Programming Languages in Software Engineering

Homework (Week 7, 2025; due on 16 February) Maximum points: 32 (5 bonus)

This week we are designing and implementing a bytecode-like programming language. We've already done much of the design work during the lecture. Now it's time to pin down that design and make an implementation.

Setup

Pick a buddy to work with. Together, make a fork of `https://github.com/jesyspa/cub-plsd-2024-homework` and create a subdirectory for your pair in the `bytecode/` directory. Make a `README.md` file and list your GitHub handles.

For the implementation part of this assignment, please use one of the following languages: C++, Haskell, Kotlin, Python, Rust. If you would *really* like to use something else, discuss it with me beforehand.

These exercises are written primarily with Python and Kotlin in mind. If the exercises ask you to define something that has no equivalent in your language (e.g. a class in Haskell), do whatever would be idiomatic in your language of choice.

In your `README.md`, include instructions on how to build and execute your code, including how to specify the bytecode file to run. If your code requires libraries, include instructions on how to install dependencies.

When you are done with the assignment, open a pull request to the original repository and assign `jesyspa` as a reviewer.

Design

In the lecture, we designed a bytecode machine and an instruction set for it.

Exercise 1 (12 pts): In your `README.md`, write a short description of the design. Make sure to address:

- What state does the machine have? (Registers, heap, etc.)
- What instructions are there?
- What arguments do these instructions take?
- What do the instructions do?

Implementation

Our implementation will take a sequence of instructions as a string, turn it into a more convenient representation, and then execute it by interpreting the instructions. We do this in three steps:

1. Lexing: convert the string to a sequence of tokens.
2. Parsing: build instructions from the token stream.
3. Interpretation: step through the instructions and execute them.

For a simple language like this, splitting lexing and parsing may seem like overkill. I recommend doing it this way so that you will be able to reuse the lexer in future homeworks. However, if you are familiar and with lexing and parsing and want to use a more advanced technique like parser generators or parser combinators, feel free to do so.

Lexing

A lexer takes a string as input, splits it into tokens, and attaches information to these tokens that will be useful for the parser. For example, a lexer for arithmetic could take the string `2*(a + b)` and

split it into the tokens `2`, `*`, `(`, `a`, `+`, `b`, `)`. As output, it could give the following "cleaned up" versions of these tokens:

Haskell	Kotlin, Rust	Python
`Literal 2`	`Literal(2)`	`('LITERAL', 2)`
`Operation "*"`	`Operation("*")`	`('OPERATION', '*')`
`OpenParenthesis`	`OpenParenthesis()`	`('OPEN_PAREN', None)`
`Variable "a"`	`Variable("a")`	`('VARIABLE', 'a')`
`Operation "+"`	`Operation("+")`	`('OPERATION', '+')`
`Variable "b"`	`Variable("b")`	`('VARIABLE', 'b')`
`CloseParenthesis`	`CloseParenthesis()`	`('CLOSE_PAREN', None)`

Note that while the Kotlin and Rust versions are similar, the implementation is different: in Rust you would use an `enum` with `Literal`, `Operation` and such as cases, while in Kotlin you would have a `sealed interface Token` with `Literal` and `Operation` as implementations.

While we could see the lexer as a function that takes a string and returns a list of tokens, providing an interface that lets you get the next token on demand will make the parser simpler.

Exercise 2 (1 pt): Define a `Token` type.

Exercise 3 (2 pts): Implement a `Lexer` class that takes a string at construction, and provides a `next` function to get the next token (if any).

Hints:
- Regex can be useful to determine what kind of token you're seeing.
- You can make a `Rule` type that maps regexes to handlers. A lexer just needs to try all rules.
- Don't forget to skip over whitespace.

Parsing

Now it's time to turn our tokens into a program. For most programming languages, parsing results in tree structures. However, since we have a very simple language we can get away with representing the program as just a list of instructions.

Exercise 4 (1 pt): Define an `Instruction` type.

Exercise 5 (2 pts): Implement a `parse` function that takes your lexer and returns a list of instructions.

Hint:
- Since you may often know what token to expect next, it may be worth adding a function to your lexer that gets a token and checks it is of a certain type.

Interpretation

We can now implement the bytecode machine itself, which we'll call the VM. We need to represent all the state that you described the machine having, and then specify for each instruction what effect it has on the state.

Exercise 6 (1 pt): Define a `VM` class that takes the program at construction, and that contains all the state that our bytecode machine has.

Exercise 7 (1 pt): Define a `Target` type with `get` and `set` methods. Implement a `resolve_target` function that takes an instruction target and returns a `Target` object.[1]

[1] The idea is that an instruction like `mov a b` should be possible to implement as something like `ra = resolve_target(a); rb = resolve_target(b); ra.set(rb.get());`

Exercise 8 (2 pts): Implement a `step` function in `VM` that interprets a single instruction and returns whether the machine has halted.

Exercise 9 (1 pt): Implement a `run` function in `VM` that runs the machine for as long as possible.

Hints:

- A little time spent writing debugging tools can save a lot of time debugging.
- Depending on the language, you may want a similar `resolve_condition` function for conditional jumps.

Next steps

Congratulations, you're done with the initial implementation! However, languages rarely stay still over time.

Exercise 10 (3 pts): Come up with a new feature for this language. Describe what it looks like and how it behaves in your `README.md`.

Exercise 11 (2 pts): Write two example programs that use your feature.

Exercise 12 (4 pts): Extend your lexer, parser, and VM to support this feature.

If you're done early…

Exercise 13 (Bonus, 1 pt): Find an ambiguity in our specification and clarify it in your README.

Exercise 14 (Bonus, 2 pts): Suppose we want to compile a language with functions to this bytecode. What would a function call look like? Write a specification for it.

Exercise 15 (Bonus, 2 pts): When writing in a low-level language we typically want to use labels rather than absolute jumps. Introduce a syntax for labels and implement a label resolution function in your interpreter.