The Creation of a Programming Language

Florian Donnelly

Autumn 2022

Abstract

Contents

Introduction

Relevance of this topic

Programming languages are in some way the very essence of the modern, digital world. Whether indirectly or directly, they define what connect, entertain, employ and often also upset today's people. Programming languages create an interface between the human mind and the computers hardware. But there is no such thing as the perfect programming language or programming ecosystem.

The Goals

The main Goal of this work is the process, to learn about various technologies and their inner workings, focusing on programming languages. And what better way is there to learn about programming languages than to create one yourself?

Theory

A brief History of Programming Languages

Implementation

Used software

Since the interpreter was going to be written in C, standard tools like the GNU Compiler Collection, gcc for short, as well as the GNU Make build-system were used.

GNU Make takes a so-called Makefile and interprets it. In a Makefile one can define build targets that depend on files or other build targets and GNU make will automatically build the requested target.

As code editor, Microsoft's Visual Studio Code was used, as it provides good C syntax highlighting and completion and it has git and GitHub integration.

Procedure

Building the Project

At the heart of building the interpreter from it's C source-code stand gcc and make, the compiler and the build system.

See the Makefile used by the project:

```
# specify destination files
TARGET := lang
TEST_INPUT = test.txt

# general compiler flags
CC := gcc
LIBS := -lm
CFLAGS := -Wall -pg -Og -g -Wno-switch -Wno-return-type
```

```
# apply optimizations for the release target
release: CFLAGS := -Wall -O3

# activate debug output for test release
test: CFLAGS += -DDEBUG

# define recursive wildcard function
rwildcard=$(wildcard $1$2) $(foreach d,$(wildcard $1*),$(call rwildcard,$d/,$2))

# search for .c files in ./src
SRCS := $(call rwildcard,src/,*.c)
OBJS := $(SRCS:%.c=%.o)

.PHONY: all clean

all: mapop $(OBJS)
	$(CC) $(CFLAGS) $(LIBS) $(OBJS) -o $(TARGET)

release: all

test: all
	clear
	$(file >> $(TEST_INPUT))
	./$(TARGET) $(TEST_INPUT)

%.o: %.c
	$(CC) -c $(CFLAGS) $< -o $@

paper: paper.org
	pandoc -f org -t pdf paper.org -o paper.pdf

# target to generate mapop.c
# requires gperf to be installed
mapop: mapop.gperf
	gperf --output-file=./src//interpreter/mappings/mapop.c ./mapop.gperf

clean:
	rm -f -- $(TARGET) $(TARGET).exe gmon.out $(call rwildcard,src/,*.o)
```

At the top, several variables define compiler flags and the destination path where the interpreter executable will be created. The rwildcard function is defined, so that the source directory can be recursively searched for .c files. Then several build targets follow, starting with the default target 'all'. This target depends on the mapop target as well as all object files. Object files end in .o and are created from their respective source, or .c, files by gcc given the compiler flags set beforehand. The all target then instructs gcc to assemble all object files into a single executable file. The test target is an extension of the all target, which after building the interpreter executable, then runs against the test.txt input file.

Compiler vs Interpreter vs JIT

Using stages

pratt parsing etc.

Results

Description of Product (objective)

Discussion

Analyzation of Product

#does it work, what doesn't, how to extend

Prospect

what would i've done different what it takes, example: rust

Conclusion

Index

Closing Words

Declaration of Autonomy

Appendix

https://jonathanabennett.github.io/blog/2019/05/29/writing-academic-papers-with-org-mode/ https://en.wikibooks.org/wiki/LaTeX/Document_Structure