CS442
Module 7: Imperative Programming

University of Waterloo

Winter 2025

> "Gotos aren't damnable to begin with. If you aren't smart enough to distinguish what's bad about some gotos from all gotos, goto hell."
>
> — Erik Naggum

1 Imperative Programming

You are almost certainly familiar with imperative programming languages. Most of the most popular programming languages in the world are imperative: C, C++, Java, JavaScript, Python, etc. Smalltalk, which you've been using in this course, is also imperative, although it's an unusual example. This module is fairly brief—really, it is, it just has several long figures to make it seem longer than it is!—both because you're expected to have familiarity with imperative programming languages and because imperative programming is fairly conceptually simple, particularly after logic programming.

In English, "imperative" is a synonym of "command"—it comes from the same root as "emperor"—and that's the core of its meaning in programming languages as well. An imperative language is a language in which the fundamental behavior is described by *imperatives*, i.e., commands, in sequence. In a functional language, *expressions* are the basic unit of behavior, and the ordering of evaluation is largely implicit. In an imperative language, *statements* are the basic unit of behavior, and a program is built from lists of statements. One statement is always completed before the next statement is executed, so the ordering is explicit in the code: the order of execution is the order of statements. Formally, an imperative is a single "step", which is usually a single statement, but some statements have nested statements within them, in which case one statement is formed from many imperatives. In practice, we will use the terms "statement", "imperative", and "command" interchangeably.

Aside: We have, of course, seen that the order of evaluation is perfectly well defined in functional languages. Furthermore, OCaml allows imperatives directly, with its `;` operator, and Haskell's `do` syntax allows imperative-like behavior as well. None of these categories are precise.

In order for commands to be able to communicate information from one to the next, all imperative languages have mutable variables. In functional languages, variables—`let` bindings and function arguments—are immutable, and mutability is encapsulated in the structure of a reference or monad. In an imperative language, statements may change variables, and the meaning of any statement can vary based on the values of the variables it uses at the time that the statement is executed.

Virtually all modern imperative languages have *procedures*. "Procedure" is, again, a normal English word, with the same meaning in programming languages as it has in English: a procedure is a list of steps, possibly parameterized. For instance, the procedure for walking a dog is parameterized by the dog, and the procedure for sorting an array is parameterized by the array. Pedantically, a procedure is distinct from a function because

- functions, as mathematical entities, are defined by expressions, not imperatives, and
- functions, as mathematical entities, are referentially transparent.

In practice, only the most pure functional languages restrict functions to this definition, so we will use the terms "function" and "procedure" interchangeably. Using these terms interchangeably also annoys pedants, which is always a laudable goal. *Procedural languages* are imperative languages with procedures. This accounts for virtually all imperative languages, so this module will mostly talk about procedural and imperative languages interchangeably. Note that although we will use "function" and "procedure" interchangeably, it is never correct to use "functional language" and "procedural language" interchangeably, as the former is a sub-paradigm of declarative languages, and the latter is a sub-paradigm of imperative languages.

There is yet another set of terms, *routine* and *subroutine*, which are also synonyms of "procedure", although some programming languages have more specific meanings for all of these terms within the context of the language. In English, "routine" is a synonym of "procedure" (albeit with different connotation), and "subroutine" is simply "routine" with the prefix "sub-" to imply the possibility of nesting. We will use the term "subroutine" when discussing Pascal, and define all of these terms more precisely for that language, but in all other contexts, we consider them all to be equivalent.

A language doesn't *need* procedures to be useful or Turing-complete, although it is impossible for a language to be Turing-complete if it has no way of repeating, so a non-procedural imperative language must have loops. Modern imperative programming languages without functions of any sort are rare. Assembly language is imperative and doesn't have functions on any real architectures, but all other examples are extremely special-purpose. Early imperative languages, in particular early versions of BASIC, lacked functions as well.

Another sub-paradigm within imperative programming is *structured programming*. Virtually all modern imperative languages are structured languages, so we will also mostly use these terms interchangeably. In a structured programming language, control flow—i.e., conditions and loops—are explicitly represented in the structure of the code. For instance, in our exemplar language, Pascal, to execute some statements `a`, `b`, and `c` only if a condition `x` is true, you write

```
if x then
  begin
  a;
  b;
  c
  end
```

The structure of the condition—that `a`, `b`, and `c` are only to be executed if `x` is true—is reflected in the structure of the code: the `a`, `b`, and `c` statements are nested inside of the `if` statement. We'll look more at Pascal syntax soon, but if you're more familiar with C syntax, the equivalent is

```
if (x) {
  a;
  b;
  c;
}
```

In an unstructured language, all control flow is done with jumps, such as `goto` in C, or `jmp` in many assembly languages. For instance, we could rewrite the above C example (quite badly) like so:

```
if (x) goto xTrue;
afterCondition:
[...]
xTrue:
a;
b;
c;
goto afterCondition;
```

In an unstructured language, the order of statements is less clear, since `goto`s can cause the order of execution to differ substantially from the order that statements appear. `goto`s are generally considered harmful [1], so we will mostly be looking at structured languages. Note that all functional languages are structured; the control flow is quite different, but it's still reflected in the structure of the code.

So, when we look at imperative languages, we're really going to be looking at structured, procedural programming languages. It just so happens that that accounts for nearly all modern imperative languages.

2 Examples

It is more difficult to select a single exemplar for imperative languages than any other paradigm. Most programming languages are imperative, and most imperative programming languages before object-oriented programming became popular were uniformly imperative, with little pollution from other paradigms, so there are dozens of programming languages which are, or at least were, purely imperative. As such, we will briefly discuss some other potential exemplars, before describing the exemplar we will be using, Pascal.

The obvious exemplar for imperative programming is C, and for the most part, it's purely imperative. Unlike many other examples—in fact, unlike even Pascal—C has remained purely imperative, and has no object-oriented or functional features to speak of. This is largely because C's object-oriented descendants have become just that, descendants, rather than new versions of the language. If you want to use objects in C, you should use C++ or Objective-C. If you want to use objects in Pascal, you should use a recent implementation of Pascal. Perversely, even if you want to use objects in COBOL, you need only use a recent version of COBOL. So, if C is such a good exemplar, why wasn't it chosen? Simple: it's not. C's pointers are pervasive and persistent, and require manual memory management as a constant fact of life. This is fine, but places it into an intersecting paradigm, systems programming languages, which we will discuss in Module 10, for which C is the exemplar. Further, C has some unusual syntactic anomalies—in particular, the C preprocessor—which make it difficult to relate to other languages.

Another possible exemplar was Forth, a stack-based language which is perhaps the most strictly imperative language in existence. Forth has no expressions, only commands. It accomplishes this by operating on a stack, precisely like the RPN calculator we developed in the Smalltalk component of Module 1. For instance, this Forth program computes $1 + 2 * 3$ and prints the result:

```
1 2 3 * + .
```

The behavior of a number command in Forth is to push that number to the stack. The behavior of a mathematical operator is to pop two numbers, perform the specified operator, and push the result to the stack. The behavior of `.` is to pop a number from the stack and print it. You may also define new commands in Forth from sequences of existing commands, forming procedures. Forth was not selected because its style is so foreign that it's hard to connect lessons learned about Forth to other imperative languages. Also, no two people can agree on how to capitalize Forth (FORTH? Forth? forth?), and my compromise of small-caps is hard to type.

Another option was BASIC, the programming language that defined the 80s. BASIC is a fairly straightforward, dynamically typed, imperative language. At this point, it would be reasonable to provide a short example of BASIC, but that's the problem: BASIC was pre-loaded on many 80s microcomputers, but each implementation was subtly incompatible, and the language evolved considerably over the course of the 80s, 90s, and early 2000s. So, the simple reason not to choose BASIC as an exemplar is that there is no BASIC language; it's more like a family of superficially related languages.

Arguably, Turing machines are imperative, and could serve as an exemplar as well. But, quite simply, very few languages derive their syntax or core structure from Turing machines. The Turing machine is a good theoretical basis, but not a good programming language.

3 Exemplar: Pascal

Pascal is an imperative, structured, procedural programming language originally designed by Niklaus Wirth around 1970 [2]. Pascal is statically typed but weakly typed. We won't focus much attention on its types, and the formal semantics we define will not have types. Pascal is an excellent exemplar, in particular, of procedural programming, because a Pascal program is fundamentally a tree of procedures. Many imperative programming languages do not allow procedures to nest, in the same way that functions can nest in functional languages, but Pascal does; it allows procedures to be defined anywhere where variables may be defined, and scopes everything lexically.

There are three kinds of procedures in Pascal: `program`s, `procedure`s, and `function`s. By the common definition of the term "procedure", and the definition we're using in this module, they are *all* procedures; Pascal simply has its own definitions for these terms. To avoid confusion, we will therefore use the term "subroutine" to refer to all of them, and the more specific terms to refer to the more specific structures in Pascal. When not discussing Pascal,

we will continue to use the terms interchangeably. In Pascal, a `program` is the outer subroutine for, predictably, a program, and is technically just a "routine" since it's not sub- to anything; a `procedure` is a subroutine which does not return anything, so is only used for its side effects; and a `function` is a subroutine which has a return value.

A Pascal subroutine consists of a *subroutine header*, which defines which of the three types of subroutines it is; a *declaration list*, which defines the environment for the subroutine; and a *subroutine body*. For instance, this is the classic Pascal "Hello, world!" program, with the addition of a variable to hold the string "Hello, world!" simply to demonstrate variable declarations:

```
1 program Hello;
2 var greeting: string;
3 begin
4     greeting := 'Hello, world!';
5     writeln(greeting)
6 end.
```

The line `program Hello;` is the subroutine header. A `program` subroutine has no properties other than its name (no arguments), and this subroutine's name is `Hello`. Naming a program in Pascal is mostly just for documentation, but becomes important with linking libraries; we will not be discussing libraries in this module, so for us, the name is just documentation.

The line `var greeting: string;` is a variable declaration, in this case declaring the variable `greeting` of the type `string`. There could be any number of variable declarations between `program` and `begin`, but variable declarations are *only* allowed there, not in the body. This separation of variable declarations from subroutine code is rare in modern imperative languages, but was the norm for much of imperative-language history. C only allowed mixing of declarations and statements in the C99 standard in 1999[1]!

The `begin` and `end` lines define a *block*, which is a list of statements, and this block forms the subroutine body. They are akin to C's `{` and `}`, which takes some getting used to, since they don't visually match in the same way. Lines 4 and 5 are statements. Statements in Pascal are separated by `;`. Note that they are *separated* by `;`, not *terminated* by `;`, so the final statement does not need a `;`, which is also true of Smalltalk's dot. Like Smalltalk, the final `;` (dot in Smalltalk) is allowed, so we could have written line 5 as `writeln(greeting);` if we wished. Technically, this is allowed because Pascal allows an empty statement, so if we end a statement list with `;`, then we're ending it with an empty statement. An empty statement does nothing. Like Smalltalk and OCaml, the assignment operator is `:=`.

Note that "block" is the common name for a statement list, but Smalltalk uses the term differently, to refer to (essentially) an anonymous function. Do not confuse the two! A block in this context is nothing more than a statement list. Blocks in Pascal (and most imperative languages) are not values, they're just a feature of the syntax.

One kind of declaration is a subroutine declaration, so we can have subroutines as part of the environment of other subroutines. For instance, if we wanted to use a subroutine to generate the greeting, we could define it like so:

```
1  program Hello;
2
3  var greeting: string;
4
5  function genGreeting(): string;
6  begin
7      genGreeting := 'Hello, world!'
8  end;
9
10 begin
11     greeting := genGreeting();
12     writeln(greeting)
13 end.
```

Line 5 is a `function` declaration: it consists of the keyword `function`, the name of the function (in this case, `genGreeting`), a list of arguments surrounded by parentheses (in this case, there are no arguments), a colon, the return type of the function, and then a semicolon. So, `genGreeting` is a zero-argument function which returns a string; a function is, of course, a kind of subroutine. To return a value in Pascal, you simply assign the return value to the name of the function, as in line 7. This doesn't end the subroutine, just set its return value, so we could have

[1] Some, but not all, pre-C99 C compilers allowed this as well, but C99 is when it was standardized.

added more statements after line 7, and they would have run before `genGreeting` returned. This is unusual, but fits the goals of structured programming: one cannot simply jump out of a block (in this case a subroutine body) whenever one chooses; blocks run to completion. All declarations in a declaration list are terminated by semicolons, so there is a semicolon after the `end` on line 8 to indicate that that is the end of the declaration of `genGreeting`. The `program` declaration is not part of a declaration list, but the entire program must end with a dot.

Declaration lists form environments with lexical scoping, so subroutines declared there can see and interact with variables defined there or in any enclosing scope. For instance, we could rewrite `genGreeting` as a `procedure` (subroutine without a return value) by making it directly modify `greeting`, like so:

```
1  program Hello;
2
3  var greeting : string;
4
5  procedure genGreeting();
6  begin
7      greeting := 'Hello, world!'
8  end;
9
10 begin
11     genGreeting();
12     writeln(greeting)
13 end.
```

A `procedure` declaration is similar to a `function` declaration, but with the word `procedure` and no return type (since there is no return).

All subroutines have declaration lists, but a declaration list may be empty. We could add declarations between lines 5 and 6 of our `genGreeting` example to create local variables of `genGreeting`, and even nested subroutines. Just like in functional languages, each call to a subroutine will have its own set of local variables, but if, for instance, two instances of `genGreeting` are called, that will *not* create two instances of `greeting`, because that's a variable of the program, not the subroutine.

Although `program`s are subroutines, their name does not form part of their own scope, so a program cannot call itself, and so cannot be directly recursive. Other subroutines are defined in an environment visible within the subroutine (the surrounding scope), so can be recursive. For instance, here's a program with a recursive implementation of a factorial function, in which the main subroutine simply outputs five factorial:

```
1  program FiveFac;
2
3  function fac(x: integer): integer;
4  begin
5      if x = 1 then
6          fac := 1
7      else
8          fac := fac(x-1) * x
9  end;
10
11 begin
12     writeln(fac(5))
13 end.
```

`if` statements behave similarly to other imperative languages, and have the same form: `if` *condition* `then` *statement* `else` *statement*. The `else` and its statement are optional. If you wish to perform multiple steps in a condition, you can use a block:

```
1  program FiveFac;
2
3  function fac(x: integer): integer;
4      var y: integer;
5  begin
6      if x = 1 then
7          fac := 1
8      else
9          begin
10         y := fac(x-1);
11         fac := y * x
12         end
```

```
13 end;
14
15 begin
16     writeln(fac(5))
17 end.
```

Technically speaking, because multiple statements can be nested inside an `if`, `if` is a statement, but not an imperative; an imperative is a single step.

Most imperative languages have loops, rather than just recursion, and indeed, some imperative languages do not support recursion. Most imperative languages have some form of compound data type—i.e., a way for a single variable to store or reference many values—through arrays, records, or both. We will discuss both of these features later in this module.

Pascal also has many features we will not discuss at all in this module. In particular, call-by-reference, Object Pascal, and units (libraries) are simply not relevant for us.

4 The Simple Imperative Language

The λ-calculus was a good starting point for functional languages, because it's fundamentally built out of functions (abstractions). But, it doesn't fit imperative languages well. Although our small-step semantics (the $\rightarrow$ morphism) does describe its reduction in terms of steps, there is no explicit statement of an order; no imperatives. Instead, we'll create a new fundamental language on which to build imperative concepts.

Like the λ-calculus, we want a language that is reasonably simple, yet captures most of our ideas about imperative programming. The language commonly used for these purposes is known as the Simple Imperative Language. There are many slight variants of the Simple Imperative Language, and the variant we will use is presented below:

Definition 1. The *Simple Imperative Language* consists of the string derivable from $\langle stmtlist \rangle$ below:

$$
\begin{aligned}
\langle stmtlist \rangle ::= \;& \langle stmt \rangle \\
\mid \;& \langle stmt \rangle \,;\, \langle stmtlist \rangle \\
\langle stmt \rangle ::= \;& \text{skip} \\
\mid \;& \text{begin } \langle stmtlist \rangle \text{ end} \\
\mid \;& \text{while } \langle boolexp \rangle \text{ do } \langle stmt \rangle \\
\mid \;& \text{if } \langle boolexp \rangle \text{ then } \langle stmt \rangle \text{ else } \langle stmt \rangle \\
\mid \;& \langle var \rangle := \langle intexp \rangle \\
\langle boolexp \rangle ::= \;& \text{true} \\
\mid \;& \text{false} \\
\mid \;& \text{not } \langle boolexp \rangle \\
\mid \;& \langle boolexp \rangle \text{ and } \langle boolexp \rangle \\
\mid \;& \langle boolexp \rangle \text{ or } \langle boolexp \rangle \\
\mid \;& \langle intexp \rangle > \langle intexp \rangle \\
\mid \;& \langle intexp \rangle < \langle intexp \rangle \\
\mid \;& \langle intexp \rangle = \langle intexp \rangle \\
\langle intexp \rangle ::= \;& 0 \mid 1 \mid \cdots \\
\mid \;& \langle var \rangle \\
\mid \;& \langle intexp \rangle + \langle intexp \rangle \\
\mid \;& \langle intexp \rangle * \langle intexp \rangle \\
\mid \;& - \langle intexp \rangle \\
\langle var \rangle ::= \;& a \mid b \mid c \mid \cdots
\end{aligned}
$$

In this definition, we see familiar constructions of integer and boolean expressions, as well as a looping construction and a conditional construction. The `skip` statement does nothing, and is thus fairly pointless; its purpose will become clear when we discuss the semantics of the Simple Imperative Language. Notice the exclusion of parentheses and other grouping constructions. Like with the λ-calculus, the syntax in our definition is assumed to be an abstract syntax, in which parsing has already been done, so we will use parentheses to disambiguate, but don't need them in the described syntax. A particular concrete syntax for the Simple Imperative Language would likely include grouping constructions in some fashion. Notice also that many familiar constructions, including subtraction, have been excluded from this language. Many of these can be introduced later as additional syntax or syntactic sugar. For example, the expression $a - b$ can be viewed as syntactic sugar for $a + -b$.

Also notable about the Simple Imperative Language is the absence of a `goto` statement. Although simply called "imperative", the Simple Imperative Language is, more precisely, a structured imperative language.

Formulating an operational semantics for an imperative language requires some care; imperative languages are fundamentally different from functional languages, in that they are based on the execution of commands, rather than the evaluation of expressions. Thus, we cannot formulate a semantics for an imperative language based on some "final" computed value, because there is no such value. Imperative languages are run for their side effects.

Our formulation of an operational semantics for imperative languages must begin with an understanding of what a program in an imperative language "does". With a real program, what we probably care about is how it interacts with the user, but modeling input and output is typically beyond the scope of formal semantics. Instead, we observe that computation is performed by assigning values to mutable variables, and results are obtained by reading off the values of one or more of the variables. Variables are nothing more or less than named locations in a mutable store (the "memory"). Hence, an operational semantics for an imperative language should include some notion of a "store" of values, and the final store will be a program's result.

We've seen a store before, σ. But, the term "store" and the concept of a store are more general than we've used before: a store is simply a place to store things. In fact, our heap, Σ, is also a kind of store, but we usually only use the term "store" to refer to stores indexed by variable names.

In functional languages, variables are always immutable (invariable?), with mutation allowed by having variables store references, and references allow mutation. In our formal semantics, this was represented by σ being immutable (it never changed across $\rightarrow$), but Σ (the heap) being mutable. In the Simple Imperative Language, we will simply allow σ to mutate, and thus won't need Σ at all. Note that as semantics get more complicated, it's not uncommon to reintroduce this two-way link (variable in σ to label in Σ), simply because it's difficult to make σ simultaneously represent both the environment, which changes as you go down the code's syntax tree, and the mutable store, which changes across steps of reduction. The Simple Imperative Language has only a single global scope, so we can use σ for both, and when we introduce procedures, we'll use a different trick to stick to just σ.

In terms of the actual structure and operations allowed, σ in the Simple Imperative Language behaves the same as σ in previous modules.

The important values for our semantics of the Simple Imperative Language will be stores, and terminal values will be stores with no further statements to execute. That is, our program has terminated when it has run every statement, and the value it produces is simply the store. In addition to returning a store as our final answer, we must keep in mind that the meaning of a command like `x := x + 1` can only be determined in the context of a store as well; we need to know the value bound to `x`! Thus, just like with `let` bindings, our semantics will operate over pairs of the form $\langle \sigma, S \rangle$, so we will be defining the morphism $\langle \sigma, S \rangle \rightarrow \langle \sigma', S' \rangle$, where S is a program ($\langle stmtlist \rangle$) and σ is a store. As σ will change over $\rightarrow$, it represents mutable state, so we can also describe this as a change in both program and state. Only the `:=` statement actually modifies σ, so for all other statements and expressions, if $\langle \sigma, X \rangle \rightarrow \langle \sigma', X' \rangle$, then $\sigma = \sigma'$. Our program will start with an empty store.

In functional languages, expressions are primary, and so every component of a program that can be evaluated is an expression. In the Simple Imperative Language, there are both statements and expressions. We can separate concerns, and thus make our semantics easier to understand, by defining the semantics for an expression sublanguage separately from the semantics for the statement language, then linking them together with a rule that says one step a statement can take is an expression within the statement taking a step. The semantics for an expression sublanguage will look similar to the semantics for a functional language, because expressions are primary there. The Simple Imperative Language possesses two expression sublanguages: the sublanguage of integer expressions and the sublanguage of boolean expressions.

To formulate a semantics for these sublanguages, we can use our semantics for numbers in the λ-calculus as a model:

Definition 2. (Operational Semantics for Integer Expressions)

We identify the set of terminal values in the semantics of the sublanguage of integer expressions within the Simple Imperative Language as the set of integers. Let the metavariables M, x, σ, and N range over integer expressions, integer variables, stores, and integers, respectively. A small-step semantics for integer expressions is as follows:

$$\textsc{IntOpLeft}\ \frac{op \in \{+, *\} \qquad \langle \sigma, M_1 \rangle \to \langle \sigma, M_1' \rangle}{\langle \sigma, M_1\ op\ M_2 \rangle \to \langle \sigma, M_1'\ op\ M_2 \rangle} \qquad \textsc{IntOpRight}\ \frac{op \in \{+, *\} \qquad \langle \sigma, M \rangle \to \langle \sigma, M' \rangle}{\langle \sigma, N\ op\ M \rangle \to \langle \sigma, N\ op\ M' \rangle}$$

$$\textsc{Add}\ \frac{N_1 + N_2 = N_3}{\langle \sigma, N_1 + N_2 \rangle \to \langle \sigma, N_3 \rangle} \qquad \textsc{Mul}\ \frac{N_1 N_2 = N_3}{\langle \sigma, N_1 * N_2 \rangle \to \langle \sigma, N_3 \rangle}$$

$$\textsc{NegStep}\ \frac{\langle \sigma, M \rangle \to \langle \sigma, M' \rangle}{\langle \sigma, -M \rangle \to \langle \sigma, -M' \rangle} \qquad \textsc{Neg}\ \frac{N' = -N}{\langle \sigma, -N \rangle \to \langle \sigma, N' \rangle}$$

$$\textsc{Var}\ \frac{N = \sigma(x)}{\langle \sigma, x \rangle \to \langle \sigma, N \rangle}$$

The semantics are very similar to those for natural numbers from Module 3, with a few exceptions:

- Operators in the Simple Imperative Language are infix ($a + b$) instead of prefix ($+\ a\ b$).
- Numbers in the Simple Imperative Language are integers, not naturals, so there are no special cases.
- We've abbreviated the repetitive rules for reducing the left before the right into IntOpLeft and IntOpRight, which match any binary operation.

The integer expression sublanguage is pure functional, and so σ is never changed. We could have defined $\to$ without the σ on the right at all, since it's guaranteed not to change, but this makes the definition of $\to^*$ awkward, so we've defined it with both sides having the same form. Ultimately, what we care about for a Simple Imperative Language program is how it changes the store, but as the integer sublanguage does not change the store, for it, we care about the value generated, just like in functional languages. Generally speaking, expression sublanguages of all imperative languages behave in this way.

Note that if $\sigma(x)$ is not defined for a particular variable x in an application of the semantic rules, these rules get stuck. That is the only circumstance under which these rules can get stuck: as we will see when we get to `:=`, all variables store integers in the Simple Imperative Language, so there's no possibility of any other type error.

The formulation of a semantics for boolean expressions is similar:

Definition 3. (Operational Semantics for Boolean Expressions)

We identify the set of terminal values in the semantics of the sublanguage of boolean expressions within the Simple Imperative Language as the two-element set $B = \{\text{true}, \text{false}\}$, which is the set of boolean values. Let the metavariables B, M, σ, N, and V range over boolean expressions, integer expressions, stores, integer values, and boolean values, respectively. Then a big-step semantics for boolean expressions is as follows:

$$\textsc{BoolOpLeft}\ \frac{op \in \{>, <, =\} \qquad \langle \sigma, M_1 \rangle \to \langle \sigma, M_1' \rangle}{\langle \sigma, M_1\ op\ M_2 \rangle \to \langle \sigma, M_1'\ op\ M_2 \rangle}$$

$$\textsc{BoolOpRight}\ \frac{op \in \{>, <, =\} \qquad \langle \sigma, M \rangle \to \langle \sigma, M' \rangle}{\langle \sigma, N\ op\ M \rangle \to \langle \sigma, N\ op\ M' \rangle}$$

$$\textsc{GtTrue}\ \frac{N_1 > N_2}{\langle \sigma, N_1 > N_2 \rangle \to \langle \sigma, \text{true} \rangle} \qquad \textsc{GtFalse}\ \frac{N_1 \leq N_2}{\langle \sigma, N_1 > N_2 \rangle \to \langle \sigma, \text{false} \rangle}$$

$$\textsc{LtTrue}\ \frac{N_1 < N_2}{\langle \sigma, N_1 < N_2 \rangle \to \langle \sigma, \text{true} \rangle} \qquad \textsc{LtFalse}\ \frac{N_1 \geq N_2}{\langle \sigma, N_1 < N_2 \rangle \to \langle \sigma, \text{false} \rangle}$$

$$\textsc{EqTrue}\ \frac{N_1 = N_2}{\langle \sigma, N_1 = N_2 \rangle \to \langle \sigma, \text{true} \rangle} \qquad \textsc{EqFalse}\ \frac{N_1 \neq N_2}{\langle \sigma, N_1 = N_2 \rangle \to \langle \sigma, \text{false} \rangle}$$

$$\textsc{NotSub}\ \frac{\langle \sigma, B \rangle \to \langle \sigma, B' \rangle}{\langle \sigma, \text{not } B \rangle \to \langle \sigma, \text{not } B' \rangle}$$

$$\textsc{NotTrue}\ \frac{}{\langle \sigma, \text{not true} \rangle \to \langle \sigma, \text{false} \rangle} \qquad \textsc{NotFalse}\ \frac{}{\langle \sigma, \text{not false} \rangle \to \langle \sigma, \text{true} \rangle}$$

$$\textsc{AndLeft}\ \frac{\langle \sigma, B_1 \rangle \to \langle \sigma, B_1' \rangle}{\langle \sigma, B_1 \text{ and } B_2 \rangle \to \langle \sigma, B_1' \text{ and } B_2 \rangle} \qquad \textsc{AndSC}\ \frac{}{\langle \sigma, \text{false and } B \rangle \to \langle \sigma, \text{false} \rangle}$$

$$\textsc{AndRight}\ \frac{}{\langle \text{true and } B, \sigma \rangle \to \langle B, \sigma \rangle}$$

$$\textsc{OrLeft}\ \frac{\langle \sigma, B_1 \rangle \to \langle \sigma, B_1' \rangle}{\langle \sigma, B_1 \text{ or } B_2 \rangle \to \langle \sigma, B_1' \text{ or } B_2 \rangle} \qquad \textsc{OrSC}\ \frac{}{\langle \sigma, \text{true or } B \rangle \to \langle \sigma, \text{true} \rangle}$$

$$\textsc{OrRight}\ \frac{}{\langle \sigma, \text{false or } B \rangle \to \langle \sigma, B \rangle}$$

The rules for and, or, and not should be familiar from Module 3, with the following changes:

- Like integer operations, boolean binary operations are infix (a or b) not prefix (or a b).
- Short-circuiting (only evaluate the right of and/or if necessary) is described slightly differently, though with the same results.

In addition, these rules add integer comparisons. Note that the subexpressions of a comparison operator ($<$, $>$, $=$) *must*, by the syntax of the Simple Imperative Language, be integer expressions. We will see later that our rules do not allow variables to hold booleans, so integer expressions can only evaluate to integers. This creates a strict stratification of expressions: boolean expressions may contain integer expressions, but integer expressions may not contain boolean expressions.

The rules in the above definitions provide us with a complete semantics for the functional subset of the Simple Imperative Language. We may now consider the formulation of a semantics for the language as a whole:

Definition 4. (Operational Semantics for the Simple Imperative Language)

The set of terminal values in the Simple Imperative Language is pairs of a store and the `skip` statement[2], i.e., $\langle \sigma, \text{skip} \rangle$. We will define reductions over both statement lists and individual statements. Let the metavariables B, M, σ, V, N, Q, and L range over boolean expressions, integer expressions, stores, boolean values, integer values, statements, and statement lists, respectively. Then a small-step semantics for the Simple Imperative Language is as follows:

[2]Our syntax doesn't allow empty statement lists, but we may have a statement list containing only the statement `skip`, which is effectively empty.

$$\textsc{BlockRest}\ \frac{}{\langle \sigma, \text{begin } L_1 \text{ end}; L_2\rangle \to \langle \sigma, L_1; L_2\rangle} \qquad \textsc{BlockOnly}\ \frac{}{\langle \sigma, \text{begin } L \text{ end}\rangle \to \langle \sigma, L\rangle}$$

$$\textsc{Skip}\ \frac{}{\langle \sigma, \text{skip}; L\rangle \to \langle \sigma, L\rangle} \qquad \textsc{StmtList}\ \frac{\langle \sigma, Q\rangle \to \langle \sigma', Q'\rangle}{\langle \sigma, Q; L\rangle \to \langle \sigma', Q'; L\rangle}$$

$$\textsc{AssignStep}\ \frac{\langle \sigma, M\rangle \to \langle \sigma, M'\rangle}{\langle \sigma, x := M\rangle \to \langle \sigma, x := M'\rangle} \qquad \textsc{Assign}\ \frac{\sigma' = \sigma[x \mapsto N]}{\langle \sigma, x := N\rangle \to \langle \sigma', \text{skip}\rangle}$$

$$\textsc{IfCond}\ \frac{\langle \sigma, B\rangle \to \langle \sigma, B'\rangle}{\langle \sigma, \text{if } B \text{ then } Q_1 \text{ else } Q_2\rangle \to \langle \sigma, \text{if } B' \text{ then } Q_1 \text{ else } Q_2\rangle}$$

$$\textsc{IfTrue}\ \frac{}{\langle \sigma, \text{if true then } Q_1 \text{ else } Q_2\rangle \to \langle \sigma, Q_1\rangle}$$

$$\textsc{IfFalse}\ \frac{}{\langle \sigma, \text{if false then } Q_1 \text{ else } Q_2\rangle \to \langle \sigma, Q_2\rangle}$$

$$\textsc{While}\ \frac{}{\langle \sigma, \text{while } B \text{ do } Q\rangle \to \langle \sigma, \text{if } B \text{ then begin } Q; \text{while } B \text{ do } Q \text{ end else skip}\rangle}$$

We've used Q instead of S for statements because S has previously been used for substitutions, and we will soon be using substitutions in the Simple Imperative Language, though they aren't needed yet.

The BlockRest, BlockOnly, Skip, and StmtList rules are over statement lists; the rest are over individual statements. BlockRest and BlockOnly simply describe the destructuring of a block: if we have a block as a statement, we simply replace it with its constituent statements. There are two such cases because there are two forms for ⟨*stmtlist*⟩. The Skip rule tells us that the `skip` statement does nothing, so if it is the first statement, it will simply be removed. The StmtList rule specifies that if there is a reduction rule for the first statement in a statement list, then we can use that to reduce the first statement in the list. The individual statement reduction rules are designed to eventually reduce to `skip`, so that multiple steps of the StmtList (and sometimes Block*) rules will apply, and then a final Skip when the statement is done.

The AssignStep rule reduces the right-hand side of an assignment statement. Syntactically, the right-hand side of an assignment statement can only be an integer expression, so assuming the reduction doesn't get stuck, it will always reduce to an integer. The Assign rule then assigns that integer to a variable, by replacing σ with a modified σ', which is σ with the mapping for the variable to its value added.

The IfCond rule takes a single step over the condition of an `if` statement. Multiple steps of IfCond will be taken until the condition is either `true` or `false`. At that point, the IfTrue or IfFalse rule, respectively, will replace the `if` statement with the sub-statement which should actually be executed; the first for `true`, the second for `false`. Other rules may then continue to reduce the statement.

4.1 Loops

The While rule is unusual, in that it "reduces" a `while` statement into a *longer* `if` statement, and that longer `if` statement actually contains the original `while` statement! To understand why this works, and examine the rest of the reductions while we're at it, let's run a simple program that increments the variable `x` from 0 to 2 in a loop. The resulting reduction steps are shown in Figure 1. Since a `while` statement reduces to an `if`, its body is only run conditionally. Since the whole `while` statement is at the end of the `then` case of the `if` statement, after the loop body finishes running, the `while` statement simply runs again. This cycle continues until the generated `if` statement's condition is false, at which point it reduces to `skip`.

Of course, a real implementation of an imperative language doesn't mutate the statement list in this way, but this is a reasonable description of the steps to performing a loop. Each iteration is a simple conditional, and it is the fact that the condition ends by repeating the loop that defines `while`'s behavior.

$$\langle \{\}, \texttt{x := 0; while x < 2 do x := x + 1} \rangle$$
$$(\textsc{Assign}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{0}\}, \texttt{skip; while x < 2 do x := x + 1} \rangle$$
$$(\textsc{Skip}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{0}\}, \texttt{while x < 2 do x := x + 1} \rangle$$
$$(\textsc{While}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{0}\}, \texttt{if x < 2 then begin x := x + 1; while x < 2 do x := x + 1 end else skip} \rangle$$
$$(\textsc{Var}, \textsc{LtTrue}) \rightarrow^*$$
$$\langle \{\texttt{x} \mapsto \texttt{0}\}, \texttt{if true then begin x := x + 1; while x < 2 do x := x + 1 end else skip} \rangle$$
$$(\textsc{IfTrue}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{0}\}, \texttt{begin x := x + 1; while x < 2 do x := x + 1 end} \rangle$$
$$(\textsc{BlockOnly}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{0}\}, \texttt{x := x + 1; while x < 2 do x := x + 1} \rangle$$
$$(\textsc{Var}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{0}\}, \texttt{x := 0 + 1; while x < 2 do x := x + 1} \rangle$$
$$(\textsc{Add}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{0}\}, \texttt{x := 1; while x < 2 do x := x + 1} \rangle$$
$$(\textsc{Assign}, \textsc{Skip}) \rightarrow^*$$
$$\langle \{\texttt{x} \mapsto \texttt{1}\}, \texttt{while x < 2 do x := x + 1} \rangle$$
$$(\textsc{While}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{1}\}, \texttt{if x < 2 then begin x := x + 1; while x < 2 do x := x + 1 end else skip} \rangle$$
$$(\textsc{Var}, \textsc{LtTrue}) \rightarrow^*$$
$$\langle \{\texttt{x} \mapsto \texttt{1}\}, \texttt{if true then begin x := x + 1; while x < 2 do x := x + 1 end else skip} \rangle$$
$$(\textsc{IfTrue}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{1}\}, \texttt{begin x := x + 1; while x < 2 do x := x + 1 end} \rangle$$
$$(\textsc{BlockOnly}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{1}\}, \texttt{x := x + 1; while x < 2 do x := x + 1} \rangle$$
$$(\textsc{Var}, \textsc{Add}, \textsc{Assign}, \textsc{Skip}) \rightarrow^*$$
$$\langle \{\texttt{x} \mapsto \texttt{2}\}, \texttt{while x < 2 do x := x + 1} \rangle$$
$$(\textsc{While}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{2}\}, \texttt{if x < 2 then begin x := x + 1; while x < 2 do x := x + 1 end else skip} \rangle$$
$$(\textsc{Var}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{2}\}, \texttt{if 2 < 2 then begin x := x + 1; while x < 2 do x := x + 1 end else skip} \rangle$$
$$(\textsc{LtFalse}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{2}\}, \texttt{if false then begin x := x + 1; while x < 2 do x := x + 1 end else skip} \rangle$$
$$(\textsc{IfFalse}) \rightarrow$$
$$\langle \{\texttt{x} \mapsto \texttt{2}\}, \texttt{skip} \rangle$$

Figure 1: Reduction steps for `x := 0; while x < 2 do x := x + 1`

`while` loops in Pascal have the same form as `while` loops in the Simple Imperative Language. Pascal also has two other kinds of loops (`for`-`do` loops and `repeat`-`until` loops), but both can easily be viewed as syntactic sugar for `while` loops, so we will not discuss them here.

4.2 The Initialization Problem

The Simple Imperative Language is quite nearly type safe, in spite of not having any explicit types. This is because of our expression stratification; although there exist boolean and integer values in the language, variables can only contain integers, and it is syntactically impossible to write the wrong type. We will lose this almost-type-safety when we introduce other primitives into the language, but we can also examine why we don't have true type safety now.

The Simple Imperative Language is not *quite* type safe, as demonstrated by this simple program which gets stuck: `x := y`. Since we never defined `y`, the premise for the Var reduction doesn't match, and so the reduction is stuck. Not all variables which are used are guaranteed to be in the store. As with all cases where semantics get stuck, "getting stuck" isn't meaningful for a real language implementation, but it is an indication of a real problem: we haven't guaranteed that variables are initialized before they are used. This is called *the initialization problem*, and is a surprisingly pervasive problem in the definition of imperative languages.

There are four solutions (of which the first is a non-solution) to the initialization problem: garbage values, default values, forced initialization, and static analysis.

A language that uses *garbage values* leaves the value of an uninitialized variable unpredictable and has no guarantees whatsoever, because its value is simply whatever was in a slot of memory before the variable was assigned to that space. Pascal and, famously, C and C++ use garbage values, so if you fail to initialize a variable, it could have any unpredictable value. With a data type such as integers, garbage values are harmless in terms of type safety, since all sequences of bits can be interpreted as valid integers. This is not true of other data types, however, so garbage values are generally not safe. Formal semantics usually do not model garbage values, because formal semantics usually do not actually model RAM, but it is possible to model garbage values with a rule like this one:

$$\textsc{GarbageVar}\ \frac{x \notin \sigma \qquad N \in \mathbb{Z} \qquad \sigma' = \sigma[x \mapsto N]}{\langle \sigma, x \rangle \to \langle \sigma', N \rangle}$$

It is exceedingly rare for a semantics to have a rule like this one, because it's non-deterministic! This allows us to take the step $\langle \sigma, x \rangle \to \langle \sigma', N \rangle$ for *any* integer N, and any such step is valid. Of course, that accurately models the actual behavior of such a language, so as rare as it is, it is correct; formal semantics authors just usually don't want non-deterministic semantics.

> **Aside:** Another way to semantically define garbage values is to semantically model memory. We'll look at this option again in Module 10.

In a language with *default values*, every type has a default value (or, if there are no types, there is a single default value), and every variable is assigned that default value until it is initialized. For instance, we could represent default values in the Simple Imperative Language by adding a rule to handle uninitialized variables, like so:

$$\textsc{UninitVar}\ \frac{x \notin \sigma}{\langle \sigma, x \rangle \to \langle \sigma, 0 \rangle}$$

Languages like Java and JavaScript use default values.

An example of *forced initialization* is the `let` bindings we saw in functional languages. The syntax of a `let` binding forces the programmer to give a value to the variable, and the variable is only defined in the scope of the `let` binding, so the initialization problem simply never arises. Forced initialization has its drawbacks, however: it makes recursive data types (such as a circular linked list) extremely hard or impossible to define. Forced initialization is uncommon among imperative languages, and common among functional languages.

Finally, *static analysis* is a general name for analyzing code. In some cases, it's possible to inspect code and reject programs which might access variables before they've been initialized. It is rare for a language to make such static analysis part of the definition of the language, because that means the language is defined by a particular algorithm, but it is common to use static analysis to provide warnings for the possibility of an uninitialized variable. Most C, C++, and Pascal compilers warn programmers about possibly uninitialized variables using static analysis.

You may wonder why we don't simply demand that for any variable `x`, an assignment `x := M` appear before any use of `x`. The problem is the definition of "before". With conditionals and loops, it's not always obvious whether a given statement will definitely happen before another, and when we add procedures, arrays, and records, this problem will only get worse. In the Simple Imperative Language, we could probably become safe by demanding that `x` be initialized in a non-conditional statement before its first use, which is actually an extremely simplistic form of static analysis.

4.3 Types

The Simple Imperative Language has few type errors, since variables can only hold integers and the language doesn't allow to to mix integer expressions and boolean expressions. But, the Simple Imperative Language does have an initialization problem: nothing in the language prevents us from using a variable before it's been defined. We can use type judgment to solve the initialization problem, by writing it with the simplistic static analysis described at the end of the last subsection. Note that we have no explicit type declarations in the Simple Imperative Language, but as seen with the polymorphic λ-calculus, explicit types are not actually necessary to perform type judgment.

Recall that our type judgments have previously been of the form $\Gamma \vdash E : \tau$, where Γ is a type environment, E is an expression, and τ is the determined type of that expression. But, what if instead of an expression, we had a statement? Statements do not actually have a type; statements are an action. In this case, we simply write a type judgment that does not yield a type (has no τ). We can judge a statement to be well-typed, but cannot actually give a type to the statement, since it does not yield a value. Remember that type judgment served two purposes: to predict the type of each expression once it's evaluated to a value, and to guarantee that our semantics won't get stuck due to type errors, or concretely, that there will be no type errors at runtime. We can't give the type of a statement, but we can still assure that executing it won't raise type errors. We therefore write type judgments of statements as $\Gamma \vdash Q$. That is, our type judgments are written without an actual type ($\Gamma \vdash Q$ instead of $\Gamma \vdash E : \tau$). This states that Q is well-typed in Γ, which means that the execution of Q won't get stuck. In our type judgment for the Simple Imperative Language, we will see both forms, since we have both expressions and statements.

As an additional simplification, we will assume that all programs consist of a statement list in which the last statement is explicitly specified as `skip`. All programs can be rewritten like this simply by adding `; skip` to the end. The reason for this is that our type environment will carry information from the first statement in a statement list to the rest of the statement list, so structurally we want to guarantee that the rest of the statement list will always be there. This wasn't necessary for our semantics, but would be a harmless change there.

First, we'll need a type language. We only have two types, so our type language is insultingly simple:

$$\langle \textit{type} \rangle ::= \text{ int } | \text{ bool}$$

Every value is either an integer (int) or a boolean (bool).

Now, let's type integer expressions. Integer expressions *do* have a type—every integer expression should yield an integer—so the type judgment of integer expressions will be of the form $\Gamma \vdash M : \text{ int}$. In a language with more diverse expressions, "int" could of course be any valid type.

Definition 5. (Type Judgment for Integer Expressions)

Let the metavariables Γ, τ, M, x, and N range over type environments, types, integer expressions, variables, and integers. A type judgment for integer expressions is as follows:

$$\text{T\_IntBinOp}\ \frac{op \in \{+, *\} \qquad \Gamma \vdash M_1 : \text{ int} \qquad \Gamma \vdash M_2 : \text{ int}}{\Gamma \vdash M_1\ op\ M_2 : \text{ int}}$$

$$\text{T\_Neg}\ \frac{\Gamma \vdash M : \text{ int}}{\Gamma \vdash -M : \text{ int}} \qquad \text{T\_Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{T\_Int}\ \frac{}{\Gamma \vdash N : \text{ int}}$$

Again, quite simple: since integer expressions are syntactically isolated, all we need to do is check that all subexpressions type correctly. Even that is only necessary because T_Var requires that the variable actually be defined.

Next, let's type boolean expressions:

Definition 6. (Type Judgment for Boolean Expressions)

Let the metavariables Γ, τ, B, and M range over type environments, types, boolean expressions, and integer expressions. A type judgment for boolean expressions is as follows:

$$\text{T\_CmpOp}\ \frac{op \in \{>, <, =\} \qquad \Gamma \vdash M_1 : \text{ int} \qquad \Gamma \vdash M_2 : \text{ int}}{\Gamma \vdash M_1\ op\ M_2 : \text{ bool}}$$

$$\text{T\_BoolBinOp}\ \frac{op \in \{\text{and, or}\} \qquad \Gamma \vdash B_1 : \text{ bool} \qquad \Gamma \vdash B_2 : \text{ bool}}{\Gamma \vdash B_1\ op\ B_2 : \text{ bool}}$$

$$\text{T\_Not}\ \frac{\Gamma \vdash B : \text{ bool}}{\Gamma \vdash \text{not } B : \text{ bool}} \qquad \text{T\_True}\ \frac{}{\Gamma \vdash \text{true } : \text{ bool}} \qquad \text{T\_False}\ \frac{}{\Gamma \vdash \text{false } : \text{ bool}}$$

Again, the rules are straightforward. Comparisons are well-typed if their operands are both integers. Other boolean operations are well-typed if their subexpressions are booleans.

Finally, the type judgment for the whole program. Statements do not have types, so this is where we'll use $\Gamma \vdash Q$, without a resulting type τ.

Definition 7. (Type Judgment for the Simple Imperative Language)

Let the metavariables Γ, τ, B, M, Q, and L range over type environments, types, boolean expressions, integer expressions, statements, and statement lists. A type judgment for Simple Imperative Language programs suffixed with `; skip` follows:

$$\text{T\_SkipOnly}\ \frac{}{\Gamma \vdash \text{skip}} \qquad \text{T\_SkipRest}\ \frac{\Gamma \vdash L}{\Gamma \vdash \text{skip}; L} \qquad \text{T\_Block}\ \frac{\Gamma \vdash L_1 \qquad \Gamma \vdash L_2}{\Gamma \vdash \text{begin } L_1 \text{ end}; L_2}$$

$$\text{T\_Assign}\ \frac{\Gamma \vdash M : \tau \qquad \forall \tau_1.\, \langle x, \tau_1 \rangle \notin \Gamma \qquad \langle x, \tau \rangle + \Gamma \vdash L}{\Gamma \vdash x := M; L}$$

$$\text{T\_Reassign}\ \frac{\Gamma \vdash M : \tau \qquad \Gamma(x) = \tau \qquad \Gamma \vdash L}{\Gamma \vdash x := M; L}$$

$$\text{T\_If}\ \frac{\Gamma \vdash B : \text{ bool} \qquad \Gamma \vdash Q_1;\ \text{skip} \qquad \Gamma \vdash Q_2;\ \text{skip} \qquad \Gamma \vdash L}{\Gamma \vdash \text{if } B \text{ then } Q_1 \text{ else } Q_2; L}$$

$$\text{T\_While}\ \frac{\Gamma \vdash B : \text{ bool} \qquad \Gamma \vdash Q;\ \text{skip} \qquad \Gamma \vdash L}{\Gamma \vdash \text{while } B \text{ do } Q; L}$$

Most of these rules are fairly trivial: a statement types if its sub-statements and subexpressions type. T_Assign and T_Reassign are the interesting cases, but let's quickly look at the other cases. Our statement list is guaranteed to be terminated by `skip`, so only T_SkipOnly and T_SkipRest are concerned with the possibility of not having a "rest" of the statement list. Everything else assumes that all statement lists are of the form `Q; L`. To make this work with the sub-statements in `if` and `while` statements, we explicitly append `; skip`, turning a statement into a statement list.

Now, let's examine T_Assign and T_Reassign. These are the only type judgments that affect Γ. T_Assign is for initial assignments, and so checks that the assigned variable, x, is *not* present in Γ ($\forall \tau_1 . \langle x, \tau_1 \rangle \notin \Gamma$, i.e., there is no type associated with x in Γ). It judges the remaining statements L in an environment that includes the variable x. Once a variable has been given a type, we don't allow it to change, since if it changes *conditionally*, this will make it possible for a variable to have values of different types in different circumstances. Thus, the T_Reassign rule rejects assignments where x is present but has a different type. Of course, in the Simple Imperative Language, syntactically, only one type is possible, int. This rule will be more interesting when we introduce other types.

Consider how T_Assign and T_Reassign relate to nested statements and nested blocks. If we first assign a variable inside of an `if` statement, then its type will not be visible outside the `if` statement. But, if we first assign it *before* the `if` statement, its type *will* be visible within the `if` statement. In this way, we've actually given some lightly lexical scoping to the Simple Imperative Language, which only has a single global scope. This is possible and correct only because of the strict ordering of statements.

Most typed imperative languages would instead require explicit variable declarations, and extend Γ with declared types. If those variable declarations have lexical scopes, then the semantics must be modified to nest σ correctly as well.

5 Procedures

The Simple Imperative Language has no procedures, and no nested scopes. There is only a global scope and global variables. As we've discussed, procedures are not necessary for an imperative language to be Turing-complete—and indeed, as long as our integers have unlimited range, the Simple Imperative Language *is* Turing-complete, albeit quite awkward to use—but procedures are extremely common. There are several ways to represent procedures, but the simplest uses substitution and "freshening" (creating new, distinct variable names) like we saw in Module 2. To do this, we will need a syntax for procedures. We will put procedures and variables in the same store—in Pascal, they're in the same environment. Our procedures will have a similar form to Pascal procedures: a subroutine header, a declaration list, and a body. For simplicity, we will implement `procedure`s, and not `function`s. The declaration list will only consist of names, since the Simple Imperative Language is not typed. And, our procedures will be *statements*, not declarations; essentially, they are a form of assignment statement in which the assignment itself is implicit (we are assigning a variable to a procedure).

> **Aside:** Just like typed functional languages tend to have a file syntax distinct from their expression syntax, most typed imperative languages allow procedure and variable declarations at the global scope, but no expressions, and many do not allow procedure definitions as statements in other procedures. This creates several syntaxes within the same language which partially overlap. We've somewhat sidestepped this issue by using Pascal as our exemplar, because its global scope *is* a procedure!

This introduces another use of semicolon, since procedure declarations use semicolons after the header and after each declaration. This is unambiguous with the semicolons which separate statements only because a procedure must end with a `begin`-`end` block. Unfortunately, this will create some hard-to-read syntax; remember to look for the corresponding `begin`-`end` block whenever you see a procedure.

We will call our extended version of Simple Imperative Language with procedures SIL-P. This is our Pascal factorial program from above, rewritten in SIL-P, and storing the result of `fac(5)` in the variable `x`:

```
procedure fac(n); begin if n = 1 then x := 1 else begin fac(n+-1); x := x * n end end; fac(5)
```

Rewriting this with some indentation for clarity:

```
procedure fac(n);
begin
    if n = 1 then
        x := 1
    else
        begin
        fac(n + -1);
        x := x * n
        end
end;
fac(5)
```

We extend the Simple Imperative Language as follows:

$$
\begin{aligned}
\langle stmt \rangle &::= \cdots \\
&\mid \langle procdec \rangle \\
&\mid \langle var \rangle\,(\langle arglist \rangle) \\
\langle procdecl \rangle &::= \text{procedure } \langle var \rangle\,(\langle paramlist \rangle);\langle decllist \rangle \text{ begin } \langle stmtlist \rangle \text{ end} \\
\langle arglist \rangle &::= \epsilon \\
&\mid \langle intexp \rangle\,\langle arglistrest \rangle \\
\langle arglistrest \rangle &::= \epsilon \\
&\mid ,\langle intexp \rangle\,\langle arglistrest \rangle \\
\langle paramlist \rangle &::= \epsilon \\
&\mid \langle var \rangle\,\langle paramlistrest \rangle \\
\langle paramlistrest \rangle &::= \epsilon \\
&\mid ,\langle var \rangle\,\langle paramlistrest \rangle \\
\langle decllist \rangle &::= \epsilon \\
&\mid \langle var \rangle\,;\langle decllist \rangle
\end{aligned}
$$

Now, we need semantic rules for procedures. Let the metavariables A, P, and D range over argument lists, parameter lists, and declaration lists, respectively. We'll start with the procedure declaration itself, which adds it to σ:

$$\textsc{ProcDecl}\ \frac{Q = \text{procedure } x(A); D \text{ begin } L \text{ end} \qquad \sigma' = \sigma[x \mapsto Q]}{\langle \sigma, Q \rangle \to \langle \sigma', \text{skip} \rangle}$$

Now, we need procedure calls. Procedure declaration lists form environments—that is, each procedure can see its own declared variables, and can see surrounding variables, but cannot see or interfere with other procedure calls' variables—so we need some way of distinguishing the variables within a procedure call from the variables outside of it. The Simple Imperative Language's version of σ does not form a tree, just a single map, and there's no easy way to make it serve both roles. Our solution to this will be substitution: when we call a procedure, we create fresh new variable names for all of the variables it defines, and substitute the variables in the procedure body for their new names. In this way, the procedure's variables are isolated from other variables.

We will not formally define the function to generate new variable names for all variables in a procedure. Suffice it to say, for each variable x in the parameter list or declaration list of a procedure Q, $freshen(Q)$ will create a fresh variable name x' and substitution $[x'/x]$. $freshen(Q)$ returns this substitution list. Since we now have substitutions, we will let the metavariable S range over substitutions and substitution lists.

Now, we have the necessary framework to define the semantics of a procedure call. In addition, we need to resolve each argument of a procedure call, and we can do that as well:

$$\textsc{CallArg}\ \frac{\langle \sigma, M_1 \rangle \to \langle \sigma, M_1' \rangle}{\langle \sigma, x(N_1, N_2, \cdots, N_n, M_1, M_2, \cdots, M_m) \rangle \to \langle \sigma, x(N_1, N_2, \cdots, N_n, M_1', M_2, \cdots, M_m) \rangle}$$

$$\textsc{ProcCall}\ \frac{\sigma(x_1) = Q = \text{procedure } x_2(x_{a,1}, x_{a,2}, \cdots, x_{a,n}); D \text{ begin } L \text{ end} \qquad S = freshen(Q)}{\langle \sigma, x_1(N_1, N_2, \cdots, N_n) \rangle \to \langle \sigma, \text{begin } (x_{a,1}\ S) := N_1; (x_{a,2}\ S) := N_2; \cdots; (x_{a,n}\ S) := N_n; (L\ S) \text{ end} \rangle}$$

Let's take these one at a time.

CallArg says that if you have a call with arguments $N_1, N_2, \cdots, N_n, M_1, M_2, \cdots, M_m$—that is, the first n arguments have been reduced, then the $(n+1)$th argument can be reduced.

ProcCall is the actual call. It replaces a call $(x_1(N_1, N_2, \cdots, N_n))$ with a `begin-end` block. That block starts with n assignment commands to each corresponding parameter $(x_{a,1} \cdots x_{a,n})$, then has the procedure's body (L) with its variables freshened. From this point, each statement in the procedure will be executed, and since the declared variable names have been replaced, it effectively has its own scope.

To demonstrate, let's follow through the reduction of our SIL-P program to with the `fac` procedure, but to keep it reasonable, only to `fac(2)`. There are many correct ways to implement $freshen$, so we will assume that variables are suffixed with a counter. The reduction steps are shown in Figure 2. Since we freshened our variables, the recursion can simply be flattened into a sequence of statements. Note that the `n1` and `n2` variables are from separate, recursive calls to `fac`, but both are present in some program states. `x`, on the other hand, since it was *not* defined within the procedure, refers to the same (global) `x`.

Another aspect of this definition worth noting is that it pollutes σ with an only-increasing number of entries; σ never shrinks. If we implemented this directly on a real computer, we could easily chew through all of memory! Real implementations need techniques to clear out the store: for the stack, popping stack frames, and for the heap, either explicit memory management (`malloc`/`free`) or implicit memory management (garbage collection). Luckily, we operate in the world of mathematical logic, so our response to this pollution is ¯_(ツ)_/¯. σ will simply get polluted, and we don't care.

5.1 Procedures and Types

Procedures complicate types for two reasons: first, there is the question of whether our procedures should be first-class values, and second, the ordering of statements becomes more complex with procedures. We will of course need a procedure type as well.

In most procedural languages, procedures are not first-class. SIL-P is no exception. By our semantics, procedures cannot be assigned to variables or arguments, because our semantics get stuck if they reduce to anything but an integer (remember, N is a metavariable over integers). This means that any time we call a procedure, we know with certainty what procedure we're calling[3].

Second, consider the ordering of statements, in particular with respect to T_Assign and T_Reassign. If a procedure assigns to a global variable, but is defined before the first definition of that global variable, we need to assure that its affect on that global is the same as the global definition. This is back to the initialization problem: with unpredictable ordering, it's unclear who's in charge of initialization. The simple solution to this, and the solution that Pascal uses, is explicit type declarations. If we had bothered to define our global variables, it would be trivial to assure that they're used correctly.

[3]Technically, our semantics as defined allow us to re-define or conditionally define procedures, so one can write a program in which it's non-obvious which procedure is being called, but with types, we would probably introduce a rule similar to T_Reassign to prevent this.

$\langle \{\}$, `procedure fac(n); begin if n = 1 then x := 1 else begin fac(n+-1); x := x * n end end; fac(2)` $\rangle$

(ProcDecl) $\rightarrow$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...` $\}$, `skip; fac(2)` $\rangle$

(Skip) $\rightarrow$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...` $\}$, `fac(2)` $\rangle$

(ProcCall) $\rightarrow$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...` $\}$,
`begin n1 := 2; if n1 = 1 then x := 1 else begin fac(n1+-1); x := x * n1 end end` $\rangle$

(BlockOnly) $\rightarrow$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...` $\}$, `n1 := 2; if n1 = 1 then x := 1 else begin fac(n1+-1); x := x * n1 end` $\rangle$

(Assign, Skip) $\rightarrow^*$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2` $\}$, `if n1 = 1 then x := 1 else begin fac(n1+-1); x := x * n1 end` $\rangle$

(Var, EqFalse) $\rightarrow^*$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2` $\}$, `if false then x := 1 else begin fac(n1+-1); x := x * n1 end` $\rangle$

(IfFalse) $\rightarrow$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2` $\}$, `begin fac(n1+-1); x := x * n1 end` $\rangle$

(BlockOnly) $\rightarrow$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2` $\}$, `fac(n1+-1); x := x * n1` $\rangle$

(Var, Neg, Add) $\rightarrow^*$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2` $\}$, `fac(1); x := x * n1` $\rangle$

(ProcCall) $\rightarrow$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2` $\}$,
`begin n2 := 1; if n2 = 1 then x := 1 else begin fac(n2+-1); x := x * n2 end end; x := x * n1` $\rangle$

(BlockRest) $\rightarrow$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2` $\}$,
`n2 := 1; if n2 = 1 then x := 1 else begin fac(n2+-1); x := x * n2 end; x := x * n1` $\rangle$

(Assign, Skip) $\rightarrow^*$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2`, `n2` $\mapsto$ `1` $\}$,
`if n2 = 1 then x := 1 else begin fac(n2+-1); x := x * n2 end; x := x * n1` $\rangle$

(Var, EqTrue) $\rightarrow^*$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2`, `n2` $\mapsto$ `1` $\}$,
`if true then x := 1 else begin fac(n2+-1); x := x * n2 end; x := x * n1` $\rangle$

(IfTrue) $\rightarrow$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2`, `n2` $\mapsto$ `1` $\}$, `x := 1; x := x * n1` $\rangle$

(Assign, Skip) $\rightarrow^*$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2`, `n2` $\mapsto$ `1`, `x` $\mapsto$ `1` $\}$, `x := x * n1` $\rangle$

(Var, Var, Mul, Assign) $\rightarrow^*$

$\langle \{$`fac` $\mapsto$ `procedure fac(n)...`, `n1` $\mapsto$ `2`, `n2` $\mapsto$ `1`, `x` $\mapsto$ `2` $\}$, `skip` $\rangle$

Figure 2: Reduction steps for `procedure fac(n); begin if n = 1 then x := 1 else begin fac(n+-1); x := x * n end end; fac(2)`

For a procedure type, we need a constructed type defined by the argument types to the procedure, e.g.:

$$\begin{aligned}
\langle type \rangle &::= \cdots \mid \text{procedure}(\langle typelist \rangle) \\
\langle typelist \rangle &::= \epsilon \\
&\mid \langle type \rangle \, \langle typelistrest \rangle \\
\langle typelistrest \rangle &::= \epsilon \\
&\mid , \langle type \rangle \, \langle typelistrest \rangle
\end{aligned}$$

Note that unlike functions in functional languages, procedures may have zero arguments, and have no return type. Of course, procedures with returns are common in procedural languages (e.g. `function`s in Pascal), so a return type is also possible.

Some languages, such as C, define a `void` type for the return type from procedures which don't return values, but this complicates typing, since there are no values of type `void`. It also opens a whole range of bizarre behaviors. Consider, for instance, the following valid C snippet:

```
void a() {
    /* perform some task... */
}
void b() {
    return a();
}
```

The `b` function has a `return` statement in spite of not returning a value, and this is valid because `a()` is of type `void`, and so isn't a value. This adds confusion to the semantics of C, since a `void` function can have a `return` statement with an expression to return, but the step to evaluate that expression should not produce a value. The easiest way to avoid this problem is Pascal's solution: separate subroutines which do return values (`function`s) from subroutines which don't (`procedure`s).

Exercise 1. Write the type judgments for procedures. Consider how to deal with the initialization problem.

6 Arrays and References

Arrays are as fundamental to most imperative languages as lists are to most functional languages. An array is—and I apologize for pedantically defining this when you've undoubtedly been using arrays for years—a mapping from integer indices to values, in which the integer indices are all part of a continuous domain, typically either $(0, n)$ or $(1, n)$, where n is either one less than the size of the array or the size of the array, respectively. The decision of whether to start arrays from 0 or 1 has ended friendships, ruined lives, and sparked several small-scale wars [citation needed], but ultimately doesn't matter. Either definition works fine, and neither has reliably proved to be any easier to use than the other; programmers frequently make off-by-one errors in any languag.e

Arrays generally have a fixed size, and are usually implemented such that access to a field within an array is quite fast. Pascal has two kinds of arrays, static and dynamic arrays, but we will only focus on the latter, as the former can easily be rewritten in terms of the latter.

Arrays in Pascal are declared with a specific element type, e.g. `x : array of integer`. Arrays are thus a constructed type. SIL-P has no types, of course, so we will not need any such declaration.

Before using an array in Pascal, you must first allocate it. This is done with the built-in function `setLength`. For instance, to allocate space for 15 integers in `x` declared above, one calls `setLength(x, 15)`. You can then access elements of the array with square brackets, from 0 to the size of the array minus one (in this case, 15). For instance, the following (pointless) procedure allocates an array and fills it with the squares of its indices:

```
procedure foo();
    var x: array of integer;
    var i: integer;
begin
    setLength(x, 15);
    i := 0;
    while i < 15 do
        x[i] := i*i
end
```

Pascal does not perform any bounds checking, so if you attempt to access the array at an index below zero or greater than or equal to the length of the array, unpredictable behavior will occur. Quite precisely, it will write to memory at an address outside the range of the array, but since our formal semantics don't model memory, that's unpredictable to us; we will revisit this again in Module 10.

Consider a program to generate the Fibonacci sequence. The dynamic programming version of this algorithm involves carrying the last two Fibonacci numbers in variables, but if we're saving all of the Fibonacci numbers in an an array, then we always have the previous two available. If we put this in a procedure, we need to do one of three things:

- Allocate the array ourselves and return it.
- Take the array as an argument and fill it.
- Share an array in a variable in the global scope.

The first option creates the additional confusion of what it means to return an array; should we duplicate it, or return a reference? The second option creates the same confusion, with respect to arguments. The third option is impractical, so we will discard it. Ultimately, we would like to be able to pass an array to a procedure, have the procedure change it, and see those changes in the calling procedure. We already made a solution to this problem in functional languages: *references*. Arrays are a *reference type*.

A reference type is a type that is always referential; i.e., values of a referential type are always stored by way of a link to the heap (Σ), and all that is ever present in an expression or the store (σ) is a label. Thus, we need to reintroduce Σ to our reduction. In addition, as we previously used the metavariable N for numbers, but we now have another kind of terminal value, labels, at this point, we have to reinterpret the meaning of N: the metavariable N is now over *all* terminal values, including numbers and labels, and even procedures.

While `setLength` works well for Pascal (where its behavior requires both call-by-reference and stack-based memory management, which we won't be discussing), it is not the natural way to describe arrays in our semantics. Instead, we will invent a new syntax for allocating arrays, which is an expression: `array[`M`]`. Previously, we stratified our expressions into boolean and integer expressions, but with the introduction of arrays, we will need to broaden integer expressions to simply "expressions", and so we will rename $\langle intexp \rangle$ to $\langle exp \rangle$. Note that this still excludes boolean expressions (and, similarly, N still excludes boolean values).

We extend SIL-P to SIL-PA (Simple Imperative Language with Procedures and Arrays) with our new expressions for allocating and accessing arrays, plus a new statement for writing to them, and create a new syntax for arrays in the heap:

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \cdots \\
& | & \text{array}[\langle exp \rangle] \\
& | & \langle exp \rangle\,[\langle exp \rangle] \\
\langle stmt \rangle & ::= & \cdots \mid \langle var \rangle\,[\langle exp \rangle] := \langle exp \rangle \\
\langle array \rangle & ::= & [] \\
& | & [\langle arglist \rangle]
\end{array}
$$

Note that $\langle array \rangle$ is not referred to by any other production. We will store arrays in our heap, but you cannot write an array literal in SIL-PA; it is not part of the language's syntax. Also, the array syntax we've described technically lets arbitrary expressions be part of an array, but in practice, they will always be reduced to values. Additionally, we need labels for our reductions, but as with labels previously, they will not have any defined syntax.

Labels just need to be unique. Labels are values, but arrays are not, since no expression can evaluate to an array anyway, only to a label that references an array.

Now, let's add semantics. Because we've reintroduced a heap, we will need to expand what we're reducing over to a triple again, $\langle \Sigma, \sigma, x \rangle$. You may assume that all previously defined reductions don't touch the the heap. Let ℓ range over labels, M range over expressions, and N range over values. All other metavariables will have the same range as they previously had.

$$\textsc{AllocStep}\ \frac{\langle \Sigma, \sigma, M \rangle \to \langle \Sigma', \sigma, M' \rangle}{\langle \Sigma, \sigma, \text{array}[M] \rangle \to \langle \Sigma', \sigma, \text{array}[M'] \rangle}$$

$$\textsc{Alloc}\ \frac{\ell \text{ is a fresh label} \qquad N \in \mathbb{N} \qquad \Sigma' = \Sigma[\ell \mapsto [0_1, 0_2, \cdots, 0_N]]}{\langle \Sigma, \sigma, \text{array}[N] \rangle \to \langle \Sigma', \sigma, \ell \rangle}$$

$$\textsc{IndexLeft}\ \frac{\langle \Sigma, \sigma, M_1 \rangle \to \langle \Sigma', \sigma, M_1' \rangle}{\langle \Sigma, \sigma, M_1[M_2] \rangle \to \langle \Sigma', \sigma, M_1'[M_2] \rangle} \qquad \textsc{IndexRight}\ \frac{\langle \Sigma, \sigma, M \rangle \to \langle \Sigma', \sigma, M' \rangle}{\langle \Sigma, \sigma, N[M] \rangle \to \langle \Sigma', \sigma, N[M'] \rangle}$$

$$\textsc{Index}\ \frac{\Sigma(\ell) = [N_{a,0}, N_{a,1}, \cdots, N_{a,N}, \cdots, N_{a,n}]}{\langle \Sigma, \sigma, \ell[N] \rangle \to \langle \Sigma, \sigma, N_{a,N} \rangle}$$

$$\textsc{ArrAssignLeft}\ \frac{\langle \Sigma, \sigma, M_1 \rangle \to \langle \Sigma', \sigma, M_1' \rangle}{\langle \Sigma, \sigma, x[M_1] := M_2 \rangle \to \langle \Sigma', \sigma, x[M_1'] := M_2 \rangle}$$

$$\textsc{ArrAssignRight}\ \frac{\langle \Sigma, \sigma, M \rangle \to \langle \Sigma', \sigma, M' \rangle}{\langle \Sigma, \sigma, x[N] := M \rangle \to \langle \Sigma', \sigma, x[N] := M' \rangle}$$

$$\textsc{ArrAssign}\ \frac{\begin{array}{ll}\sigma(x) = \ell & \Sigma(\ell) = [N_{a,0}, N_{a,1}, \cdots, N_{a,v-1}, N_{a,v}, N_{a,v+1} \cdots, N_{a,n}] \\ v = N_1 & \Sigma' = \Sigma[\ell \mapsto [N_{a,0}, N_{a,1}, \cdots, N_{a,v-1}, N_2, N_{a,v+1}, \cdots, N_{a,n}]]\end{array}}{\langle \Sigma, \sigma, x[N_1] := N_2 \rangle \to \langle \Sigma', \sigma, \text{skip} \rangle}$$

The AllocStep, IndexLeft, IndexRight, ArrAssignLeft, and ArrAssignRight rules simply reduce a subexpression.

Alloc defines the allocation of an array. Note that an array is allocated on the heap, so Alloc reduces to a label. In our definition, the array starts filled with 0s, hence $0_1, 0_2, \cdots, 0_N$. This label can then be used to access the array with Index. Index requires a label ℓ as its target, and that $\Sigma(\ell)$ maps to an array, and, implicitly, that its index, N, is a number, and that the array has at least N elements. The elements are labeled $N_{a,0}$ through $N_{a,n}$, and so we extract the Nth element, reducing to $N_{a,N}$. Our semantics will get stuck if we try to access an element outside the bounds of the array, or if we try to index an array with something other than an integer.

ArrAssign is the most sophisticated reduction in this set, and perhaps the most sophisticated reduction we've seen in this course. Let's take it one condition at a time:

- To match the left-hand side of $\to$, it must be an assignment to a variable x indexed by a value N_1. ArrAssignLeft assures that N_1 will be a value (or that the reduction will get stuck before reaching this point).
- $\sigma(x) = \ell$ specifies that the variable x must be in the store, and furthermore, that x must refer to a label ℓ.
- $v = N_1$ simply renames N_1 as v, since otherwise it will be difficult to read in the next step.
- $\Sigma(\ell) = [\cdots]$ specifies that ℓ must be in the heap, that it must reference an array, and that that array must have an element v ($N_{a,v}$).

- $\Sigma' = \Sigma[\ell \mapsto [\cdots]]$ defines a new array value in a new heap Σ', identical to Σ except that ℓ has been remapped to a new array, which is in turn identical to the original array except that $N_{a,v}$ has been replaced by the value we were assigning, N_2. N_2 is guaranteed to be a value by ArrAssignRight.

Note that we've modified Σ, but not σ. As a consequence, if we reassign a different variable—or an argument to a function—to refer to the same array, changes made to the array will be visible in both, because they both share the same label. This definition of array assignment demands that the target be a variable, but most languages allow an expression, so long as that expression evaluates to a label. Our semantics for assignment will get stuck in the same situations that our semantics for indexing would get stuck. Note that nothing in our semantics has required us to store integers in our array—SIL-PA is a dynamically typed language—so we can store nested arrays, or even, if we're feeling perverse, procedures in our array.

Now, let's return to our example. We have the infrastructure in SIL-PA required to make a procedure which generates the Fibonacci sequence into an array. We will define a procedure `fibarr` which is called with an array and a number, where the number specifies the length of the Fibonacci sequence to generate into the array.

```
procedure fibarr(arr, ct);
idx;
begin
    idx := 0;
    while idx < ct do
        begin
        if idx = 0 then
            arr[0] := 1
        else if idx = 1 then
            arr[1] := 1
        else
            arr[idx] := arr[idx + -1] + arr[idx + -2];
        idx := idx + 1
        end
end
```

Exercise 2. Work out the values of σ and Σ every time they change in the evaluation of the statement `fibarr(array[4], 4)`.

Extending our type system to work with arrays would be complicated. Most type systems in languages with arrays abandon one aspect of type safety: out-of-bounds access gets stuck. Integrating array bounds into the type system has proved to be a continuing difficulty in language design, and it and similar problems spawned the area of *dependent type systems*, in which types can be defined by values (such as the size of an array). Dependent type systems are beyond the scope of this course, but are the right area to study if you find this problem interesting.

7 Records

An alternate to arrays for storing compound data is *records*. If we were to implement records in an extension to the Simple Imperative Language, we would simply take them as syntactic sugar for arrays—they are no more powerful, just more usable—so we will not discuss their semantics, just their form in Pascal.

Record types, familiar to C programmers as `struct`s, define their own stores, mapping a specific set of names to values. These name-value pairs are called *fields*, and every value of a given record type has the same field names, but not (necessarily) the same field values. The order of the names theoretically shouldn't matter for the type, but is usually important for how the record is stored. If we were to convert records into arrays, for example, then the first field would become index 0, and the second index 1, etc.

Record types are constructed types, so we need a syntax for constructing them. In Pascal, a record with two fields, `x` and `y`, both of type `integer`, is written as follows:

```
record
    x: integer;
    y: integer;
end
```

It gets cumbersome to write such a long type everywhere you need it, so Pascal allows you to write *type aliases*, which are declarations that give abbreviations to types. We can give this record the name `point` as follows:

```
type point = record
    x: integer;
    y: integer;
end
```

We can then write `point` in place of the long record type.

In a language with types, records are particularly important because arrays are always composed of elements of a single type, while records may be of multiple types. For instance, we can associate an array of integers with a single integer like so:

```
record
    samples: array of integer;
    median: integer;
end
```

In Pascal, declaring a variable with a record type is sufficient to make space for it. In fact, like in C, this allocates it for the particular subroutine (on the stack), but the record becomes unusable as soon as the subroutine ends. So, we can define a subroutine with several `point`s like so:

```
function foo(bx, by, ex, ey: integer);
    var b: point;
    var e: point;
begin
    ...
end
```

Accessing fields of records is similar to accessing elements of arrays, but instead of brackets, a dot followed by a name is used. This name is not a variable in the surrounding scope, but the name of the field, so the field name must be written explicitly into the code. You cannot access an arbitrary field named by an expression, only a specific field. Let's finish our `function` above to one that calculates the Manhattan distance between two points, rather pointlessly adding them to records to do so:

```
function manhattan(bx, by, ex, ey: integer);
    var b: point;
    var e: point;
begin
    b.x := bx;
    b.y := by;
    e.x := ex;
    e.y := ey;
    manhattan := (e.x-b.x) + (e.y-b.y)
end
```

Records don't actually add any power to our language: everything records can do, arrays can (awkwardly) do as well. But, this grouping of values by named fields became a fundamental building block for *object-oriented programming*, which is the next module.

8 Fin

In the next module, we will look at arguably the most popular and successful programming paradigm in existence: object-oriented programming. Assignment 5 will focus on implementing imperative programming like you saw in this module.

References

[1] Edsger W Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[2] Niklaus Wirth. The programming language Pascal. *Acta informatica*, 1(1):35–63, 1971.

Rights