Dope - An Interpreted Programming Language

Omkar Prabhune
Department of Computer Science
Vishwakarma Institute of Technology
Pune, Maharashtra, India
omkar.prabhune19@vit.edu

***Abstract* — Dope is a dynamically typed, automatically managed programming language with an Object Oriented Paradigm as well as an Imperative one. It can be run either on a text file or through using the REPL (Read, Execute, Print, Loop) Shell. The interpreter itself runs using a recursive tree-walk approach.**

Keywords* — *Programming, Languages, Dope, Object Oriented Programming, Interpreter, Dynamic Allocation.

I. Introduction

Dope is a dynamically typed, automatically managed programming language with an Object Oriented Paradigm as well as an Imperative one. It can be run either on a text file or through using the REPL (Read, Execute, Print, Loop) Shell. Implemented in Java, it operates on the Java Virtual Machine (JVM) and also uses the inbuilt Garbage Collector built into the VM.

The interpreter itself runs using a recursive tree-walk approach.

II. Features

Dope is dynamically typed. Variables can store values of any type, and a single variable can even store values of different types at different times.

If you try to perform an operation on values of the wrong type — Say, dividing a number by a string, then the error is detected and reported at runtime.

High-level languages exist to eliminate error-prone, low-level drudgery, and what could be more tedious than manually managing the allocation and freeing of storage? So, Dope will be using a custom Garbage Collector built and implemented in Java.

Apart from the rest (statements, control flow and functions), Dope is also an object oriented language implemented with prototypes , allowing the use of classes and objects with basic OOP concepts.

III. Methodology

Dope is implemented through a series of layers, with the input being passed to the interpreter either through using the REPL (Read, Execute, Print, Loop) Shell or by passing a .dope file to the dope command

A. Command Line Interface

The Command Line Interface (CLI) can be called using the dope command with usage: `"dope <file-name>"` . If called without a filename, It automatically opens the REPL Shell which can be terminated with the exit() command. The input at this point is still just a string. After this, it gets passed to the parser

B. Parser

The main job of the parser is to split the source code string into a list of tokens where a token can be represented as:

```
class Token {
   TokenType type;
   String lexeme;
   Object literal;
   int line;
}
```

Once the code is translated to Tokens, it is passed to the Resolver.

C. Resolver

The resolver is used for static analysis of the code, checking if variables exist in the environment, etc.

D. Interpreter

The interpreter is the real meat of the program. It is where we've defined statements in dope as statements in Java. It accepts a list of Tokens from the Parser through the Resolver and actually executes them.

E. Standard Functions

Apart from statements, declarations and all, Dope has two (with potential for more) standard functions built in. They are print() which prints a statement to the terminal and clock() which returns the current system time.

IV. Tech Stack

Dope has been implemented in Java using basic inbuilt data structures such as HashMaps and Lists. Basic Input/Output is done using the System.In and .Out functionalities. Apart from these the language is also implemented with RuntimeExceptions.

V. Syntax Grammar

The syntactic grammar is used to parse the linear sequence of tokens into the nested syntax tree structure. It starts with the first rule that matches an entire Dope program (or a single REPL entry).

```
program        → declaration* EOF ;
```

A. Declarations

A program is a series of declarations, which are the statements that bind new identifiers or any of the other statement types.

```
declaration    → classDecl
               | funDecl
               | varDecl
               | statement ;

classDecl      → "class" IDENTIFIER ( "<" IDENTIFIER )?
                 "{" function* "}" ;
funDecl        → "fun" function ;
varDecl        → "var" IDENTIFIER ( "=" expression )?
                 ";" ;
```

B. Statements

The remaining statement rules produce side effects, but do not introduce bindings.

```
statement      → exprStmt
               | forStmt
               | ifStmt
               | printStmt
               | returnStmt
               | whileStmt
               | block ;

exprStmt       → expression ";" ;
forStmt        → "for" "(" ( varDecl | exprStmt | ";" )
                 expression? ";" expression? ")"
                 statement ;
ifStmt         → "if" "(" expression ")" statement
                 ( "else" statement )? ;
printStmt      → "print" expression ";" ;
returnStmt     → "return" expression? ";" ;
whileStmt      → "while" "(" expression ")" statement ;
block          → "{" declaration* "}" ;
```

C. Expressions

Expressions produce values. Dope has a number of unary and binary operators with different levels of precedence. Some grammars for languages do not directly encode the precedence relationships and specify that elsewhere. Here, we use a separate rule for each precedence level to make it explicit.

```
expression     → assignment ;

assignment     → ( call "." )? IDENTIFIER "="
                   assignment | logic_or ;

logic_or       → logic_and ( "or" logic_and )* ;

logic_and      → equality ( "and" equality )* ;
equality       → comparison ( ( "!=" | "==" )
                 comparison )* ;
comparison     → term ( ( ">" | ">=" | "<" | "<=" )
                 term )* ;
term           → factor ( ( "-" | "+" ) factor )* ;
factor         → unary ( ( "/" | "*" ) unary )* ;

unary          → ( "!" | "-" ) unary | call ;
call           → primary ( "(" arguments? ")"
               | "." IDENTIFIER )* ;
primary        → "true" | "false" | "nil" | "this"
               | NUMBER | STRING | IDENTIFIER
               | "("  expression ")"
               | "super" "." IDENTIFIER ;
```

D. Utilities

In order to keep the above rules a little cleaner, some of the grammar is split out into a few reused helper rules.

```
function       → IDENTIFIER "(" parameters? ")" block;
parameters     → IDENTIFIER ( "," IDENTIFIER )* ;
arguments      → expression ( "," expression )* ;
```

E. Lexical Grammar

The lexical grammar is used by the scanner to group characters into tokens. Where the syntax is context free, the lexical grammar is regular—note that there are no recursive rules.

```
NUMBER         → DIGIT+ ( "." DIGIT+ )? ;
STRING         → "\"" <any char except "\"">* "\"" ;
IDENTIFIER     → ALPHA ( ALPHA | DIGIT )* ;
ALPHA          → "a" ... "z" | "A" ... "Z" | "_" ;
DIGIT          → "0" ... "9" ;
```

VI. Results

The language works as intended, implementing blocks, scopes and variable environments without logical or semantic contradiction. It does not suffer any major time flaws apart from the natural ones expected of a dynamic language.

It's only flaw could be said to be its lacking standard library which would have been impossible to implement in the given time frame by one student anyways.

VII. Future Scope

The language in its current state is fairly rudimentary, with only basic imperative programming and only the barest bones of an Object Oriented paradigm. The current standard library is also basically non-existent, having only two functions - print() and clock().

The first things to implement would be a wider array of standard library functions, a way to get input would certainly be nice. Post that, an increased amount of object orientation (Inheritance, Interfaces, etc.) would also be helpful.

Acknowledgment

I express my profound gratitude and deep regards to my college, Vishwakarma Institute of Technology's faculty for their guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by him fare a crucial part in the completion of this paper.

References

[1] David F. Bacon, Perry Cheng, V.T. Rajan, "A Unified Theory of Garbage Collection", IBM Research

[2] Crafting Interpreters, http://www.craftinginterpreters.com/

[3] William Wold, FreeCodeCamp, "The programming Language Pipeline";https://www.freecodecamp.org/news/the-programming-language-pipeline-91d3f449c919/