COSC 4780 - Principles of Programming Languages - Spring 2022

Lecture 6: A Circuit Emulator

Finley McIlwaine

University of Wyoming

fmcilwai@uwyo.edu

February 8, 2022

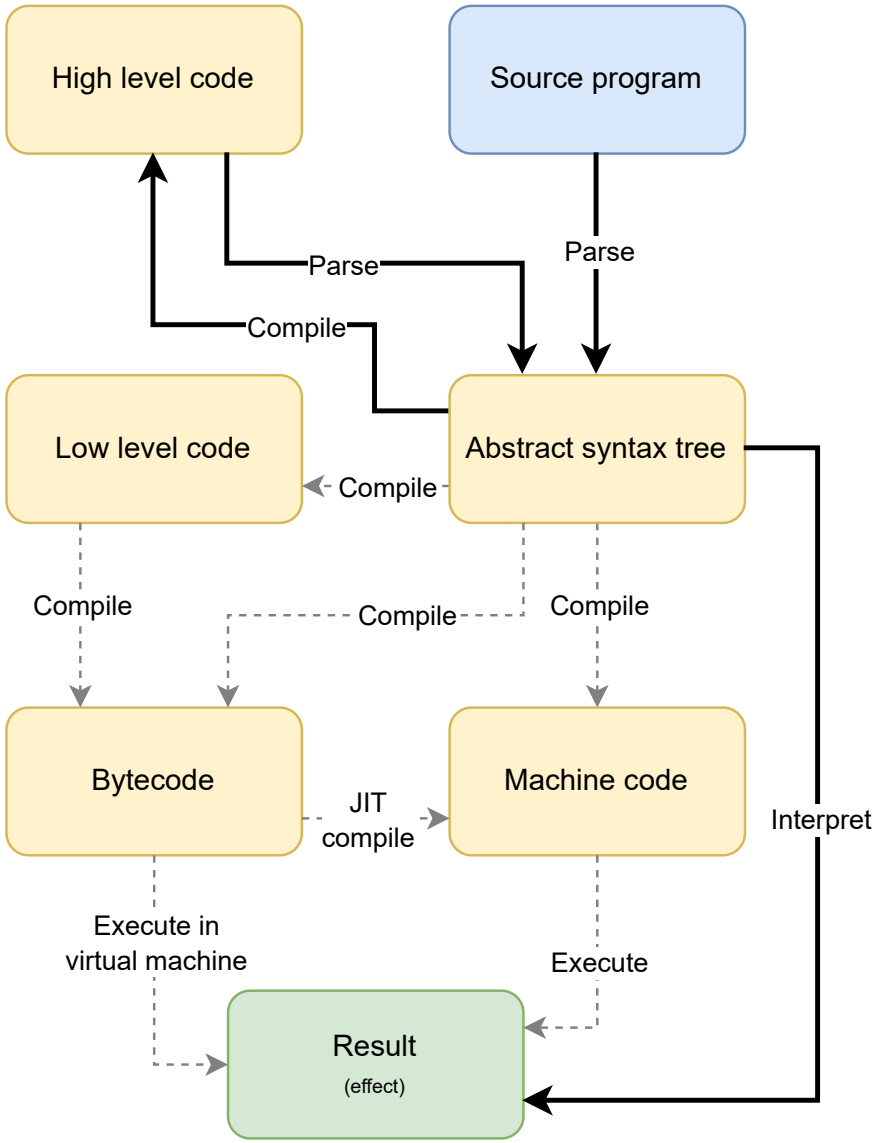
Review - Zoom Out

Many of our lectures so far have been case studies, implementing small compiler-related tools and features. I'm aware that these can all start to feel a little unmotivated and head-spinny, so let's zoom out really quick and revisit the goal of this course, where we're at, and where we're heading.

Review - Zoom Out

Here's the map I provided in the first lecture of the course. The black arrows represent everything we've covered so far, the dotted arrows represent things we haven't touched.

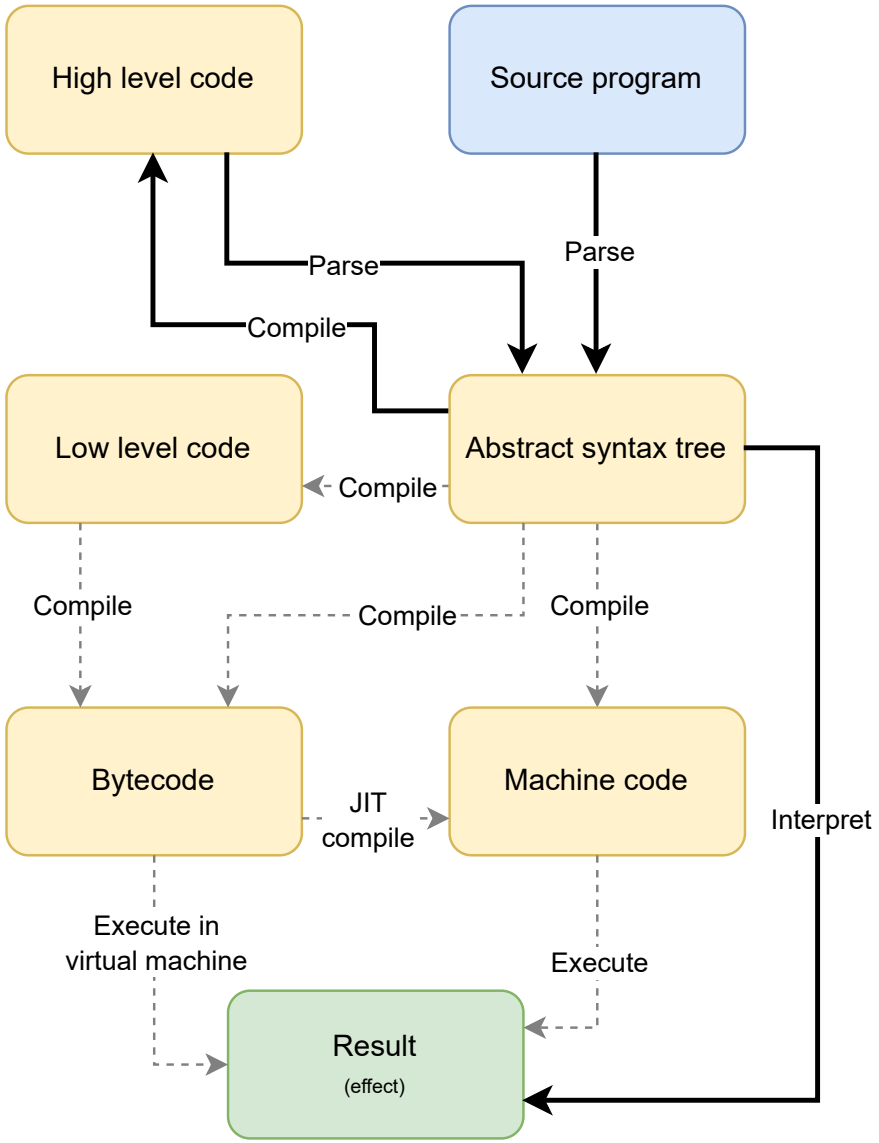
We won't be able to cover everything here in this class alone, but we'd like to cover as much as possible!



Review - Zoom Out

The assignments so far have been small explorations of these connections. Small interpreters, compilers.

After homework 2, we'll dive into a much larger project, where we'll implement a real language front to back, slowly building our way up to a few more dotted arrows on this diagram.



Review

Last time, we created a transpiler from a simple imperative language to JavaScript. The transpiler itself was very simple, because it didn't do any analysis of the source code, and didn't track any state.

The compiler in Homework 2 will not be so simple, and will require a sort of state tracking. So today, we're going to develop a *circuit emulator* that will show us what this sort of state tracking can look like.

Advent of Code

This problem is from Advent of Code, 2015, day 7. Advent of Code is a website that posts coding puzzles every year on the 25 days leading up to Christmas. The problems vary in difficulty, and a lot of them can be solved with the things we're learning in this class!

Problem Description

In this problem, we are given a textual representation of a circuit. Every line contains a different *gate* in the circuit, and each gate has inputs and outputs. Here's the example circuit we're given:

```
123 -> x
456 -> y
x AND y -> d
x OR y -> e
x LSHIFT 2 -> f
y RSHIFT 2 -> g
NOT x -> h
NOT y -> i
```

Wires are represented as variables identified by lowercase characters.

Problem Description

Examples of the different types of gates:

Gate	Description
`123 -> x`	Signal 123 is given to wire `x`.
`x -> y`	Signal from wire `x` is given to wire `y`.
`x AND y -> z`	Bitwise AND of wires `x` and `y` given to wire `z`.
`x OR y -> z`	Bitwise OR of wires `x` and `y` given to wire `z`.
`x LSHIFT 2 -> y`	Wire `x` is left-shifted by 2 and given to wire `y`.
`x RSHIFT 2 -> y`	Wire `x` is right-shifted by 2 and given to wire `y`.
`NOT x -> y`	Bitwise complement of wire `x` is given to wire `y`.

Problem Description

The goal is to "emulate" any given circuit so we can find out what the eventual signals on each of the wires is. For example, our example circuit will end up having the following signals on each wire:

```
123 -> x
456 -> y
x AND y -> d
x OR y -> e
x LSHIFT 2 -> f
y RSHIFT 2 -> g
NOT x -> h
NOT y -> i
```

```
d: 72
e: 507
f: 492
g: 114
h: 65412
i: 65079
x: 123
y: 456
```

Each wire carries a 16 bit unsigned signal value.

Solution

This is a compiler problem! We have a source language that we want to parse and transform into some sort of structure that we can emulate/interpret. It's similar to Homework 2 in that you will need to transform the regular expressions into some sort of structure that makes it easy to emit the target code.

Parsing

What is the grammar of this circuit language? What are our terminals, non-terminals? Think hierarchically about the structure of a circuit.

Parsing

What is the grammar of this circuit language? What are our terminals, non-terminals? Think hierarchically about the structure of a circuit.

Terminals appear to be our gate operators (`AND`,`OR`,etc.), wire identifiers, integer literal signals, and the `->` token.

Parsing

What is the grammar of this circuit language? What are our terminals, non-terminals? Think hierarchically about the structure of a circuit.

Terminals appear to be our gate operators (`AND`,`OR`,etc.), wire identifiers, integer literal signals, and the `->` token.

Circuit non-terminal will be our start symbol. A circuit is just a list of gates, and a gate has an input expression and a single output wire.

Parsing

Let's start putting this into LBNF. Starting with the start symbol (circuit) and gates:

```
Circuit. Circuit ::= [Gate] ;

Gate. Gate ::= InExp "->" Ident ;
separator Gate "\n" ;
```

So a circuit is a newline separated list of gates, and gates are input expressions followed by `->` and an identifier. What about `InExp`?

Parsing

Here's some examples of what we want `InExp` to parse:

```
123
x
x AND 1
1 AND x
x LSHIFT 2
NOT x
```

So integer literal signals and wires are both used as signals. Let's capture that in a non-terminal that we'll call `Signal`.

```
SigIdent. Signal ::= Ident ;
SigInt.   Signal ::= Integer ;
```

Parsing

Now the rules for `InExp` fall into place:

```
EConst.  InExp ::= Signal ;
EAnd.    InExp ::= Signal "AND" Signal ;
EOr.     InExp ::= Signal "OR" Signal ;
ELShift. InExp ::= Signal "LSHIFT" Integer ;
ERShift. InExp ::= Signal "RSHIFT" Integer ;
ENot.    InExp ::= "NOT" Signal ;
```

Note that we can restrict the right side of shift gates to be integers.

Parsing

Let's try to parse some example inputs using this grammar to make sure it works...

Emulation

We now have the circuit in an abstract syntax tree that we can use for emulation! First let's think about how this will work.

- We don't want to stop emulation unless the circuit is "saturated", i.e. signal cannot propagate any further.
- We can only propagate the signal at a given gate if all of it's inputs have signal already.
- We need to track the signal on each wire.

Emulation

So the general algorithm will go like this:

- Track wire signals in a map from wire ID to signal value. Initialize as empty map.
- Repeat until the signal map does not change:
  - For each gate in the circuit:
    - If an input wire is not in the signal map, do nothing, continue.
    - Calculate the output signal and map the output wire ID to that value in the signal map.

We will see how to implement this algorithm in Java and Haskell, and talk about the differences.

Haskell Emulation

In both languages, the emulator will be a function over the abstract syntax tree. It will be different than the functions we've seen before, though, because we are tracking and updating some state during emulation.

In Haskell, we will track the signals on each of the wires using a map from wire identifiers to signal values:

```
import AbsCircuit
import Data.Bits
import qualified Data.Map as M
import Data.Maybe
import Data.Word
import ParCircuit

type CircuitEnv = M.Map Ident Word16
```

Haskell Emulation

The normal boilerplate for the `main` function. If parsing succeeds, we pass the circuit to our emulation function `emulate` and print the result.

```
main :: IO ()
main = do
  source <- getContents
  case pCircuit (myLexer source) of
    Left s -> putStrLn $ "parse error! " ++ s
    Right circuit -> print $ emulate circuit
```

Haskell Emulation

The emulation function needs to do two things:

- "Run" each gate in the circuit and update the environment.
- Check if the resulting environment is equal to the result from the previous iteration, and return the environment if so.

Haskell Emulation

To achieve this, we use a helper function that takes the previous iteration's environment, runs each gate, and then compares the result.

```
emulate :: Circuit -> CircuitEnv
emulate circuit = emulate' M.empty circuit
  where
    emulate' :: CircuitEnv -> Circuit -> CircuitEnv
    emulate' env (Circuit gates) =
      let env' = foldr runGate env gates
       in if env' == env then env else emulate' env' circuit
```

We start by calling the helper function with an empty environment. The traversal of the gates is implemented as a fold with a function that runs gates and updates the environment.

Haskell Emulation

The function `runGate` takes a gate and an environment and checks if the output wire already has signal (i.e. we've already ran this gate on a previous "pass"). If not, it calculates the output signal (if the inputs have signal) and sets the output signal in the environment.

This is the most important/complex function of the emulator.

Haskell Emulation

Here's one way to implement it:

```
runGate :: Gate -> CircuitEnv -> CircuitEnv
runGate (Gate inExp ident) env =
  if ident `M.member` env
    then env
    else case inExp of
      EConst sig -> setSignal ident $ signalValue env sig
      EAnd sig1 sig2 -> setSignal ident $ sigAnd env sig1 sig2
      EOr sig1 sig2 -> setSignal ident $ sigOr env sig1 sig2
      ELShift sig n -> setSignal ident $ sigShift env sig n
      ERShift sig n -> setSignal ident $ sigShift env sig (negate n)
      ENot sig -> setSignal ident $ sigComplement env sig
  where
    setSignal ident signal = fromMaybe env $ M.insert ident <$> signal
    <*> pure env
```

Haskell Emulation

There's `Maybe`s involved, because if a wire does not have a signal in the current environment, then looking up that wire identifier in the environment will return `Nothing`.

The way I wrote the `runGate` function uses the `Maybe` applicative instance pretty heavily, just to keep the code from exploding in size. The underlying logic is quite simple, as we will see in the Java code.

Java Emulation

The key part of the Java code is an `Emulator` class that implements the visitor methods for all of the non-terminals in our grammar.

```
public class Emulator implements
        circuits.Absyn.Circuit.Visitor<Void, Void>,
        circuits.Absyn.EGate.Visitor<Void, Void>,
        circuits.Absyn.InExp.Visitor<Optional<Integer>,Void>,
        circuits.Absyn.Signal.Visitor<Optional<Integer>,Void> {
...
```

Notice the only visitors that actually return something are the visitors that calculate the signals. All other parameters are `Void`, whose only value is `null`.

Java Emulation

Now, state handling is much easier. We just track the circuit environment as a class-level variable of type `HashMap<String,Integer>`:

```
...
    private HashMap<String, Integer> circuitEnv;

    public Emulator() {
        this.circuitEnv = new HashMap<>();
    }

    public HashMap<String,Integer> getCircuitEnv() {
        return this.circuitEnv;
    }
...
```

Java Emulation

The visitor for the `Circuit` non-terminal handles the "run until saturated" logic:

```
...
    @Override
    public Void visit(Circ p, Void arg) {
        while (true) {
            HashMap<String,Integer> oldEnv = new HashMap<>(circuitEnv);
            for (EGate gate : p.listegate_) {
                gate.accept(this, null);
            }
            if (oldEnv.equals(circuitEnv)) break;
        }
        return null;
    }
...
```

Java Emulation

The visitor for the `Gate` has the same logic as `runGate` from the Haskell code:

```
...
    @Override
    public Void visit(Gate p, Void arg) {
        if (circuitEnv.containsKey(p.ident_)) return null;
        Optional<Integer> signal = p.inexp_.accept(this, null);
        signal.ifPresent(sig -> circuitEnv.put(p.ident_, sig));
        return null;
    }
...
```

Java Emulation

All of the `InExp` visit methods have similar logic:

```
...
    @Override
    public Optional<Integer> visit(EOr p, Void arg) {
        Optional<Integer> signal1 = p.signal_1.accept(this, null);
        if (signal1.isPresent()) {
            Optional<Integer> signal2 = p.signal_2.accept(this,null);
            if (signal2.isPresent()) {
                return Optional.of(signal1.get() | signal2.get());
            }
        }
        return Optional.empty();
    }
...
```

Java Emulation

Finally, the visitors for `Signal`s.

```
...
    @Override
    public Optional<Integer> visit(SigIdent p, Void arg) {
        return Optional.ofNullable(circuitEnv.get(p.ident_));
    }

    @Override
    public Optional<Integer> visit(SigInt p, Void arg) {
        return Optional.of(p.integer_);
    }
...
```

Java Emulation

Main takeaways from the Java code:

- State is kept "globally", at the class level, so we don't have to pass things around as much.
- Watch out for references and mutability.
- Optionals are a good way to avoid sketchy null checks, and make the behavior of different functions much more obvious.
- State can be kept in many places, and there's a lot of ways to abstract it. Choose what feels "best" to you!

Homework 2 Discussion

Let's talk about how this relates to homework 2...

The End