All the languages that we have studied so far have been purely functional. That is, the evaluation of expressions in these languages is side-effect free and (recursive) function call is the main computational device that drives the evaluation. Functional programming is contrasted by imperative programming where the main computational device is state mutation. We have argued that the functional programming paradigm has certain advantages over the imperative paradigm. In particular, the absence of side-effects makes it much easier to reason about the behavior of a program. Nevertheless, imperative programming is important in practice, specifically for performance critical code, as imperative language primitives map more directly to the underlying hardware architecture that ultimately executes a program. In this class, we will extend our simple language with the two central primitives of imperative programming languages: mutable variables and assignments. That is, we introduce state and mutation.

Variables and Assignments in JavaScript

Before we formalize our language with mutable variables and assignments, we analyze several examples to better understand how these primitives work in a language such as JavaScript.

In JavaScript mutable variables are declared by var declarations, which are like const declarations except that the keyword const is replaced by var:

\[ \text{var } x = e_d; e_b \]

The difference to a const declaration is that the binding of \( x \) to the value obtained from \( e_d \) can be modified in the body \( e_b \) using assignments

\[ x = e \]

For example, the following program declares a mutable variable \( x \) which is initialized to the value 1 and subsequently modified to \( x + 1 \):

\[
\begin{align*}
\text{var } x &= 1; \\
x &= x + 1; \\
x
\end{align*}
\]
This program evaluates to 2 since the final occurrence of $x$ evaluates to the new value that $x$ is updated to by the assignment.

As in most imperative programming languages, assignments are expressions that evaluate to a value—the value obtained from the right side of the assignment. For example, the following code also evaluates to 2.

```javascript
var x = 1;
x = x + 1
```

In particular, assignments can be nested inside other assignments. For example, the following program nests assignments to $x$ and $y$ by first assigning $x$ to the value 3 and then subsequently assigning the same value to $y$

```javascript
var x = 2;
var y = 2;
y = x = 3;
x + y
```

This program thus evaluates to 6.

At first, it seems that we can eliminate `var` declarations by replacing them with `const` declarations and also replacing every assignment to a variable $x$ by a new `const` declaration that redeclares $x$ with a new value, shadowing the previous binding of $x$. For example, we can rewrite our first program as follows:

```javascript
const x = 1;
{ const x = x + 1;
x }
```

The additional curly braces are needed because the scope of a `const` declaration in JavaScript is the entire basic block of statements in which the declaration occurs. By wrapping the second declaration in curly braces we start a new basic block. This program still evaluates to 2.

Unfortunately, the proposed elimination technique for `var` declarations only works for a straight-line sequence of declarations and assignments. If assignments are nested within function bodies, then the result of evaluation changes if we replace assignments by `const` declarations. For example, consider the following program

```javascript
var x = 2;
const f = function (y) { x = y; return x }; f(3);
x
```

This program evaluates to 3. This is because when $f$ is called, the nested assignment mutates the variable $x$ to 3. This mutation is globally observable in the entire scope of $x$. Hence, when we access $x$ after the call returns, we obtain the new value 3.

On the other hand, consider the program
const x = 2;
const f = function (y) {
    const x = y;
    return x
};
f(3);
x
This program evaluates to 2 since the scope of x declared by the const declaration inside of f is restricted to the body of f. Thus, the occurrence of the variable x after the call to f refers to the declaration on the first line. Hence, we obtain the value 2.

The fact that variable assignments have globally observable side-effects is one of the reasons why imperative programs are more difficult to reason about.

A Simple Language with Variables and Assignments

We start from the language in class 15 and extend it with variable declarations, assignments, addresses, and a dereference operator. The grammar of the extended language is as follows:

\[
\begin{align*}
    n \in \text{Num} & \quad \text{numbers (double)} \\
b \in \text{Bool} & := \text{true} \mid \text{false} \quad \text{Booleans} \\
a \in \text{Addr} & := \mathbb{N} \quad \text{addresses} \\
x \in \text{Var} & \quad \text{variables} \\
\tau \in \text{Typ} & := \text{bool} \mid \text{number} \mid (x : \tau_1) \Rightarrow \tau_2 \quad \text{types} \\
v \in \text{Val} & := n \mid b \mid a \mid \text{function} p(x : \tau) t e \quad \text{values} \\
e \in \text{Expr} & := x \mid v \mid e_1 \text{bop} e_2 \mid uop e_1 \mid e_1 \text{ ? } e_2 : e_3 \mid \text{mut } x = e_d ; e_b \mid e_1 (e_2) \quad \text{expressions} \\
bop \in \text{Bop} & := + \mid * \mid \& \mid | | \mid = \quad \text{binary operators} \\
uop \in \text{Uop} & := * \quad \text{unary operators} \\
p & := x \mid \epsilon \quad \text{function names} \\
t & := \tau \mid \epsilon \quad \text{return type annotations} \\
\text{mut} \in \text{Mut} & := \text{const} \mid \text{var} \quad \text{mutabilities}
\end{align*}
\]

Note that we introduce variable declarations by generalizing const declarations in our previous language to a declaration that is parameterized by a mutability, mut \in \text{Mut}. A mutability mut is either \text{var} or \text{const}. We introduce addresses, a \in \text{Addr}, as a new type of values that denote locations in memory. We define \text{Addr} = \mathbb{N}. However, the specific definition of \text{Addr} is immaterial. We only rely on the fact that \text{Addr} is an infinite set (i.e., we never run out of addresses when we need fresh ones for memory allocation). The assignment operator = is a binary operator and the dereference operator * is a unary operator.

Addresses a and dereference expressions \text{ * } e \ are included in program expressions because they arise during evaluation. However, there is no way to

---

\[1\text{Even if we assumed dynamic scoping semantics, this program would still evaluate to 2.}\]
explicitly write these expressions in the source program (i.e., they are not part of the language’s concrete syntax). These primitives are an example of an enrichment of program expressions as an intermediate form solely for evaluation.

**Small-Step Semantics**

Next, we adapt our small-step operational semantics from class 16 to account for the addition of variables and assignments. We will see that adding state and mutation to our language forces us to do a rather global refactoring of our operational semantics.

**Modeling State**

We model the state of an expression using a mapping \( M \) that we refer to as the memory. The memory is both an input and output of the small step reduction relation. That is, the small-step judgment form is now as follows:

\[
\langle M, e \rangle \rightarrow \langle M', e' \rangle
\]

This judgment says informally, “In memory \( M \), expression \( e \) steps to a new configuration with memory \( M' \) and expression \( e' \)”. The presence of a memory \( M \) that gets updated during evaluation is the hallmark of imperative computation.

In our current language, we can only assign values to variables. If we think of this restriction in terms of computer systems architecture, this means that a memory \( M \) only needs to model the stack of activation records for function calls, which stores the values of all local variables that are currently in scope. It is therefore tempting to define \( M \) as a partial mapping from variable names to values, similar to our notion of environment that we used in the dynamic scoping semantics studied in class 11. However, we want to maintain a static scoping semantics in our new language. We therefore introduce one level of indirection to model memory access and mutation with static scoping correctly. In our model, a memory \( M \) is a partial function from addresses to values, \( M : \text{Addr} \rightarrow \text{Var} \). We can then think of a \texttt{var} variable \( x \) as a \texttt{const} variable that stores an address to a memory location in \( M \). Whenever we use a \texttt{var} variable \( x \) in an expression, we implicitly dereference the address \( a \) stored in \( x \) to retrieve the value \( M(a) \) at the associated memory location in \( M \).

**Inference Rules**

The small-step reduction relation is defined by the rules shown in Figures 1 and 2. As usual, the rules are syntax-driven. The rules for the new imperative language primitives are given in Figure 1. Observe the interplay between the rules \texttt{DoVarDecl}, \texttt{DoDeref}, and \texttt{DoAssignVar}. The rule \texttt{DoVarDecl} handles \texttt{var} declarations. It first allocates a fresh memory address \( a \) to store the value \( v_d \) obtained from the defining expression of the declaration. Allocation of a fresh address is modeled by nondeterministically choosing some address \( a \).
\[
\frac{\langle M, e_2 \rangle \rightarrow \langle M', e'_2 \rangle}{\langle M, * a = e_2 \rangle \rightarrow \langle M', * a = e'_2 \rangle} \quad \text{SearchAssign}_2
\]
\[
\frac{a \in \text{dom}(M)}{\langle M, * a = v \rangle \rightarrow \langle M[a \mapsto v], v \rangle} \quad \text{DoAssignVar}
\]
\[
\frac{a \in \text{dom}(M)}{\langle M, * a \rangle \rightarrow \langle M, M(a) \rangle} \quad \text{DoDeref}
\]
\[
\frac{a \notin \text{dom}(M)}{\langle M, \text{var} x = v_d \rightarrow v_b \rangle \rightarrow \langle M', e_b[\ast a/x] \rangle} \quad \text{DoVarDecl}
\]

Figure 1: Small-step operational semantics of imperative primitives

that satisfies the condition \( a \notin \text{dom}(M) \). Then, the memory state \( M \) is updated accordingly to obtain the new memory state \( M' = M[a \mapsto v_d] \). The new expression is the body \( e_b \) where all free occurrences of the declared variable \( x \) have been replaced by dereference expressions \( * a \). This ensures that the rules DoDeref and DoAssignVar look-up, respectively, modify the content of the memory location \( a \) that we associate with \( x \). This is in contrast to the DoCon- stDecl rule for \texttt{const} declarations, where occurrences of \( x \) in \( e_b \) are replaced directly by the defining value \( v_d \). Note that the do rule for assignments reduces an assignment expression to the value \( v \) on the right side of the assignment, as expected.

Note that in the rule DoAssignVar we cannot replace \( x \) in \( e_b \) directly by the address \( a \). We need the additional dereference operation to model a memory look-up, correctly. Since we defined addresses as values, the DoDeref operation would otherwise have to take a step on a value, which would be inconsistent with our convention that once evaluation yields a value, the step-wise reduction terminates. Also, we will be forced to treat addresses as values, once we extend our language with mutable objects.

The rules for the non-imperative constructs are given in Figure 2. The rules are essentially identical to those given in class 16, except that we now have to thread the memory state \( M \) through all reduction steps. There is one other important detail, though. After the DoVarDecl rule has been applied for the \texttt{var} declaration of a variable \( x \), all the left sides of assignments to \( x \) are of the form \( * a \). We then have to be careful that these expressions are not reduced to values by the rule DoDeref. Otherwise, the rule DoAssignVar cannot be applied to reduce the assignment. For this reason, we exclude the assignment operator \( = \) in the SearchBop_1 rule.

You might have noticed that in our operational semantics, the memory \( M \) only grows and never shrinks during the course of evaluation. Our semantics only ever allocates memory and never deallocates. This choice is fine in a mathematical model, but a production run-time system must somehow enable collecting garbage–allocated memory locations that are no longer used by the
Figure 2: Small-step operational semantics of non-imperative primitives. The only changes compared to class 16 are the threading of the memory and the exclusion of assignment from the SearchBop1 rule.
running program. Collecting garbage may be done manually by the programmer (as in C and C++) or automatically by a conservative garbage collector (as in JavaScript, Scala, Java, C#, and Python).

Type Checking

Finally, we adapt the typing relation from class 15 to account for the new language primitives. The interesting new case are assignment expressions $e_1 = e_2$. Specifically, the following expression should be considered well-typed:

```
var x = 3; x = 5
```

whereas the following expression should not be well-typed:

```
const x = 3; x = 5
```

That is, we should only allow assignment to variables that have actually been declared by a `var` declaration. In order to be able to distinguish the two cases above during typing, we have to provide additional information in the typing environment. Namely, in addition to the type of every free variable $x$ that occurs in the expression being typed, the typing environment must also record $x$’s mutability. We thus modify the signature of typing environments $\Gamma$ as follows:

$$\Gamma : \text{Var} \rightarrow \text{Mut} \times \text{Typ}$$

The inference rules that define the new typing relation $\Gamma \vdash e : \tau$ are given in Figures 3 and 4. The rules in Figure 3 are identical to the corresponding rules discussed in class 15 except that the signature of $\Gamma$ has changed. However, this change is irrelevant for these rules. The new, respectively, modified rules are all summarized in Figure 4. All variable binding constructs (i.e., declarations and function abstractions) now also store the mutability of the declared name in the typing environment, together with the actual type inferred or annotated in the declaration. Note that function parameters and the names of recursive functions are considered to have `const` mutability. That is, function parameters cannot be reassigned in the function body. This is in contrast to JavaScript, where function parameters have `var` mutability. We will extend our language with the ability to reassign function parameters when we discuss parameter passing modes in class 20.

The only rule that uses the mutability information in the typing environment is the rule for assignments, `TypeAssignVar`. This rule ensures that the left-hand side of an assignment is always a variable and that this variable has indeed been introduced using a `var` declaration, rather than a `const` declaration or a function abstraction. Further note that the rule also checks that the two sides of an assignment agree on the type $\tau$. That is, we do not allow a variable $x$ to be reassigned to a value whose type is different from the type of $x$’s initialization expression. This is in contrast to JavaScript, which allows such reassignments since JavaScript is dynamically typed. The restriction to
Figure 3: Type checking rules for non-imperative primitives (no changes compared to class 15)

type-consistent reassignment in the rule TypeAssignVar is crucial for proving the preservation property of the new static typing relation.

There are no rules for typing addresses \( a \in \text{Addr} \) and dereference operations \( * \ e \) since these expressions are only introduced during evaluation.
\[
\Gamma \vdash e_d : \tau_d \quad \Gamma' = \Gamma\{x \mapsto (\text{mut}, \tau_d)\} \quad \Gamma' \vdash e_b : \tau_b
\]

\[
\Gamma \vdash \text{mut} \ x = e_d ; e_b : \tau_b
\]

\[
x \in \text{dom}(\Gamma) \quad \Gamma(x) = (\text{mut}, \tau) \quad \Gamma \vdash x : \tau
\]

\[
\Gamma(x) = (\text{var}, \tau) \quad \Gamma \vdash e : \tau
\]

\[
\Gamma' = \Gamma\{x \mapsto (\text{var}, \tau)\} \quad \Gamma' \vdash e : \tau'
\]

\[
\Gamma' = \Gamma\{x \mapsto (\text{const}, \tau)\} \quad \Gamma' \vdash e : \tau'
\]

\[
\Gamma' = \Gamma\{x \mapsto (\text{const}, \tau)\} \quad \Gamma' \vdash e : \tau'
\]

\[
\Gamma' = \Gamma\{x \mapsto (\text{const}, \tau)\}[x_2 \mapsto (\text{const}, \tau_2)]\quad \Gamma' \vdash e : \tau' \quad \tau_1 = (x_2 : \tau_2) \Rightarrow \tau' \quad \Gamma \vdash \text{function} \ x_1 (x_2 : \tau_2) : \tau' \quad e : \tau_1
\]

Figure 4: Type checking rules for imperative primitives