This unit is about some more aspects of variables, beyond their scope, that vary between different programming languages. We will see some different options for how things work, including perhaps some surprising revelations about languages you know and love. Remember to always ask yourself, why do you think the language designers made the choices they did? There are probably good reasons, even if you disagree or find it annoying!

1 Assignments

Every language that has variables must have some way of associating values with those variables. This is called an assignment, and we have seen some varying syntaxes for it:

```
new x := 10;
x := 5;
(define x 10)
(set! x 5)

int x = 10;
x = 5;
```

This might seem like a completely obvious thing, and you might be wondering what more you could possibly learn about it. Well, you might be surprised! In different languages, what an assignment does can have different meanings (having to do with how things are stored and when things are copied). Furthermore, what kinds of things are allowed on either side of an assignment varies widely between languages. Why this variation, and what kinds of things are allowed?

1.1 Model of variables

Consider the following C++ program:

```
int x = 5;
int y = x;
++x;
```

After executing this program, of course x will equal 6, but y will still equal 5! This is because C++ uses the *value model of variables* by default, which means that the values are copied in memory every time there is an assignment. So the second line (assigning x to y) causes there to be two “5”s stored in memory, only one of which is changed in the third line.

The same exact thing will happen in Java if we run the same program. But what about this Java code snippet?

```
ArrayList<String> a = new ArrayList<String>();
ArrayList<String> b = a;
a.add("boo");
```
After this, a is clearly an ArrayList that contains exactly one String, 'boo'. But what about b? Does it contain 'boo' too, or is it empty? The answer is that, in Java, a and b will share the same memory here, and so they both contain 'boo'. There is really only a single ArrayList in memory in this program, and a and b are just two references to it. This is because, for Objects like ArrayList, Java follows the reference model of variables, which means that assignments don’t cause the values to get copied, but just makes the new name point to the same thing as the old name.

We can make C++ follow the reference model too, if we want, by using reference variables. Here’s the example above, with reference variables:

```
int x = 5;
int &y = x;
++x;
```

After executing this code, both x and y will equal 6, because they’re both just references to the same variable in memory!

### 1.2 l-values and r-values

Every assignment statement has two parts: a left-hand side and a right-hand side. Any kind of code that could go on the right hand side of an assignment is called an r-value. Similarly, anything that could appear on the left is called - you got it - an l-value.

R-values are pretty straightforward: in most languages, any expression (i.e., any code that returns a value) can be an r-value.

The more interesting question is about l-values. What can be an l-value? It depends on the language. Here are some options:

- **Names.** This will be possible in every language. That’s what assignments are for! In some languages, like Scheme, a name is the only thing that can be on the left-hand side.

- **Array references.** In C, C++, Java, and the like, we can write code like `A[5] = 10;`. This is a little bit interesting, because it’s not the same as evaluating `A[5]` as an expression (which would return an int presumably), and then doing the assignment. The compiler or interpreter has to do something different to make this kind of thing work!

- **Function returns.** This one is not possible in Java, but it is in C++. In fact, you probably did something like this in one of your labs:

```
stack<int> S;
S.push(10);
S.top() = 11;
```

See that last line? We are assigning to a function call! The reason this is possible in C++ is that the type that that function returns is actually a reference (to the top of the stack). In most languages, such things are simply not allowed.

We saw more examples than this in class, and there are more kinds of things still in C++. I encourage you to experiment, research, and see if you can find any more examples of these in the language.

Looking at how assignments can be used in the context of a program provides a nice way to talk about l-values in r-values. In Scheme, a definition such as

```
(define x 5)
```

is purely a statement, not an expression of any kind. So it can’t be used as an r-value or an l-value. The same is true in our SPL language; an assignment does not return a value.

In many languages, an assignment is an expression and therefore can be used as an r-value. Typically the returned value is whatever value just got assigned. In Java, for example, we can write:
\[ x = (y = 0); \]

as a shorthand way of setting \( y \) to zero, and then setting \( x \) to zero (assuming they have already been declared). Actually, the parentheses in the code above are unnecessary, as the assignment operator = is right-associative. Remember what that means?

It is relatively rare, but not unheard-of, for assignments to be valid l-values in a language. Although the following is not valid in Java or Python, we can do this in C++:

\[ (x = 1) = 2; \]

(again, assuming \( x \) has already been declared). What this does is first assign \( x \) to the value 1, then re-assign the same variable \( x \) to 2. Why would you ever want to do this? I can’t think of a good reason, really. But it’s valid in the language because C++ returns a reference from an assignment statement, and allows any reference to be used as an l-value in an assignment. You might not want to write code like this, but if you are writing a C++ compile, you had better be able to handle it!

1.3 Constants, immutables, clones

There are two distinct but unrelated concepts that I want you to understand:

- A **constant** is a name in a program whose bound value cannot be changed. For example, if we write

  \begin{verbatim}
  const int x = 3;
  \end{verbatim}

  in a C program, then \( x \) can never be reassigned to anything else. Constants are useful for documentation and assurance in a program, communicating and enforcing the programmer’s intent that a name is not reassigned.

  Beyond this, constants also allow for certain compiler optimizations, since the value of a constant could be substituted directly in the code wherever it is later accessed.

  Advanced compilers such as gcc will even do simple arithmetic with constant expressions at compile time and insert the results directly into the compiled machine code. Awesome!

- An **immutable** is an object in a program whose state cannot be changed. String objects in Java are a good example of this. Look in the String class documentation and you will discover that there are no methods which change the underlying object, only ones to return new String objects. (If you go to that page, you will also discover the variations in language choice among various sources. The Javadoc says that Strings are “constant”, although by our definition they are not constant, since a name that refers to a String can be reassigned freely; we say Strings are immutable since the value of the object itself cannot change.)

  Immutable objects can be used in contexts where mutable ones couldn’t. For example, Python makes extensive use of lists, which can also be modified in-place and are therefore **not** immutable. But this capability of lists also means they can’t be used as keys in a hashtable. The reason why not is that, since lists are mutable, the key could be modified after insertion into the hash table, leaving it potentially in the wrong spot (assuming the modification also changes the hash value). For this reason, Python provides tuples, which are just like lists except that they are immutable. Tuples can be used in places where lists cannot, such as keys in a hash table.

  Immutable objects also allow for some compiler optimizations. For example, the Java virtual machine can safely store all String objects with the same value in the same physical memory location. In other words, if the string "hello" appears in 10 different places in your program, it is only stored once, with a bunch of references to it. Since Strings are immutable, there is no danger in doing this, and it can make the code much leaner.

We started this unit by talking about the reference and value models of variables. Immutables provide a mechanism for treating a reference model as if it is a value model. Again, in the example of Strings in Java, we know that since Strings are objects, Java uses a reference model of variables in assignments of Strings. So for instance, the code

\begin{verbatim}
String a = "I only live once";
String b = a;
\end{verbatim}
results in b and a pointing to the same memory location. But since Strings are immutable in Java, the programmer
 can behave as if a and b each had their own copy of the string, since there won’t be any difference either way.

Clones provide a mechanism for using a value model of variables against the default of the language when immutables
are not an option. Really, this is just a fancy name for copying; since the value model of variables means every
assignment makes a copy, we can simulate this by manually copying something when it gets assigned. In Java,
the Cloneable interface indicates that an object supports the clone method to make a copy of itself. You can see
examples of this in the slides for this unit or in any number of places online.

2 Types

Besides assignments, the other thing that (almost) always comes along with having variables in a programming
language is types. We all know informally that a type is something like int or String that indicates what kind of
thing some value is in a program. More formally, the type of something is the information about how it can be used
and combined with other values in a program. As with assignments, there are a plethora of options when it comes
to types, and we will explore a few of them.

• Basic or derived

Basic types are things like int or char that don’t have any constituent parts and can’t be subdivided or split
up into things with other types. A derived type is something like an array or a class that is built up from
smaller pieces.

• Built-in or user-defined

We already know that built-in functions are those function that come preloaded automatically with a
programming language. Similarly, built-in types are the types that we get automatically: things like int,
String, or double[]. Observe that built-in types can be either basic or derived, but usually all the basic types
are built-in.

User-defined types are... types that the programmer gets to define! In most modern languages, class definitions
are the most common examples of user-defined types. Other kinds of things that fit this bill are structs,
enums, and typedefs. If you get to name it, and it’s a type, then it’s a user-defined type! Languages that
allow user-defined types are more easily extensible than those that force you to use pre-existing types for all
variables.

• Declared or implicit

Any Java or C++ programmer knows that every variable declaration must specify the type of that variable.
This is important not only for compile-time checks, but (at least in the case of C++) to allocate storage for
that variable as well.

Many languages, especially modern compiled languages, do not require the types of variables to be declared.
For example, in the Scheme program

\[
(\text{define} \ x \ (+ \ 1 \ 2))
\]

there is no type specified for the variable x. But rest assured that x will be an integer when this program is
run! The type of x is determined implicitly, based on the type of whatever x gets assigned to.

Some languages, such as Haskell, are flexible about the whole thing; types can optionally be declared, but if
they aren’t the compiler will attempt to determine them based on how that variable gets used. This kind of
type inference is very powerful but can sometimes infer something the programmer did not intend, leading to
frustration.

• Static or dynamic

Perhaps the most important question about typing in a given programming language is whether it is static or
dynamic. This is really a question about type safety: when (if ever) are checks made to ensure that objects of
a given type only ever get used in the proper ways? If these checks occur at compile time, the language is said
to have static typing; at run-time, it’s called dynamic typing.
Static typing generally works by constructing the AST for a program and then decorating it with the type of each expression. If the arguments to any function or operator have a type mismatch, this inconsistency is identified at compile-time, before the code is actually run. This can be frustrating for beginner programmers, but is actually a blessing in disguise: in a statically typed language, we have some assurance that values in our programs are only accessed in valid ways. However, static typing is not easy to achieve; it requires either type declarations (annoying) or automatic type inference (difficult). And because static typing associates types with names in a program rather than just values, it means that the same variable cannot be assigned to things of radically different types.

Dynamic typing is what you have implemented in your SPL interpreter. There is no decoration of an AST or other compile-time checking, but the types of arguments to functions and operators may be checked immediately before they are used. This is less efficient than static typing at run-time because the types of arguments must be checked every time a block of code is executed, instead of just once in the compilation. However, dynamically-typed programming languages are more flexible in allowing names to be associated with all kinds of different values. Dynamic typing is especially appropriate for interpreted language where there is no compilation stage anyway.

It’s important to remember that just because types are not declared does not mean that a language has no type safety! For example, the following program is invalid in Python or in C++:

```python
x = 5;
x(5);
```

(The same variable x is being assigned to an integer and then accessed like a function.) The difference is that in C++, x would have to have been declared as either an int or a function, and then one of these lines would raise a compile-time error. In Python, this error would not be realized until this part of the code is actually reached by the running program.