Programming Languages

Before we begin...

Welcome to the programming languages course! Just for the sake of better mutual understanding, let's formalize a legend that will be used in these handouts:

- Whatever text is written in a regular, black-on-white font is essential and shouldn't be skipped. This is usually something you should take your time with and read through carefully. If it's **bold** or *italic*, doubly so!
- If text appears in a darker grey box with a **"Sidenote"** caption, it's usually an interesting sidenote on the topic in question. These are usually there to tell you things you would come to realize for yourselves at some later point. These can also be cool tidbits to tell at nerdy parties.
- If text appears in a dashed, lighter grey box, with a caption like **"Don't drink too much Kool-aid"**, this is usually an example, counter-example, question or something to spend some time thinking about. We're interested in your opinions, if you'd give them.
- Code segments are given in a `monospaced font`, with explanations usually following or preceding them.
- Math expressions and grammars are in the standard *latex math* formatting.
- Anything else is a comment.

1 Introduction and Motivation

> The trouble with programmers is that you can never tell what a programmer is doing until its too late.
>
> — Seymour Cray, father of supercomputers

As computer scientists, we have to work with programming languages. Even though most programming languages seem more formalized than natural languages we know, nothing *important* actually changes between the two groups: both still have a way to write them correctly – called a **syntax**, and a way to understand what the actual meaning of the writing is, called **semantics**.

> ***Sidenote:*** You can see that even the simplest successful human communication has to have a syntax and a semantics. Looking at the previous section: the way we write things (the color of the boxes, style of the fonts, etc.) is the syntax of our handout "language", and what those mean for you is the handout semantics!

Because it's very easy to create bugs in the code we write, and more often than not painstakingly hard to find and correct them, we can see why learning *about* programming languages as a theoretical basis could be useful: if we could create better, more secure, robust and bug-free languages, anyone using them could probably reap the same benefits for their own various applications. Alongside operating systems and architecture, programming languages are the third pillar of computer science, after all. That's why we take great care to formalize our languages up to the smallest detail and prove that they do exactly what we want them to do.

> ***Don't drink too much Kool-Aid:*** *Even though the ANSI C standard (latest version from Dec 2011) has a whopping 683 pages, C still has many undefined behaviors, some triggered by even the simplest of code samples. Why do you think this is?*

Depending on what we're trying to achieve, programming languages can be **low-level** – in a very simple language, closer to what is native to the machine – or **high-level** – capable of transforming complex abstract ideas into machine instructions.

To illustrate how to formalize a syntax and semantics, it's easier to start on the low-level end of the spectrum and use a language we already know, albeit idealized and simplified for demonstrative purposes.

2 An Idealized, Simplified MIPS Assembly

> In theory, theory and practice are the same.
> In practice, they are not.
>
> — remark overheard at a CS conference

Starting off, let's limit ourselves to some of the more fundamental features of MIPS Assembly, and implement only the `addu`, `beq`, `bne`, `sltu`, `li` and `halt` instructions. We'll only address general-purpose registers in a register file (with a special **zero** register), to deviate as little as possible from our point of expressing semantics in a formalized way.

> ***Sidenote:*** As a refresher, here's what these instructions do:
>
> - **`addu`** adds the values of two specified registeres together, placing the result in a third specified register, without trapping overflows
> - **`beq`** jumps to a particular instruction if the values of two specified registers are equal
> - **`bne`** jumps to a particular instruction if the values of two specified registers are **not** equal
> - **`sltu`** compares the contents of two specified registers. If the first is less than the second, put 1 in a specified destination register, otherwise put 0
> - **`li`** loads a specified value into a specified register
> - **`halt`** is not actually a real MIPS instruction; this stops program execution

2.1 Formalizing the syntax

A formalization of the syntax behind the instructions for this small MIPS assembly language could be written like this:

$$n \in \mathbb{N} \qquad v \in \mathit{BinaryValue}$$

$$\begin{aligned}
t \in \mathit{TempRegisterId} &::= \mathbf{t}_n \\
r \in \mathit{RegisterId} &::= \textbf{zero} \mid t \\
a \in \mathit{Address} &::= n \\
i \in \mathit{Instruction} &::= \textbf{addu}\ r_1\ r_2\ r_3 \mid \textbf{beq}\ r_1\ r_2\ a \mid \textbf{bne}\ r_1\ r_2\ a \mid \textbf{sltu}\ r_1\ r_2\ r_3 \mid \textbf{li}\ r\ v \mid \textbf{halt}
\end{aligned}$$

We can imagine that a program is a sequence of instructions, and as such, we can construct syntactically correct programs with just this formalization. The **bold** words in our syntax are terminal symbols: they don't change to anything; everything else has to find its way to a terminal symbol. Because this process is strictly enabled through the rules of the syntax, we can't end up with an address where a register is expected, for example.

Colorless green ideas sleep furiously: *Pick a natural number m. For us, it will be* 3*. Your program will have that number of instructions (we'll separate them by a semicolon for clarity):*

$$i_1;\ i_2;\ i_3;$$

For every instruction, choose one option from the $i \in$ Instruction rule to replace it with. Let's say we choose i_1 to be **addu** $r_1\ r_2\ r_3$.

$$\textbf{addu}\ r_1\ r_2\ r_3;\ i_2;\ i_3;$$

We repeat this process for all non-terminal tokens, in any order: we can do registers r_1, r_2 and r_3 first, for example, from the $r \in RegisterId$ rule, which further propagates to either **zero** *or a $t \in TempRegisterId$.*

$$\textbf{addu zero}\ t_1\ t_3;\ i_2;\ i_3;$$

Doing the exact same thing for i_2 and i_3 brings us to a randomly-generated but syntactically correct program in our small assembly language, for example:

$$\textbf{addu zero}\ t_1\ t_3;\ \textbf{beq zero zero}\ 7;\ \textbf{halt};$$

Keywords: syntactically correct but semantically lacking! If you spend some time thinking about what this program means (which is what semantics is), you'll very soon see that it doesn't make much sense; very similar to a rather famous meaningless sentence invented by Noam Chomsky, used in the title of this note...

2.2 Formalizing the semantics

To talk about semantics, the meaning of a certain part of the program, we have to talk about state and state changes.

2.2.1 Statefulness

To formalize this, let us introduce *State*, as a tuple consisting of all the moving pieces available to us: registers, the instruction pointer that tells us where we are in our execution, and the memory model (which we won't even touch upon here, except to say that our instructions are written in it).

$$\sigma \in State = RegisterFile \times InstructionPointer \times Memory$$

In our semantics, all three of these have to be defined in a similar fashion: register files $f \in RegisterFile$ are maps from register identifiers to binary values (numbers in binary, basically):

$$f \in RegisterFile = TempRegisterId \to BinaryValue$$

The instruction pointer, even more simply, is just an address that we can reference:

$$ip \in InstructionPointer = Address$$

Finally, memory is a mapping from an address to an instruction:

$$mem \in Memory = Address \to Instruction$$

Sidenote: We use lots of mathy notation, don't we? Every state σ in the set of possible states *State* contains one register file $f \in RegisterFile$ holding all the registers, one instruction pointer $ip \in InstructionPointer$ and one memory mapping $mem \in Memory$ holding all the instructions in it.

One might wonder why we suddenly use "=" for semantics where we used "::=" in syntax. These two actually aren't even similar: one of them is just an equality sign; the other, "::=" is what is called a **production rule**, and means that the left side can produce the right side. Syntax rules can produce new things, while the semantic rules are just stating what things are: a state is a tuple of things, while an *Instruction* produces one of the concrete instructions, like **addu** or **halt** are.

As a heads-up: it might seem weird at first, but you can think of functions that have the same arguments and returning types as belonging to the set. So, for example, if we say that $plus \in BinaryValue \times BinaryValue \rightarrow BinaryValue$, what we are actually expressing is that a function called *plus* takes two *BinaryValue* arguments and returns a *BinaryValue*. In that, it would be similar to any other function that takes two numbers and returns one, like *mult* or *div*.

We use the cross-product sign "×" with the arguments because that's how one builds tuples in set theory, and we're mostly stealing the notation from there.

Now that we have statefulness, we can implement change. First, though, we need to write down some helper functions!

2.2.2 Helpers

We'll have a function called `update(f, r, v)`, which updates the register file f, at the register id r with the value v. What we get back from `update(f, r, v)` is a register file.

The semantics of this is as follows:

$\texttt{update} \in RegisterFile \times RegisterId \times BinaryValue \rightarrow RegisterFile$

$\texttt{update}(f, \textbf{zero}, v) = f$

$\texttt{update}(f, \mathbf{t}_n, v) = f + [\mathbf{t}_n \mapsto v]$

The expression used in the last line, $f + [\mathbf{t}_n \mapsto v]$ means "add the new mapping from $\mathbf{t}_n$ to v into the register file f". Next, we need a function to `fetch` a value from a register file f at a specific register id.

$\texttt{fetch} \in RegisterFile \times RegisterId \rightarrow BinaryValue$

$\texttt{fetch}(f, \textbf{zero}) = 0$

$\texttt{fetch}(f, \mathbf{t}_n) = f(\mathbf{t}_n)$

The one remaining thing we have to do is somehow define what our initial state will be. To get to this point, we will have to initialize our memory to hold the vector of instructions. For this, we use `initMemory`:

$\texttt{initMemory} \in \overrightarrow{Instruction} \times \mathbb{N} \times Memory \rightarrow Memory$

$\texttt{initMemory}([inst; \overrightarrow{rest}], n, mem) = \texttt{initMemory}(\overrightarrow{rest}, n + 1, mem + [n \mapsto inst])$

$\texttt{initMemory}(\varnothing, n, mem) = mem$

This might be the most complicated one, but we show off an important way of doing things: recursion!

Sidenote: For the faint of heart, here's an explanation of the above `initMemory` function: the vector of instructions is written as $[inst; \overrightarrow{rest}]$, *inst* being the first (or head) element, and $\overrightarrow{rest}$ being, well, the rest of the vector. This way of writing a sequence is usually called deconstruction.

In case that we have something like this, the function puts $[n \mapsto inst]$ into *mem*, and then recursively calls (*initMemory*), which does the same for the following instruction (which is now the first one in $\overrightarrow{rest}$), and so on until the rest is empty. Then, we just return *mem* and are done: our memory now contains all the memory mappings for all the instructions!

So far so good, we have our memory set, so the state shouldn't be hard. The final helper function is called `initState`:

$\texttt{initState} \in \overrightarrow{Instruction} \rightarrow State$

$\texttt{initState}(\overrightarrow{instructions}) = (\forall \mathbf{t}_n.\ [\mathbf{t}_n \mapsto 0], 0, \texttt{initMemory}(instructions, 0, []))$

You can see we use the expression $\forall \mathbf{t}_n.\ [\mathbf{t}_n \mapsto 0]$, which is only used because we didn't specify how many registers $\mathbf{t}_n$ there are in a register file, so we're saying "how ever many there are, for all of them, set their values to 0", in math speak.

2.2.3 Transitions

Now, finally, we can speak of semantics as a change of state. Every instruction is formalized as a function that takes a state (deconstructed into a simple tuple (f, ip, mem)), and returns a similar state. Let's give an example with **addu**:

$$\begin{aligned}&\langle \textbf{addu}\ r_1\ r_2\ r_3\rangle\ (f,\ ip,\ mem) :=\\ &\quad v = \texttt{fetch}(f,\ r_2) + \texttt{fetch}(f,\ r_3)\\ &\quad f' = \texttt{update}(f,\ r_1,\ v)\\ &\quad (f',\ ip+1,\ mem)\end{aligned}$$

Basically, it says exactly what one can expect: if **addu** $r_1\ r_2\ r_3$ is executed over a state $(f,\ ip,\ mem)$, it computes a sum of values fetched from r_2 and r_3, and updates the register r_1 with this value, moving the instruction pointer forward by one and not changing memory at all. We say that it returns this new state for whatever the next instruction might be, to continue processing.

Seeing how our simple MIPS assembly doesn't change memory with any instructions, we can skip writing that part down and write the whole thing in the form of a table, for ease of reading.

#	$mem(ip)$	Premises	f_{new}	ip_{new}
1	**addu** $r_1\ r_2\ r_3$	$v = \texttt{fetch}(f, r_2)\ +\ \texttt{fetch}(f, r_3)$	$\texttt{update}(f, r_1, v)$	$ip+1$
2	**beq** $r_1\ r_2\ a$	$\texttt{fetch}(f, r_1) = \texttt{fetch}(f, r_2)$	f	a
3	**beq** $r_1\ r_2\ a$	$\texttt{fetch}(f, r_1) \neq \texttt{fetch}(f, r_2)$	f	$ip+1$
4	**bne** $r_1\ r_2\ a$	$\texttt{fetch}(f, r_1) = \texttt{fetch}(f, r_2)$	f	$ip+1$
5	**bne** $r_1\ r_2\ a$	$\texttt{fetch}(f, r_1) \neq \texttt{fetch}(f, r_2)$	f	a
6	**sltu** $r_1\ r_2\ r_3$	$\texttt{fetch}(f, r_2)\ <\ \texttt{fetch}(f, r_3)$	$\texttt{update}(f, r_1, 1)$	$ip+1$
7	**sltu** $r_1\ r_2\ r_3$	$\texttt{fetch}(f, r_2)\ \geq\ \texttt{fetch}(f, r_3)$	$\texttt{update}(f, r_1, 0)$	$ip+1$
8	**li** $r\ v$		$\texttt{update}(f, r, v)$	$ip+1$
9	**halt**		f	ip

This style of representing program semantics is called **small-step operational semantics**. We call it "small-step" because execution proceeds one small step at a time, via multiple applications of a transition function. This is in contrast to what is called "big-step", which effectively goes from the initial state to a final state in one big step. We also say "operational", because it talks about the operational aspects of running a program, instead of abstractions that represent them (known as **denotational semantics**).

Sidenote: A quick overview of how to read this table would be the following.

- $mem(ip)$ is saying "if the current instruction pointer is pointing to this instruction"
- "and these **Premises** are true..."
- "the resulting state's register file f_{new} is given in this column,"
- "and the resulting state's instruction pointer ip_{new} is in this column"

We have the rules numbered in the first column, as a potentially easier way of referring to them. The operation + and the relational operators < and ≥ aren't defined in the semantics and are just implied to be addition over binary values and less-than and greater-or-equals-than relations over binary values, respectively.

The only thing that we didn't have already is the **Premises** row, which is used to make different assumptions about the instructions; for example, whether two values are equal or not. If the result of the premises is **false**, that rule isn't executed and the appropriate next one is selected.

Don't drink too much Kool-Aid: *What happens if our language semantics doesn't cover all of it's ground and we can end up in a situation where there is **no** appropriate next one? Does our MIPS assembly have any such holes? Also, do you see why we used* **addu** *instead of* **add***?*

An technical note about the **halt** instruction: notice how we don't increase or change the instruction pointer in that one, technically looping forever in place instead of actually terminating. We call this state being **stuck**.

Congratulations, you now know how to construct a really limited, low-level language! If you wish to expand it, you now have the know-how. This might not seem like much, but let's make a quick overview of what we did, and what we'll do next.

- We learned how to read syntactic production rules, useful for explaining what expressions represent valid programs in our language
- We learned that the semantics of a language gives meaning to syntactic expressions, defined by a state and a transition function that turns one state into another
- We understand how it works: by checking the current state, we get which instruction to execute (via instruction pointer); next, that instruction is applied to the current state, transitioning to a new state. This state is used as current for the next instruction, and we repeat this process until we reach a **halt** instruction: it changes nothing and returns the same state, thus getting stuck and halting execution
- This kind of semantics we called **small-step** and **operational**, because it makes small steps and tells us how the language operates, instead of what it represents

All of this might not seem particularly useful yet, as we are still talking about designing the syntax and semantics on paper, but as we'll see soon, this design is exactly what we need to have to be able to write a working implementation of a language. With that said, the assignments relevant to this handout in particular are:

- Assignment 1: Introduction to Scala
- Assignment 2: Implementing a SimpleScala small-step operational semantics

We use Scala for this course as it enables us to write code that very closely resembles the mathematical structures we are dealing with. Once you get to feel comfortable with Scala, we'll introduce SimpleScala, a high-level language based on Scala that you will implement from first principles described in this handout.

Practically, yes, in two weeks, you will have made a programming language you can actually write in. Thank you for your time!