Until now, we could look at a program and determine from the syntax where a variable is bound. This is possible because λ-calculus and FL as previously presented use *static scoping* (also known as *lexical scoping*). This is not, however, the only possible scoping discipline.

The *scope* of a variable is where that variable can be mentioned and used. With static scoping, the places a variable can be used are determined by the lexical structure of the program. An alternative is *dynamic scoping*, where a variable is bound to the most recent (in time) value assigned to that variable.

The difference becomes apparent when a function is applied. In static scoping, free variables in the function body are evaluated based on the context in which the function was defined. In dynamic scoping, they are evaluated in the context of the function call. The difference is illustrated by the following program:

$$\begin{array}{l}\mathsf{let}\ d = 2\ \mathsf{in}\\ \mathsf{let}\ f = \lambda x.\, x + d\ \mathsf{in}\\ \mathsf{let}\ d = 1\ \mathsf{in}\\ \quad f\ d\end{array}$$

With lexical scoping (what we have seen to this point), the block above evaluates to 3.

1. The outer d is bound to 2.
2. The f is bound to $\lambda x.\, x + d$. Since d is bound statically, it is "locked" to the d that is in scope at the point of this definition, so f will always be equal to $\lambda x.\, x + 2$. (The value of d cannot change since there is no variable assignment/mutation in this language.)
3. The inner d is bound to 1. This does not change the outer binding, it simply overrides it within the scope of the inner-most let.
4. When evaluating the expression $f\ d$, the value of d is determined based on the current environment—so it will be 1—but the free variables in the body of f are evaluated using the environment in which f was defined—where $d = 2$. We will therefore get $(\lambda x.\, x + 2)\ 1$, which evaluates to 3.

With dynamic scoping, however, the block evaluates to 2.

1. The outer d is bound to 2.
2. The f is bound to $\lambda x.\, x + d$. The free variable d is not locked to any particular binding.
3. The inner d is bound to 1.
4. When evaluating the expression $f\ d$, the variable d is evaluated in the current environment in both the argument position *and the function body*. Since d is bound to 1 in the current environment, this will produce $(\lambda x.\, x + 1)\ 1$, which evaluates to 2.

Dynamically scoped languages are quite common and include many interpreted scripting languages. Examples of languages with dynamic scoping are (in roughly chronological order): early versions of LISP, APL, PostScript, TEX, and Perl. Early versions of Python also had dynamic scoping, but it was later changed to static scoping.

There are advantages and disadvantages to both disciplines. Some advantages of dynamic scoping include:

- It is easier to implement interpreters for languages with dynamic scope.
- Dynamic scope allows developers to extend almost any piece of code by overriding the values of variables that are used internally by that code.

Some advantages of static scope include:

- It is much easier to keep code modular. With dynamic scope, the true interface of any block of code becomes the entire set of variables used by that block. With static scope, internal implementation details can be kept hidden from anyone using the code.
- A compiler can determine where the variable will be located. As a result, it can optimize variable accesses into simple memory lookups or array accesses rather than needing expensive run-time lookup mechanisms that would be required with dynamic scope.

Most modern languages have opted for static scope. The modularity advantage makes it far easier for developers to reason about and analyze, and the performance advantages are a nice add-on.

1 Scope and the Interpretation of Free Variables

To see how to formalize these notions of scope and how they impact free variables, we will translate pure CBV λ-calculus into FL in two ways, $\mathcal{S}[\![\cdot]\!]$ will implement static scope, and $\mathcal{D}[\![\cdot]\!]$ will implement dynamic scope.

Scoping rules are all about how to evaluate free variables in a program fragment. With static scope, free variables of a term $\lambda x.\, e$ are interpreted according to the syntactic context in which the term $\lambda x.\, e$ occurs. With dynamic scope, free variables of $\lambda x.\, e$ are interpreted according to the environment in effect when $\lambda x.\, e$ is applied. These are not the same in general.

Both translations will use an environment to capture the interpretation of names. An *environment* is simply a partial function with finite domain from variables x to values.

$$\rho : \mathbf{Var} \rightharpoonup \mathbf{Val}$$

As in previous lectures, we can extend or modify ρ with the rebinding operator $\rho[x \mapsto v]$ defined by

$$\rho[x \mapsto v](y) \triangleq \begin{cases} v & \text{if } x = y, \\ \rho(y) & \text{if } x \neq y. \end{cases}$$

We also need a way to represent this environment in the target language, here FL. To do this, we will encode variables in the environment into values of the target language. We will write $\ulcorner x \urcorner$ to denote the encoding of variable x into FL. The exact details of the encoding don't really matter as long as we can look up the value of a variable given its encoding and update an environment with a new or modified binding. For instance, any structure that allows us to encode variables with different names differently and check equality of encoded variable names would work. Numbers are generally a good choice, but they are not the only one.

Given that we really just need operations to look up and update the environment, if R is a representation of environment ρ, we demand two things:

1. $\mathsf{lookup}\ R\ \ulcorner x \urcorner = \begin{cases} \rho(x) & \text{if } x \in \mathrm{dom}(\rho) \\ \mathsf{error} & \text{if } x \notin \mathrm{dom}(\rho) \end{cases}$
2. $\mathsf{update}\ R\ \ulcorner x \urcorner\ v$ is a representation of $\rho[x \mapsto v]$

To simplify notation, despite a slight risk of confusion, we will use ρ to represent both the environment and the representation of that environment in FL, so we will write $\mathsf{lookup}\ \rho\ \ulcorner x \urcorner$ and $\mathsf{update}\ \rho\ \ulcorner x \urcorner\ v$.

With this in hand, we can construct our translations. Note that the translation of a λ-term e will take the representation of an environment ρ and produce an expression in FL. In other words,

$$[\![e]\!] : \mathbf{Env} \to \mathbf{Exp}_{\mathrm{FL}}.$$

1.1 Static Scoping

The translation for static scoping is as follows.

$$\begin{aligned}
\mathcal{S}[\![x]\!]\,\rho &\triangleq \mathsf{lookup}\ \rho\ \ulcorner x \urcorner \\
\mathcal{S}[\![e_1\ e_2]\!]\,\rho &\triangleq (\mathcal{S}[\![e_1]\!]\,\rho)\ (\mathcal{S}[\![e_2]\!]\,\rho) \\
\mathcal{S}[\![\lambda x.\, e]\!]\,\rho &\triangleq \lambda v.\, \mathcal{S}[\![e]\!]\ (\mathsf{update}\ \rho\ \ulcorner x \urcorner\ v) \quad v \text{ fresh}
\end{aligned}$$

One important note about this translation is that it removes all of the variables in the source program e and replaces them with fresh names that are only used immediately after being bound. As a result, there is no room for the scoping discipline of the target language to impact the behavior of the translated program.

1.2 Dynamic Scoping

The translation for dynamic scoping is as follows.

$$\begin{aligned}
\mathcal{D}[\![x]\!]\,\rho &\triangleq \mathsf{lookup}\ \rho\ \ulcorner x \urcorner \\
\mathcal{D}[\![e_1\ e_2]\!]\,\rho &\triangleq (\mathcal{D}[\![e_1]\!]\,\rho)\ (\mathcal{D}[\![e_2]\!]\,\rho)\ \rho \\
\mathcal{D}[\![\lambda x.\, e]\!]\,\rho &\triangleq \lambda v\tau.\, \mathcal{D}[\![e]\!]\ (\mathsf{update}\ \tau\ \ulcorner x \urcorner\ v) \quad v, \tau \text{ fresh}
\end{aligned}$$

This translation is a bit more interesting. In the translation of λ-abstractions, we have completely discarded ρ and replaced it with a fresh parameter τ!

This choice has two ramifications. First, the translation no longer expects one argument v, it expects two, v and τ. That means that when applying a function, as in the translation of function application, we need to pass in a second argument, which is the environment at the time of application.

Second, the environment in which the λ-abstraction is defined is completely irrelevant to the translation, and thus to what happens when we apply the function. Instead, we use the environment passed in at application time to look up the value of encoded variables. Since function *application* passes in the environment present there, a single function may execute multiple times with multiple different environments, determined *dynamically* by where it is applied.

2 Correctness of Static Scoping Translation

While we do not already have a formal definition of dynamic scoping, we already have a semantics for CBV λ-calculus that we claimed implemented static scoping. The following theorem says that this belief is correct.

Theorem 1. *Given any λ-term e and any $\rho \in$ **Env** such that $\mathrm{FV}(e) \subseteq \mathrm{dom}(\rho)$, then the translation of e is $\beta\eta$-equivalent to substituting all values of ρ in for the free variables in e. That is,*

$$\mathcal{S}[\![e]\!]\,\rho \;=_{\beta\eta}\; e[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho)].$$

Proof. The proof follows by structural induction on e. The simple cases are variables and applications.

$$\mathcal{S}[\![x]\!]\,\rho \;=\; \mathsf{lookup}\ \rho\ \ulcorner x \urcorner = \rho(x) = x[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho)]$$

$$\begin{aligned}
\mathcal{S}[\![e_1\ e_2]\!]\,\rho &= (\mathcal{S}[\![e_1]\!]\,\rho)\ (\mathcal{S}[\![e_2]\!]\,\rho) \\
&=_{\beta\eta} (e_1[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho)])\ (e_2[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho)]) \\
&= (e_1\ e_2)[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho)]
\end{aligned}$$

For the λ-abstraction case $\lambda x.\, e$, for any value v, since $\mathsf{update}\ \rho\ \ulcorner x \urcorner\ v$ is a representation of $\rho[x \mapsto v]$, the induction hypothesis gives

$$\begin{aligned}
\mathcal{S}[\![e]\!]\ (\mathsf{update}\ \rho\ \ulcorner x \urcorner\ v) &=_{\beta\eta}\ e[y \mapsto \rho[x \mapsto v](y) \mid y \in \mathrm{dom}(\rho)] \\
&=\ e[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho) - \{x\}][x \mapsto v] \\
&=_{\beta}\ (\lambda x.\, e[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho) - \{x\}])\ v \\
&=\ ((\lambda x.\, e)[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho)])\ v.
\end{aligned}$$

Using this, we can move back to the full translation of $\lambda x.\, e$, giving

$$\begin{aligned}
[\![\lambda x.\, e]\!]\ \rho\ &=\ \lambda v.\, \mathcal{S}[\![e]\!]\ (\mathsf{update}\ \rho\ \ulcorner x \urcorner\ v) \\
&=_{\beta\eta}\ \lambda v.\, ((\lambda x.\, e)[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho)])\ v \\
&=_{\eta}\ (\lambda x.\, e)[y \mapsto \rho(y) \mid y \in \mathrm{dom}(\rho)]
\end{aligned}$$

□

The pairing of a function $\lambda x.\, e$ with an environment ρ in this way is called a *closure*. Theorem 1 says that we can implement $\mathcal{S}[\![\cdot]\!]$ by forming a closure consisting of the term e and an environment ρ that tells us how to interpret free variables in e. By contrast, in dynamic scoping, the translated function does not record the environment where it is defined, so there is no need for a closure.