools, so a pkg-manager and bundler should be a priority, perhaps they could be written in Zippy after the interpreter is done.

2.7. The first elements of the project

At this stage I can say that I'm confident in my project and its scope. I have a goal in mind for it.

The key things to take away from this section are:

---- Make a high level language with a useable set of features, to replace C in many situations.

---- Keep the language readable and easy, with powerful tools available.

---- Ensure the language is well supported with tools like a pkg-manager.

2.8. Moddeling

In larger projects, when a programmer needs a data structure that the language they are writing in doesn't provide, they will need to make their own.

Bellow are a few examples of these data structures that C doesn't already provide.

2.8.1. Linked lists

this is an alternative implementation of a list, where you store some data, and the memory address to the next node. Then you can move through the list by reading the data then reading the data of the next node, and then repeating until the 'next' part of the node is empty.

A diagram showing this can be seen here:

In C this is easy to implement as you can find a memory address very easily with '&' to find where a bit of data is stored. I will need to use a 'struct', which is a bit like a class in C (however you can't attach a function to it). A simple implementation looks like this:

```
typedef struct ll {
	void *data; // the data of the node
	ll *next; // the next node

} ll;
```

The pro's of a linked list are the fact that they can have data appended to the start or end easily by changing the root node, or the next node.

Linked lists have a few downsides, for example you can't move through them backwards, and unless you store it on its own, you cant find the length of it in a fast way.

In my project I would like to use linked list in the AST (see later sections for info), and to store lists in the language.

2.8.2. Dictionaries

A dictionary is a simple data structure that just stores, a bit of data, and a number or string to identify it. A dictionary like a linked list can be implemented with a struct in c like so:

```
typedef struct dict {
	void *data;
```

```
    int id;

} dict;
```

In my project I think I could use a linked list represent a Zippy variable and an ID that i can use to identify it, this could make execution faster as i can compare ID's rather than string values

2.9. Prototyping hard features

2.9.1. Abstract Syntax Trees (AST) theory

In a programming language many abstract data types will be used to allow the code to execute, however I think the hardest part of this is an abstract syntax tree. This is a data structure that holds the code in an ordered form that can be analysed and executed in a simple way. It is a tree structure, with the top node being a root and all lower nodes being things needed to calculate the root. It can be used not only for code but also for mathematical expressions. I think the easiest way to show it is via a mathematical example

Take the follow expression for example:

`(1 + (10 * (3 - (2 * 4))))`

We know that this is equal to -49

However for a computer this is far harder to understand. This is because it has no understanding of order of operation

To solve this we use an AST (abstract syntax tree)

When you solve that expression you know to start with (2 * 4), then 3 - the answer to that and so on

We can represent the steps as a tree like so:

```
   *
 /   \
2     +
     /\
    2 2
```

*[Evalutates to 2 * (2 + 2)]*

As you can see, you need to evaluate the expression in the most brackets first, then the next, and so on, working you way up

You can evaluate code in a similar way, treating each operation (such as +-*/) as functions, doing the most deeply nested function first, then working up. Each expression can be represented in this tree, then to show a whole program you can create a list of trees

2.9.2. Implementing AST's

As a prototype i will make a program that can take mathematical expressions and evaluate them, and allowing for functions (in the form f(x)). It will do this via AST's

This prototype takes 173 lines of code, it takes a string as a cmd line argument then converts it into an abstract syntax tree, and finally it executes it. This is just a simple prototype and thus it is small in scope. It can only do simple operators (+-*/) and requires literal values to be surrounded by [] so it knows its not another expression to evaluate.

https://github.com/standenboy/school/tree/master/comp/lucas-standen-NEA/code/proto/ast

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef struct ast_node ast_node;

typedef enum op {
	ADD = 0,
	SUB = 1,
	MUL = 2,
	DIV = 3,
} op;

typedef struct ast_node {
	op operation;
	int realLeft;
	int realRight;
	ast_node *right;
	ast_node *left;
} ast_node;

void freeAst(ast_node *head){
	if (head->left != NULL)
		freeAst(head->left);
	if (head->right != NULL)
		freeAst(head->left);
	free(head);
}

int exec(ast_node *exp){
	if (exp->left != NULL){
		exp->realLeft = exec(exp->left);
		freeAst(exp->left);
		exp->left = NULL;
	}
	if (exp->right != NULL){
		exp->realRight = exec(exp->right);
		freeAst(exp->right);
		exp->right = NULL;
	}

	if (exp->operation == ADD)
		return exp->realLeft+ exp->realRight;
	if (exp->operation == SUB)
		return exp->realLeft - exp->realRight;
	if (exp->operation == MUL)
		return exp->realLeft * exp->realRight;
	if (exp->operation == DIV)
		return exp->realLeft/ exp->realRight;
	return 0;
}
```

Above is the code for the AST, it stores an operation (which is just an integer), and it stores a real left and real right value, along side two other nodes. The real values are integers, this would be the 2 numbers in reference in the expression. The 2 nodes are a recursive data structure, much like putting an object of a class inside the definition of that class itself. They are used to store values that may still be expressions, for example (+ [1] (+ [1] [1])) the second part of this expression would be in the "right" variable. When code is executed I can check if "left", or "right" are null and if they are i know that i am at the lowest expression that is only literal values. Then I can execute that node and work my way up the tree.

The exec function will execute the operation, unless there is a deeper node, if there is a deeper node, then it executes it, and places the result in the right or left spot respectively.

Expressions are taken as input with the following code, and converted into the AST:

https://github.com/standenboy/school/tree/master/comp/lucas-standen-NEA/code/proto/ast

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "astg.h"

void getBrackets(char *in, int bpos, char *out){
	// gets the content of the brackets that open at bpos
	if (in[0] != '(' && in[0] != '[')
		out = NULL;

	char *input = malloc(strlen(in) + 1); // cpy in for mem safety
	char *Pinput = input;
	memcpy(input, in, strlen(in) + 1);

	int i = 0;
	while (i != bpos){
		input++;
		i++;
	}
	i = 0;
	int depth = 0;
	while (input[0] != '\0'){ // loop through input
		out[i] = input[0];
		if (input[0] == '(' || input[0] == '[')
			depth++;
		if (input[0] == ')' || input[0] == ']')
			depth--;
		if (depth == 0){
			i++;
			break;
		}

		input++;
		i++;
	}

	out[i] = '\0';
	free(Pinput);
}

int getContents(char *brackets){ // get all the content in brackets
	int i = 0;
	char *num = malloc(strlen(brackets));
	while (brackets[0] != '\0'){
		if (brackets[0] != '[' && brackets[0] != ']'){
			num[i] = brackets[0];
			i++;
		}
		brackets++;
	}
	num[i] = '\0';
	int out = atoi(num);
	free(num);
	return out;
}

ast_node *genAst(char *expression){
	ast_node *out = malloc(sizeof(ast_node));

	/* take the expression
	 * get the first operation from it
	 * get the first number from after the expression
	 * if that number is another expression{
	 *	grab everything inside its braket and then call this function on it
	 * }
	 * get the second number from the expression
	 * if that number is another expression{
	 *	grab everything inside its braket and then call this function on it
	 * }
	 */

	int i = 0, j = 0;
	int currentTokenVal = -1;

	char *currentToken = malloc(strlen(expression)+1);
	while (expression[i] != '\0'){
		if (expression[i] == '(' || expression[i] == '['){
			getBrackets(expression, i, currentToken);
			currentTokenVal++;

			int depth = 0;

			if (i != 0){
				while (expression[i] != '\0'){
					if (expression[i] == '(' || expression[i] == '[')
						depth++;
					if (expression[i] == ')' || expression[i] == ']')
						depth--;
					if (depth == 0){
						break;
					}
					i++;
				}
			}

			if (currentTokenVal == 0){
				switch (currentToken[1]) {
					case '+':
						out->operation= ADD;
						break;
					case '-':
						out->operation = SUB;
						break;
					case '*':
						out->operation = MUL;
						break;
					case '/':
						out->operation = DIV;
						break;
				}
			}

			if (currentTokenVal == 1){
				if (currentToken[0] == '(')
					out->left = genAst(currentToken);
				else {
					out->realLeft = getContents(currentToken);
					out->left = NULL;
				}
			} else if (currentTokenVal == 2){
				if (currentToken[0] == '(')
					out->right= genAst(currentToken);
				else {
					out->realRight = getContents(currentToken);
					out->right = NULL;
				}
			}
		}
	i++;
	}
	free(currentToken);

	return out;
}

int main(int argc, char **argv){
	if (argc < 2) // die if no argument given
		exit(1);

	ast_node *head = genAst(argv[1]);

	printf("%d\n", exec(head));

	freeAst(head);
}
```

Here is an example input and output:

./ast "(+ (- [3] [1]) (- [3] [1]))"

$\boxed{4}$

Note the [] used to tell the program where the literal values are.

Overall this was a relatively successful prototype, however it isn't fully functional as a language it has fit the design.

The rest of the code is the process of converting the string input to literal values and inserting them into the AST

2.9.3. Feedback

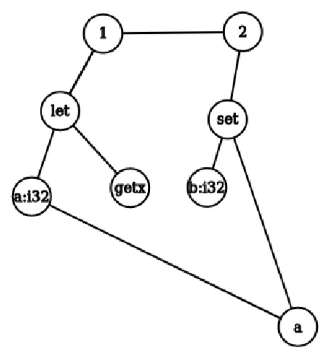
From my first Client (Amy C), she said that putting the numbers inside square brackets was inconvenient and annoying and it would be better if the numbers were separated by spaces instead of separate square bracket surrounded literals.

As this is a prototype I won't fix this issue, however in the actual language this is a needed feature that I will be implementing.

2.9.4. Mixing linked lists and AST's

Mixing these 2 data structures together you can repressent an entire program. A linked list of AST's is how Zippy will repressent all code the user writes

Here is an example of this:



In this example the linked list is represented by the numbers seen at the top, and the AST's are the tree's moving down.

As you can see when a value is referenced that is from a different AST the tree will link to another one. This will work the same for function calls, however instead of linking to value definitions it will link to function definitions.

2.10. Objectives

2.10.1. An interpreter for the Zippy language

2.10.1.1. Linked list of AST's

All of a loaded program should be represented as a linked list of individual AST's, The developer should be able to access the AST for easy hacking. Functions can be represented as a pointer to another part of the list.

2.10.1.2. A lisp like syntax

This is to ensure the language can be parsed quickly, and is easy to write.

2.10.1.3. Functional language

This language should lean into the functional programming paradigm, taking inspiration from other functional languages such as lisp, and gleam.

2.10.1.3.1. Recursion

Zippy must support recursive algorithms being implemented into it, this will make the AST, have nodes linking back to parent nodes in a linked list.

2.10.1.3.2. Higher order functions

Zippy must support the usage of higher order functions, this will mean the AST needs to have an unlimited depth as otherwise the limit would be quickly reached, it can't be hard-coded, it must be dynamic.

2.10.1.4. Performance

The interpreter must be fast and memory efficient, the language is designed to work as an alternative to C, one of the fastest languages of all time, the interpreter must be fast, however memory footprint is not as much of a requirement.

2.10.1.5. Safe

Code that the user writes must be safe, and not prone to errors. This can be handeled via the strong syntax checker and type safety.

2.10.2. Standard library for Zippy

2.10.2.1. io

The language must have a simple to use I/O library to make outputs easy.

2.10.2.2. string

The language should have a sting library that provides a string type, and many complex algorithms that can be applied to them (concatenation, insertion, appending, splitting, stripping).

2.10.2.3. sorts

The language should have a sorting library that provides algorithms used for sorting (like merge sort).

2.10.2.4. graphs

the language must have a graph library, that allows for easy creation and working with graphs, it should provide many algorithms to help traverse these graphs

2.10.3. Tooling for the Zippy language

2.10.3.1. zpypkg

Zippy must provide a package manager, that allows code to be shared between multiple users, easily. It should sync projects via git and allow them to be stored on any git host the user likes.

2.10.3.2. Syntax checker

Zippy shouldn't have a built in syntax checker, instead it should be something that can be run independently of the interpreter, this means that a lot of the checking that interpreted languages do, can be done once by the developer, before shipping the app, as opposed to every time the program is run, which brings down performance.

2.10.4. Integration with C, via a C API

2.10.4.1. C API

You should be able to execute a string of Zippy code in C using a library that is linked with interpreter. This could allow Zippy to be used as a configuration language like Lua.

2.11. Desirable features

If time allows I'd like to add some of the following features to flesh out the language:

2.11.1. Raylib support

Raylib is a powerful game engine for C, however it has been ported to most languages under the sun due to how simple it is. If I have time, porting Raylib to Zippy would make the language far more useable, as it can be use for graphics programming.

https://www.Raylib.com/

2.11.2. Vim integration.

Zippy should have integration with the Vim editor for syntax highlighting, this can be done via generating a linked list of AST's then colouring function calls a specific colour, and variables another, etc, etc.

2.11.3. LSP

A LSP (language server protocol), is used in code IDE's to auto complete code for you, I'd like one for Zippy. Although I am unsure as to how to tackle this. I believe a program called treesitter can be helpful for this.

2.11.4. Networking sockets

If possible I'd also like to provide bindings for unix network sockets, however this would be very difficult, as I would need to allow Zippy stucts to be directly converted to C stucts, when executing ELF symbols (Parts of an execuable file).

3. Design

3.1. Language specification

Like any other programming language Zippy needs to have a defined syntax, as mentioned in the objectives section of Analysis, I want the language to follow a lisp like syntax.

I also believe higher order functions should be taken as standard and many core functions will use them.

3.1.1. Data types

3.1.1.1. Basic types

i32 - signed integer of size 32 bits

u32 - unsigned integer of size 32 bits

i64 - signed integer of size 64 bits

u64 - unsigned integer of size 64 bits

char - single ascii code

float - standard C float

3.1.1.2. Advanced types

function - a function that can be used

struct - initializer for an object

obj - instance of a struct

generic - should be avoided, removes checks for data types when inputting values to functions will cause many runtime errors, however when absolutely needed it is useful.

err - error type, can be given to terminate function to bring down the program, or have info drawn on the error.

3.1.1.3. Arrays

Arrays can be show like so:

x:type[]

With x being the variable name, type being the type of variable, and [] showing its an array

All arrays are dynamic, represented by a linked list on the back end.

3.1.1.3.1. Strings

Strings, like in C are arrays of chars

3.1.2. Built in functions

3.1.2.1. defun

```
(defun a:type b:type returntype
    ...
    ...

)
```

Returns a function that take A and B as an argument (fixed types), and returns a value of returntype.

3.1.2.1.1. Errors

Can syntax error.

3.1.2.2. let

(let x:type value)

3.1.2.2.1. Errors

Can syntax error.

Creates constant x of type type to value.

3.1.2.3. set

(set x:type value)

Creates/recreates the variable value of x to value.

3.1.2.3.1. Errors

Can syntax error.

3.1.2.4. if/elif/else

(if condition function)

(elif condition function)

(else function)

Executes the function provided if the condition is true.

Elif works the same, except only if the previous if statement is false.

Else executes only if all previous statements were false.

3.1.2.4.1. Errors

Can syntax error.

3.1.2.5. for

(for i (condition) function)

Runs the function while the condition is true, and increments i every time the function is called.

3.1.2.5.1. Errors

Can syntax error.

3.1.2.6. while

(while condition function)

Runs the function if the condition is true, keeps running until it is false.

3.1.2.6.1. Errors

Can syntax error.

3.1.2.7. symbol

(symbol a:type b:type c:type returntype name:char[] elf:char[])

Returns a function that takes arguments A, B, C (of fixed types), the name of the function, and the file path of the elf.

3.1.2.7.1. Errors

Can syntax error.

Can throw other errors, via returning an err type.

3.1.2.8. struct

```
(struct
    _a:type
    b:type
    c:type

)
```

Returns a struct (a class) that can have a default value, the default value can be set using an underscore in front of the variable name. A default variable will be returned if you use the struct without specifying a value.

Use let to assign it a name:

```
(let a:struct (struct
    ...
    ...
    ...
    )

)
```

Then to instantiate the struct use the following:

(let a:obj struct)

The struct argument needs to have been set prior.

To read from the struct use this:

(if (= a.num b.num) ....)

Or if you have set a default value you can use the following:

(if (= a b.num) ....)

3.1.2.8.1. Errors

Can syntax error.

3.1.2.9. Arithmetic operations

Simple operations

(+ a b) returns a + b

(- a b) returns a - b

(* a b) returns a * b

(/ a b) returns a / b

3.1.2.10. Comparison

All return true or false

(= a b) returns if a = b

(!= a b) returns if a != b

(> a b) returns if a > b

(< a b) returns if a < b

(=> a b) returns if a => b

(=< a b) returns if a =< b

3.1.2.10.1. Errors

Can syntax error.

3.1.2.11. cast

(cast a:generic type:char[])

returns a but cast to data type type, which is a string.

3.1.2.11.1. Errors

Can syntax error.

Can throw errors, via err, if value can’t be cast to a given type.

3.1.2.12. typeof

(typeof a:generic)

returns in a string the type that variable A is.

3.1.2.12.1. Errors

Can syntax error.

3.1.2.13. terminate

(terminate error:error)

Kills the program at the current point, frees all related memory, prints error info stored in error.

3.1.2.13.1. Errors

Can syntax error.

3.1.2.14. return

(return a:type)

Must be used in defun, returns "a" from the function, "a" must be of the functions return type.

3.1.2.14.1. Errors

Can syntax error.

Can throw errors via err.

3.2. Questionnaire 2 for Rayn M

3.3. What language do you use to make a programming language

As mentioned before Zippy will be written in C, with some parts being written in Zippy itself. I will try and keep most dependencies/libraries to a minimal to make the project easier to manage.

3.3.1. What is C?

C was made by Dennis Ritchie, in 1972 at AT&T's bell labs. It was designed to make programming low level systems far easier than it had been before. It was used to create the unix operating system which would go on to inspire most modern operating systems in some way. (macos still has code from the original release of C+unix).

The language quickly caught on outside of bell labs after more available releases of unix arrived such as bsd 4.4, sun os and GNU. It was found to be able to do all the things that you could do in ASM however with far less a headache.

3.3.2. Why is C?

As mentioned C can do anything that ASM can do, meaning it is lightning fast and can take advantage of direct memory access. This allows you to make very fast lightweight executables that can rival the performance of handwritten ASM (often beating it if you enable compiler optimisations). It is this that makes C the perfect language for any and all programming languages, where speed is key, and allfeatures need to be available are present.

3.3.3. How is C?

C is compiled to ASM, the main compilers available are clang, gcc and MSVC, I will be using gcc as it is generally standard in linux environments.

Many build systems are available for C, the main ones being cmake and gnu make. Both of them have the goal of putting the compiling process in one command. Cmake is cross platform (sorta windows doesn't work well but it does work).

3.3.4. Libraries

The libraries I will use are the following:

C stdlib

C unistd

C errno

Unix device files

Zippy strings

Zippy graphs

Zippy sorts

Addition libraries (may not be implemented):

Raylib

C sockets + Zippy sockets

3.3.5. Modularization

To make the project more manageable I will split it into many C files, this is to keep it from becoming impossible to edit code.

The file layout looks as follows:

```
Makefile
ads
  Makefile
  ast
    Makefile
    ast.c
    ast.h
    types.c
    types.h
  dict
    Makefile
    dict.c
    dict.h
    dicttest.c
  ll
    Makefile
    ll.c
    ll.h
    lltest.c
execution
  Makefile
  exec.c
  exec.h
  types.c
```

```
    types.h
  libs
    Makefile
    graphs
      graphs.zpy
    io
      io.zpy
    stdlib
      Makefile
      stdlib.c
      stdlib.zpy
    string
      string.zpy
    proto
      ast
        Makefile
        ast.c
        astg.c
        astg.h
  tokenizer
    Makefile
    parser.c
    parser.h
    tokenizer.c
    tokenizer.h
    types.c
    types.h
  zpy
    Makefile
    zpy.c
    zpycheck.c
  zpycheck
    Makefile
    errors.c
    errors.h
    zpycheck.c
    zpycheck.h
  zpypkg
    deps.zpy
    download.zpy
    zpypkg.zpy
```

As you can see this is split up over around 40 files and 16 folders, each file should not go over ~500 lines of code. This is to keep everything as easy to manage as possible.

This level of modularization in needed for the development of Zippy as without it, files will become a mess that can't be worked with.

All .c files will be compiled into .o files, then the .o files can be linked with the final zpy.c to generate the final executable.

3.3.5.1. Build system

The entire project is being build with GNU make files, each folder that builds something will have its own makefile. This will mean the entire project can be compiled with a single make in the root folder of the project.

Example of make:

make -j2

This will build all files specified by 'Makefile' with 2 threads.

The project should be build with gcc, and ld. It should be build with the -O3 build flag to ensure the program runs as fast as possible. -O3 forces the compiler to build with optimizations.

When the project is finished, I will try compiling with clang and tcc, to compare performance.

3.4. Time table

The first step is to tackle the interpreter, so the zpy.c file needs to be finished. The tokenizer, execution, and libs folders need to be finished, after this point you should be able to execute Zippy code however not syntax check it or get error handling.

The next step is zpycheck, the syntax and error handler, this should be ran before code is shipped to the user. It can reuse a lot of code from the tokenizer and execution steps.

Finally I need to make zpypkg, this should be easy as most of it can be written in Zippy, and a few bits can be written in bash. It should be a good test to how Zippy can be written.

If time allows it is at this point that I will write a Raylib library and a unix/C sockets library.

4. Technical Solution

5. Testing

6. Evaluation