NEA cover page

Lucas standen

7949

ABSTRACT

1. Analysis

1.1. 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.

1.2. 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.

1.3. 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.

1.3.1. 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.

1.3.2. Client 2, a technical user, but not a programmer

some stuff about this person.

1.3.3. Client 3, a normie

some stuff about how the normie finds the completed project.

1.3.4. 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.

1.4. Examples of similar projects

As Zippy will be interpreted, I should compare it to other such languages; trying to compare it to C++/rust/go, isn't helpful as they are so very different in the way they function.

Zippy is by far not the first language, and I'm only one person, so I can't expect to beat others in everything.

Below are a few languages that zippy should be compared to throught developemen:

1.4.1. 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 programing very easy and it should try and take notes from its libaries as they are mostly well writen, and well documented.

1.4.2. 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 anymore 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.

1.4.3. 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 minimalisum, 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.

1.5. 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.

1.5.1. Questionnaire 1 for Amy C

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

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

Speed, readability, debugging ease and disk space efficiency.

1.5.1.2. 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.

1.5.1.3. 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".

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

Lightweight command line tools and web back ends.

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

No.

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

I like a good balance between the two.

1.5.1.7. 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).

1.5.1.8. 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.

1.5.1.9. 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.

1.5.1.10. 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.

1.5.2. 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.

1.5.3. 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.

1.6. Abstract data structures and there implementations

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.

1.6.1. 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.

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.

1.6.2. 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; // the data of the dict

    int id; // the id of the dict

} 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

1.7. Prototyping hard features

1.7.1. 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))))

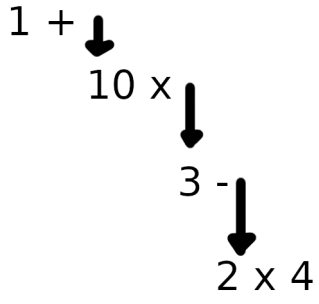
We know that this is equal to -49

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:



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

1.7.2. 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 litteral values to be surrounded by [] so it knows its not another expression to evaluate.

```
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;
```

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 expres-

sion would be in the "right" vairable. 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 litteral values. Then I can execute that node and work my way up the tree.

The execution code can be seen here.

```
int exec(ast_node *exp){

	if (exp->left != NULL)

		exp->realLeft = exec(exp->left);

	if (exp->right != NULL)

		exp->realRight = exec(exp->right);



	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;

}
```

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

2. Design

3. Technical Solution

4. Testing

5. Evaluation