Programming Language Theory

Tristan Miller (tjm3990@rit.edu)

March 28, 2023

1. Paradigms

Procedural

- Program is organized into *procedures*, data is passed between and manipulated by various procedures
- `goto`, `if` statements, `for` and `while` loops are used for control flow
- Mirrors the behavior of CPUs

Procedural example (C)

```c
int numbers[] = {5, 2, 6, 3, 4, 1};
int total = 0;
for(int i = 0; i < 6; i++) {
    total += numbers[i];
}
printf("The total is: %d\n", total);
```

Object-oriented

- Program is structured around *objects*, which contain data and code that acts on that data
- Objects are instances of classes, which describe their behavior and internal state
- Inheritance allows classes to be extended to add new capabilities
- Focus on *encapsulation* - separating the program into independent, self-contained parts

Object-oriented example (Ruby)

```
numbers = [5, 2, 6, 3, 4, 1]
total = numbers.sum
puts "The total is: " + total.to_s
```

Functional

- Program is structured around *functions*, small pieces of code which can be combined together
- Functions can be stored and manipulated much like ordinary data
- Emphasizes *immutability* and *purity* - functions don't mutate their arguments or access external state

Functional example (Haskell)

```haskell
sumList [] = 0
sumList (x:xs) = x + sumList xs

main = do
    let total = sumList [5, 2, 6, 3, 4, 1]
    putStrLn ("The total is: " ++ show total)
```

Array

- Program is structured around *arrays*, n-dimensional tables of numbers (vectors, matrices, etc.)
- Any operation that can be applied to scalars can also be applied to arrays

Array example (APL)

```
numbers ← 5 2 6 3 4 1
'The result is:', +/numbers
```

2. Parsing

What is a parsing?

- Code is easy to understand for humans but difficult for computers
- *Parser* - algorithm for converting source code to an abstract syntax tree (AST)
- Parser generators generate parsers automatically

Grammars

- A *grammer* describes the syntax of a language
- Often written in Backus-Naur form (BNF) or EBNF

EBNF example

```
expression := term
    | expression ("+" | "-") term
term := item | term ("*" | "/") item
item := number | "(" expression ")"
number := "-"? digit+
digit := "0" | "1" | "2" | "3" | "4"
    | "5" | "6" | "7" | "8" | "9"
```

Lexing (Tokenization)

- Convert the source code to a list of tokens
- *Token* - smallest indivisible component of a language: literals, keywords, identifiers, operators, etc.

Lexing example

```
let grass_touched = (420 + 69) * 0;
```

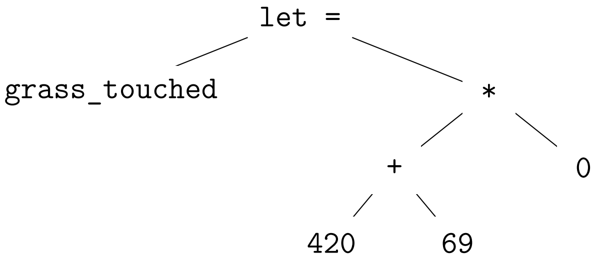
let	grass_touched	=	(	420	+	69	)	*	0	;

Parsing

- Convert the list of tokens to an AST representing the program

Parsing example

let	grass_touched	=	(	420	+	69	)	*	0	;



3. Execution

Compiled? Interpreted?

- i don’t like these words

AST Walking

- Recursively evaluate the AST directly
- Very easy to implement, but very slow (cache misses)

Immediate bytecode compilation

- Compile the AST into a more efficient bytecode form and execute that immediately
- Significant speed improvement over AST walking

Ahead-of-time bytecode compilation

- Separate compiler and bytecode interpreter (VM)
- Don't need to recompile the source every time it is run, cross-platform binaries (in theory)

Compilation to machine code

- Compile the source directly to native machine code
- 🚀 Blazingly fast 🚀, but very difficult (curse you x86) and not cross-platform

JIT compilation

- Compile bytecode to machine code on-the-fly
- VM profiles code to determine what to spend time compiling (ex. code run in tight loops)
- Best of both worlds: runs cross-platform but takes advantage of CPU architecture when possible
- Sometimes faster than ahead-of-time compilation: VM knows more about the code then the compiler.
- Witchcraft.

4. Memory Management

What is?

- How can programs get access to memory?
- How can programs give up access?

Static

- Persists throughout program lifetime (no need to acquire or free)
- Finite size that must be known at compile-time
- Ex: `static` in C and Rust

Stack-based

- Variable declarations allocate memory on the stack
- Freed automatically once out-of-scope
- Ex: Local variables in most languages

Manual

- Memory must be allocated and freed manually
- Ex: `malloc` and `free` in C

Garbage collection

- Memory allocated automatically when object is created
- Garbage collector looks for unused memory and frees it
- Ex: Python, Java, JavaScript

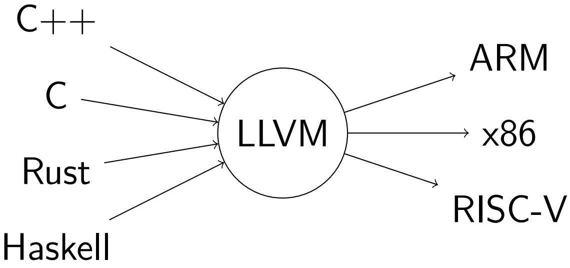
Reference counting

- Each object stores a reference count, freed once it reaches 0
- Ex: Python, `Rc<T>` in Rust

5. Miscellanea

LLVM

- Portable compiler toolchain: uses a common bytecode to compile languages to machine code



Lambda calculus

- System for expressing computations using only function creation and application
- Foundation for many functional languages

Lambda calculus

- x - variable
- $(\lambda x.M)$ - function definition
- $(M\ N)$ - function application

Lambda calculus

- $\lambda x.x$ - identity function
- $(\lambda x.x)a$ - identity function applied to a
- (simplifies to a)

Lambda calculus (cont'd)

LC	Javascript	Bird name
$I = \lambda x.x$	`I = x => x`	Idiot
$K = \lambda x.\lambda y.x$	`K = x => (y => x)`	Kestrel
$M = \lambda x.xx$	`M = x => x(x)`	Mockingbird

Lambda calculus (cont'd) (cont'd)

- Functions can behave like numbers (Church numerals)
- $0 = \lambda f.\lambda x.x$
- $1 = \lambda f.\lambda x.fx$
- $2 = \lambda f.\lambda x.f(f\ x)$
- $3 = \lambda f.\lambda x.f(f(f\ x))$
- $\mathrm{succ} = \lambda n.\lambda f.\lambda x.f(n\ f\ x)$

Esolangs

- Languages created as a proof of concept, as a joke, or to push the boundaries of programming languages
- Usually very limited; challenging (but often possible) to write useful programs

B****fuck

- Tape-based language: entire memory is one tape with cells storing integers
- Only eight instructions (`+ - > < [ ] . ,`)
- `++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.`

Fractran

- Programs are lists of fractions, input is a single number
- Repeatedly search the list for the first fraction f such that $n \cdot f$ is an integer and update n to the new value
- $\left(\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}\right)$

Javagony

- Java, but without (most) control flow
- `for`, `if`, `while`, `do while`, `switch`, `?:` are all illegal.
- How do we do things? Function calls and `try catch`.

6. The end!