Principles of Programming Languages

Hakjoo Oh

August 31, 2026

This draft book may contain errors.

Please let me know (hakjoo_oh@korea.ac.kr) if you find any errors or have suggestions.

Contents

Preface

This book summarizes the content of the "Programming Languages (COSE 212)" course taught to sophomore undergraduates at Korea University. Although the subject matter may be somewhat unfamiliar to students new to the field of programming languages, students have struggled due to the lack of a suitable textbook. I hope this book will be helpful to both the students enrolled in the course and others who wish to study programming languages.

Contents The goal is to understand the key principles of programming languages by designing and implementing them firsthand. Chapters 1 and 2 cover the fundamentals necessary for this. Starting with Chapter 3, we begin with a small programming language and gradually expand it. In Chapter 4, we extend the language to allow for the definition and use of functions and introduce concepts related to functions. In Chapter 5, lists and other features are added so that students can design and implement a more plausible functional programming language on their own. Chapters 6 and 7 explore ways to extend the language to

incorporate features of imperative languages. Chapter 8 allows students to experience the technique of equipping a programming language with a static type system to detect certain types of errors before program execution. Chapter 9 covers somewhat independent material, introducing lambda calculus, which is the origin of programming languages.

If you use this book in a course, it would be appropriate to cover the concepts of language design in class and assign students the task of implementing the language designed during the course. To this end, most chapters conclude with implementation problems suitable as programming assignments. By working through all of these problems yourself, you will acquire the fundamental skills needed to design and implement a program interpreter and analyzer. The content covered in this book is suitable for a full semester if the course combines lectures with hands-on lab work; if the course consists solely of lectures, it would cover approximately two-thirds of a semester.

References The following books were consulted while writing this book.

- Daniel P. Friedman and Mitchell Wand. Essentials of Programming Languages (3rd edition). MIT Press. 2008.
- Lee Kwang-geun. Stories of Programming Languages. 2006

- Benjamin C. Pierce. Types and Programming Languages. MIT Press. 2002.

In particular, the first two books have been used as textbooks for the COSE 212 Programming Languages course to date. I would like to note that this book was greatly influenced by their structure and content, and that I have adapted much of it. If you are looking for a deeper understanding of programming languages and type systems, I recommend reading the third book.

1

Induction

All programming languages are defined inductively. Therefore, while a programming language itself is finite, there is no limit to the number of programs that can be written in that language. In this chapter, we will learn about induction, which is the foundation of studying programming languages.

1.1 Inductive Definition of Sets

Induction is a method of defining a set using the set itself. For example, let's define the set S as the set of the smallest natural numbers that satisfy the following two conditions.

1. S contains 0 ($0 \in S$).
2. If S contains the natural number n, then S contains $n+3$ ($n \in S \implies n+3 \in S$).

This definition is called inductive because it uses elements that already belong to S to define the elements belonging to S.

What is the set S defined in this way? First, by the first condition, the set S must contain 0. If S contains 0, then by

the second condition, it must contain 3. Similarly, if S contains 3, it must contain 6. Repeating this process, we can see that the set S must contain all multiples of 3. However, a set that satisfies both conditions does not always contain only multiples of 3. For example, the set of all natural numbers, $\{0, 1, 2, 3, \ldots\}$, satisfies the two conditions above, and the set $\{0, 3, 6, 9, \ldots\} \cup \{1, 4, 7, 10, \ldots\}$ also satisfies them. In this way, there are infinitely many sets that satisfy both conditions. Therefore, in the definition above, S is defined as the "smallest set" that satisfies both conditions, and that set is the set of multiples of 3 ($\{0, 3, 6, 9, \ldots\}$). The set of multiples of 3 satisfies both conditions and is contained in any set that satisfies both conditions.

As in the example above, an inductive definition of a set consists of two types of rules. The first type consists of rules that specify the initial elements belonging to the set without any conditions ("S contains 0"). The second type consists of rules that define elements constructed using elements already belonging to the set ("If S contains a natural number n, then S contains $n + 3$"). In particular, the base elements must always be well-defined. This is because an inductive definition without a base element always implies the empty set ($\emptyset$).

Inference Rules

From now on, when defining sets inductively, we will frequently use inference rules. An inference rule looks like this:

$$\frac{A_1 \quad A_2 \quad \cdots \quad A_n}{B}$$

An inference rule consists of premises (A_1, A_2, ..., A_n) and a conclusion (B). If A_1, A_2, ..., A_n are all true, then B is also true. An inference rule that contains no assumptions is called an axiom, meaning that the conclusion is true without any conditions.

For example, if we define the set of multiples of 3, S, defined above, as an inference rule, it is as follows.

$$\frac{}{0 \in S} \qquad \frac{n \in S}{(n+3) \in S}$$

The left rule means that the proposition $0 \in S$ is true without any conditions. In other words, it means that the set S contains the natural number 0. The right-hand rule means that if the set S contains a natural number n, then S must also contain the natural number $n + 3$. The set defined by the inference rule is the smallest set that satisfies both conditions, but the condition "smallest set" is not explicitly stated.

The meaning becomes clearer if we interpret the inference rule as a proof rule as follows: We define the set S as the set of natural numbers n that can be used to prove the proposition $n \in S$ using the above inference rule. For example, S includes the natural number 6 because it can be used to prove proposition $6 \in S$ as follows.

$$\dfrac{\dfrac{\dfrac{}{0 \in S}}{3 \in S}}{6 \in S}$$

A proof is a process that begins with rules corresponding to axioms and involves repeatedly applying inference rules until the target of the proof is derived. This proof process is also called a proof tree. In the example above, the proposition $0 \in S$ was derived using the axioms, and then the right-hand rule was applied twice to derive $6 \in S$. Since $0 \in S, 3 \in S, 6 \in S$ and $9 \in S, \ldots$ are both provable, the set defined by the above inference rule includes all multiples of 3. On the other hand, since $1 \in S$ and $4 \in S$ cannot be proven, they do not belong to the set S. Therefore, the set S contains exactly the multiples of 3. When defining inference rules by interpreting them as proof rules in this way, there is no need to consider the condition of the "smallest set" separately; the rule naturally implies the smallest set among those that are closed under the rule.

For the sake of conciseness, when defining a set using an in-

ference rule, the set currently being defined is usually omitted. For example, the inference rule above can be written as follows:

$$\frac{}{0} \qquad \frac{n}{n+3}$$

In addition, inference rules are sometimes expressed using context-free grammar as follows:

$$n \to 0 \mid n+3$$

This means that if the set to be defined contains 0 as a basic element and includes n, then it must also include $n+3$. Again, the condition that it is the smallest set is omitted.

Examples of Inductive Definitions

Let's look at a few examples of sets defined inductively.

Example 1 The inference rule below also defines the set of multiples of 3, $\{0, 3, 6, 9, \ldots\}$.

$$\frac{}{0} \qquad \frac{}{3} \qquad \frac{x \quad y}{x+y}$$

The first and second rules mean that the set contains 0 and 3 as its basic elements. The third rule means that if x and y are

elements of the set, then $x + y$ is also an element.

Example 2 The set of strings with balanced parentheses

$$S = \{\texttt{()}, \texttt{()()}, \texttt{(())}, \texttt{()()()}, \texttt{(()())}, \texttt{()(())}, \texttt{(())()}, \texttt{((()))}, \ldots\}$$

can be defined by the following inference rules.

$$\frac{}{\texttt{()}} \qquad \frac{s}{\texttt{(}s\texttt{)}} \qquad \frac{s_1 \quad s_2}{s_1 s_2}$$

By the first rule, the string `()` belongs to the set S. By the second rule, if a string s belongs to the set S, then `(`s`)` must also belong to that set. According to the third rule, if the strings s_1 and s_2 belong to the set S, then $s_1 s_2$ must also belong to that set. For example, the string `(()())` is constructed as follows:

$$\dfrac{\dfrac{\dfrac{}{\texttt{()}}\ \text{(First Rule)}}{\texttt{()()}}\ \text{(Third Rule)}}{\texttt{(()())}}\ \text{(Second Rule)}$$

This indicates which inference rules were applied at each step of the proof.

Example 3 Let us define the set S as follows.

$$S = \{\texttt{a}^n\texttt{b}^{n+1} \mid n \in \mathbb{N}\}$$

Here, $\mathbb{N}$ denotes the set of all natural numbers, and x^n denotes a string defined as follows (ϵ is a string of length 0).

$$\begin{aligned} x^0 &= \epsilon \\ x^{i+1} &= xx^i \ (i \geq 0) \end{aligned}$$

Therefore, the set above corresponds to $S = \{\mathtt{b}, \mathtt{abb}, \mathtt{aabbb}, \mathtt{aaabbbb}, \dots\}$, and when defined as an inference rule, it is as follows.

$$\frac{}{\mathtt{b}} \qquad \frac{s}{\mathtt{a}s\mathtt{b}}$$

The first rule means that the string $\mathtt{b}$ belongs to the set S. The second rule states that if the set S contains the string s, then the string $\mathtt{a}s\mathtt{b}$ also belongs to that set.

Example 4 The set of all strings that can be formed using the set of English letters $\Sigma = \{\mathtt{a}, \dots, \mathtt{z}\}$ can be defined by the following inference rule:

$$\frac{}{\epsilon} \qquad \frac{s}{\mathtt{a}s} \qquad \frac{s}{\mathtt{b}s} \qquad \cdots \qquad \frac{s}{\mathtt{z}s}$$

By adding conditions to the right-hand side of the inference rule, multiple rules of the same form—excluding the first one—can be

concisely expressed as follows.

$$\frac{}{\epsilon} \qquad \frac{s}{xs}\ x \in \{\mathtt{a}, \ldots, \mathtt{z}\}$$

If the string s belongs to the set and x is any English letter, then the string xs also belongs to that set. The same set can be defined grammatically as follows.

$$\begin{array}{rcl} s & \rightarrow & \epsilon \\ & | & xs \quad (x \in \{\mathtt{a}, \ldots, \mathtt{z}\}) \end{array}$$

Example 5 The set of lists whose elements are integers can be defined as follows.

$$\frac{}{\mathsf{nil}} \qquad \frac{l}{n \cdot l}\ n \in \mathbb{Z}$$

The first rule means that the empty list (nil) is an integer list. The second rule states that if l is an integer list, then the list formed by adding an integer to the front of it ($n \cdot l$) is also an integer list. For example, $-7 \cdot 3 \cdot 14 \cdot \mathsf{nil}$ is a list generated by the above rule. This is because it can be generated by repeatedly

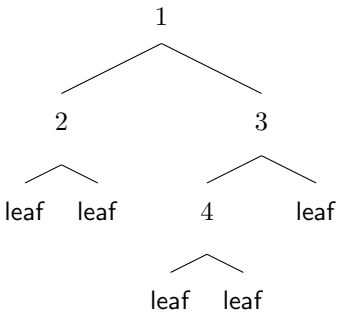
applying the inference rules, starting from the axioms as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{nil}}}{14 \cdot \mathsf{nil}}\ 14 \in \mathbb{Z}}{3 \cdot 14 \cdot \mathsf{nil}}\ 3 \in \mathbb{Z}}{-7 \cdot 3 \cdot 14 \cdot \mathsf{nil}}\ -7 \in \mathbb{Z}$$

The above list is defined grammatically as follows:

$$\begin{array}{rcl} l & \to & \mathsf{nil} \\ & | & n \cdot l \quad (n \in \mathbb{Z}) \end{array}$$

Example 6 Consider a binary tree that looks like this.



Since it is cumbersome to draw a tree structure every time, let's represent the binary tree above as follows.

$$(1, (2, \mathsf{leaf}, \mathsf{leaf}), (3, (4, \mathsf{leaf}, \mathsf{leaf}), \mathsf{leaf}))$$

From now on, we will frequently use this method of representing two-dimensional data as a one-dimensional sequence using parentheses. A set of binary trees that look like the one above can be defined by the following inductive rule.

$$\frac{}{\mathsf{leaf}} \qquad \frac{t_1 \quad t_2}{(n, t_1, t_2)}\ n \in \mathbb{Z}$$

This defines two ways to construct a binary tree.

1. The first rule refers to an empty binary tree containing only a single leaf node (leaf).

2. The second rule states that, given the binary trees t_1 and t_2, it is possible to create a new binary tree (n, t_1, t_2) by making the integer n the root node and using t_1 and t_2 as its left and right children, respectively, to form a new binary tree (n, t_1, t_2).

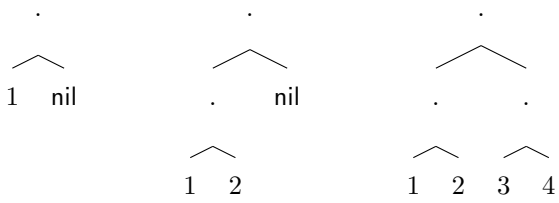
For example, according to the rule above,

$$(1, (2, \mathsf{leaf}, \mathsf{leaf}), (3, (4, \mathsf{leaf}, \mathsf{leaf}), \mathsf{leaf}))$$

is a binary tree. This can be proven using inference rules as follows:

$$\dfrac{\dfrac{\overline{\mathsf{leaf}}\quad\overline{\mathsf{leaf}}}{(2,\mathsf{leaf},\mathsf{leaf})}\ 2\in\mathbb{Z}\qquad\dfrac{\dfrac{\overline{\mathsf{leaf}}\quad\overline{\mathsf{leaf}}}{(4,\mathsf{leaf},\mathsf{leaf})}\ 4\in\mathbb{Z}\qquad\overline{\mathsf{leaf}}}{(3,(4,\mathsf{leaf},\mathsf{leaf}),\mathsf{leaf})}\ 3\in\mathbb{Z}}{(1,(2,\mathsf{leaf},\mathsf{leaf}),(3,(4,\mathsf{leaf},\mathsf{leaf}),\mathsf{leaf}))}\ 1\in\mathbb{Z}$$

Example 7 Let's define a binary tree in a different style. Consider a binary tree where data is stored in leaf nodes instead of internal nodes, as shown below.



Here, nil means that the branch has no child nodes. If we represent the trees above using parentheses in one dimension—such as 1, $(1,\mathsf{nil})$, $((1,2),\mathsf{nil})$, and $((1,2),(3,4))$, we can define the set of binary trees inductively as follows:

$$\overline{n}\ n\in\mathbb{Z}\qquad\dfrac{t}{(t,\mathsf{nil})}\qquad\dfrac{t}{(\mathsf{nil},t)}\qquad\dfrac{t_1\quad t_2}{(t_1,t_2)}$$

For example, $((1,2),(3,\mathsf{nil}))$ is constructed as follows:

$$\dfrac{\dfrac{\overline{}\ 1 \in \mathbb{Z} \quad \overline{}\ 2 \in \mathbb{Z}}{(1,2)} \qquad \dfrac{\overline{}\ 3 \in \mathbb{Z}}{(3,\mathsf{nil})}}{((1,2),(3,\mathsf{nil}))}$$

1.2 Inductive Definition of Programming Languages

A programming language is a set of programs constructed inductively. The rules that define the form of programs belonging to that language—that is, the rules for writing programs—are called the syntax of the programming language, while the rules that define the meaning of programs belonging to that language are called the semantics. Both are defined inductively.

Syntax

Consider the integer expression language, which is the set of integer expressions that can be formed using integers and the four arithmetic operators. For example, the following integer expressions are programs belonging to this language.

```
1,  1+2,  1+(2*(3-4)),  (1+2)/(3*(4-5)),  ...
```

(a) `1+(2*(3-4))` (b) `((1+2)*3)-4`

Figure 1.1: A program represented as a tree

As shown above, programs are usually written as one-dimensional strings, but in reality, computer programs have a two-dimensional tree-like structure. For example, the string

```
1+(2*(3-4))
```

represents the program shown in Figure 1.1a, and

```
((1+2)*3)-4
```

refers to the program shown in Figure 1.1b. For reference, the technique of converting a one-dimensional program written as a string into a two-dimensional tree structure is called syntax analysis or parsing. This book does not cover methods for analyzing grammatical structures; instead, it assumes that the structure of a program is clearly given in the form of a tree.

The grammatical structure of an integer expression language can be defined inductively using the following inference rules.

$$\overline{n}\ n \in \mathbb{Z} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ - } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ + } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ * } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ / } E_2}$$

It is more common to define the grammatical structure of a programming language using a grammar rather than inference rules. The grammatical structure of an integer expression language can be expressed as follows.

$$\begin{aligned} E &\rightarrow n \quad (n \in \mathbb{Z}) \\ &\mid E_1 \texttt{ + } E_2 \\ &\mid E_1 \texttt{ - } E_2 \\ &\mid E_1 \texttt{ * } E_2 \\ &\mid E_1 \texttt{ / } E_2 \end{aligned}$$

The rules above imply that there are five ways to form an integer expression. According to the first rule, any integer (n) can become an integer expression. The remaining rules mean that, given two integer expressions E_1 and E_2, a new integer expression can be formed by connecting them using the four arithmetic operators. Due to the nature of inductive rules, an infinite number of arbitrary integer expressions can be generated from these five finite rules.

For example, the integer expression `1+(2*(3-4))` is constructed as follows.

$$\dfrac{\dfrac{\dfrac{\overline{\texttt{3}}\quad\overline{\texttt{4}}}{\texttt{2}\quad\texttt{3 - 4}}}{\texttt{1}\quad\texttt{2 * (3 - 4)}}}{\texttt{1 + (2 * (3 - 4))}}$$

If we flip the proof tree upside down and emphasize its tree structure, we obtain Figure 1.1a. The grammatical structure of a programming language defined inductively in this way consists of rules that generate programs with a two-dimensional structure. To express this two-dimensional structure unambiguously in one dimension, we simply use parentheses appropriately, as in `1+(2*(3-4))`.

Semantic Structure

If the grammatical structure is the set of rules that defines the form of programs in a programming language, then the semantic structure is the set of rules that determines the execution semantics of a program.

While the meaning of a program can be defined in various ways, in the case of integer expressions, it is natural to define the meaning as the integer value obtained when the expression is evaluated. Let's express statements such as "Evaluating the integer expression E yields the integer n" or "The meaning of

the integer expression E is n" as follows.

$$E \Rightarrow n$$

For example, `1+(2*(3-4))` $\Rightarrow -1$ means that the program `1+(2*(3-4))` has the value -1. Here, $\Rightarrow$ represents a binary relation between programs and integers. That is, if we denote the set of programs as $\mathbb{P}$ and the set of integers as $\mathbb{Z}$, then $\Rightarrow$ is a subset of the Cartesian product $\mathbb{P} \times \mathbb{Z}$, and (i.e., $\Rightarrow \subseteq \mathbb{P} \times \mathbb{Z}$) Defining the semantic structure of integer expressions is equivalent to defining the set $\Rightarrow$. For integer expressions, the set $\Rightarrow$ can be defined using the following inference rule.

$$\frac{}{n \Rightarrow n} \quad \text{E-Num}$$

$$\frac{E_1 \Rightarrow n_1 \quad E_2 \Rightarrow n_2}{E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2} \quad \text{E-Plus}$$

$$\frac{E_1 \Rightarrow n_1 \quad E_2 \Rightarrow n_2}{E_1 \texttt{ - } E_2 \Rightarrow n_1 - n_2} \quad \text{E-Minus}$$

$$\frac{E_1 \Rightarrow n_1 \quad E_2 \Rightarrow n_2}{E_1 \texttt{ * } E_2 \Rightarrow n_1 * n_2} \quad \text{E-Mult}$$

$$\frac{E_1 \Rightarrow n_1 \quad E_2 \Rightarrow n_2}{E_1 \texttt{ / } E_2 \Rightarrow n_1/n_2} \; n_2 \neq 0 \quad \text{E-Div}$$

The first rule (E-Num) means that, for any integer n, (n, n) is

an element of the set $\Rightarrow$. The meaning of the integer expression n is defined as the integer value n. The second rule (E-PLUS) states that if (E_1, n_1) and (E_2, n_2) are elements of $\Rightarrow$, then $(E_1$ `+` $E_2, n_1 + n_2)$ is also an element of $\Rightarrow$.[1)] The meanings of the remaining rules are defined similarly. However, in the case of division, the meaning is defined only for cases where n_2 is not zero.

For example, the set $\Rightarrow$, defined by the above rules, contains (`((1+2)*(3/3)`, 3) as an element. The meaning of the integer expression `(1+2)*(3/3)` has been defined as 3. We can prove that (`((1+2)*(3/3)`, 3) is an element of the set $\Rightarrow$ as follows.

$$\dfrac{\dfrac{\dfrac{}{\texttt{1} \Rightarrow 1}\ \text{E-NUM} \quad \dfrac{}{\texttt{2} \Rightarrow 2}\ \text{E-NUM}}{\texttt{1+2} \Rightarrow 3}\ \text{E-PLUS} \quad \dfrac{\dfrac{}{\texttt{3} \Rightarrow 3}\ \text{E-NUM} \quad \dfrac{}{\texttt{3} \Rightarrow 3}\ \text{E-NUM}}{\texttt{3/3} \Rightarrow 1}\ \text{E-DIV}}{\texttt{(1+2)*(3/3)} \Rightarrow 3}\ \text{E-MULT}$$

On the other hand, some integer expressions may be syntactically correct but have no defined meaning. For example, for the integer expression `(3*4)/((1*2)-(1+1))`, n cannot be proven to be `(3*4)/((1*2)-(1+1))` $\Rightarrow n$ using the inference rules above, regardless of the integer n. Since the entire expression involves

1) Here, `+` is a symbol used in the program to denote addition, while $+$ is the operator denoting addition in mathematics. Therefore, $n_1 + n_2$ denotes the integer corresponding to the sum of n_1 and n_2. In other words, `+` is a symbol belonging to the object language currently being defined, while $+$ is a symbol belonging to the metalanguage used to define that language. Unless there is a risk of confusion, we will not distinguish between these two in the future.

division, the only way to prove it would be to apply rule E-DIV; however, in this case, the denominator evaluates to 0, so applying E-DIV is impossible. Thus, not every program defined by the syntactic structure of a programming language has an execution meaning. There may be programs that do not execute properly even after passing the compiler's syntax analysis stage.

So far, we have defined the semantic structure from the perspective of inference rules that define sets, but the above rules can also be interpreted intuitively from the perspective of program execution rules. For example, rule E-PLUS represents the process of evaluating expression E_1 + E_2. To evaluate E_1 + E_2, it first recursively calculates the values of expressions E_1 and E_2 to obtain n_1, n_2, and then adding these two integers. Interpreting the semantic structure in this way means that the process of constructing a proof tree becomes the process of executing the program, and implementing the semantic structure definition as a recursive function directly yields an interpreter for the programming language.

1.3 Inductive Proof

Induction is also a proof method used to prove properties possessed by elements of a set defined inductively. Suppose there is a set S defined by induction. To show that a certain property

$P(x)$ holds for every element x in the set S, it suffices to show the following two things.

1. If x is a fundamental element of S, then $P(x)$ follows directly.

2. If x is defined inductively using other elements $y_1, y_2, \ldots, y_n$ belonging to S,

$$P(y_1), P(y_2), \ldots, P(y_n)$$

assuming that all of these are true, and then use them to show that $P(x)$ is true. In this case, $P(y_1)$, $P(y_2)$, $\ldots$, $P(y_n)$ are called the induction hypothesis.

This method of proof is called structural induction. Mathematical induction is a special case of structural induction that applies when the set S is the set of natural numbers.

Example 1 Let S be the set defined by the following inference rule.

$$\frac{}{3} \qquad \frac{x \quad y}{x+y}$$

Let us prove by structural induction that every element in the set S, defined in this way, is divisible by 3.

- First, we show this for the base elements. If x is 3, it is clearly divisible by 3.
- Next, we prove this for the elements generated inductively. These are the elements generated by the following rule.

$$\frac{x \quad y}{x+y}$$

The induction hypothesis (I.H.) is as follows:

$$x \text{ and } y \text{ are divisible by 3.}$$

By the induction hypothesis, let $x = 3k_1$ and $y = 3k_2$ hold. We will show the following:

$$x + y \text{ is divisible by 3.}$$

The proof is as follows.

$$\begin{aligned} x + y &= 3k_1 + 3k_2 \quad \cdots \text{inductive hypothesis} \\ &= 3(k_1 + k_2) \end{aligned}$$

Example 2 Let S be the set defined by the following inference rule.

$$\frac{}{()} \qquad \frac{x}{(x)} \qquad \frac{x \quad y}{xy}$$

Furthermore, let x be an element of the set S. Then, $l(x)$ and $r(x)$ denote, respectively, the number of left and right parentheses, respectively, contained within x. It is defined inductively as follows.

$$\begin{array}{rclcrcl} l(\texttt{()}) & = & 1 & & r(\texttt{()}) & = & 1 \\ l(\texttt{(}x\texttt{)}) & = & 1 + l(x) & & r(\texttt{(}x\texttt{)}) & = & 1 + r(x) \\ l(xy) & = & l(x) + l(y) & & r(xy) & = & r(x) + r(y) \end{array}$$

Now, let's prove that for every element x in S, the number of left parentheses equals the number of right parentheses. The proposition to be proven can be written as follows.

$$\text{For every } x \in S,\ l(x) = r(x) \text{ holds.}$$

This can be proven using structural induction as follows.

- The base element is $x = \texttt{()}$. By the definitions of the functions l and r, $l(x) = 1 = r(x)$ holds.
- There are two cases in which induction is applied, as shown below.

$$\frac{x}{\texttt{(}x\texttt{)}} \qquad \frac{x \quad y}{xy}$$

  In the case of the first rule, the inductive hypothesis is as

follows:

$$l(x) = r(x)$$

We must prove that $l(\texttt{(}x\texttt{)}) = r(\texttt{(}x\texttt{)})$. The proof is as follows.

$$\begin{aligned} Definition of l(\texttt{(}x\texttt{)}) &= l(x) + 1 && \cdots l(\texttt{(}x \\ &= r(x) + 1 && \cdots \text{inductive hypothesis} \\ &= r(\texttt{(}x\texttt{)}) && \cdots \text{definition of } r(\texttt{(}x\texttt{)}) \end{aligned}$$

For the second rule, the inductive hypothesis is

$$l(x) = r(x), \qquad l(y) = r(y)$$

It is equal to , and the proof is as follows.

$$l(xy) = r(xy).$$

The proof is as follows.

$$\begin{aligned} l(xy) &= l(x) + l(y) && \cdots \text{Definition of } l(xy) \\ &= r(x) + r(y) && \cdots \text{Inductive Hypothesis} \\ &= r(xy) && \cdots \text{Definition of } r(xy) \end{aligned}$$

2

Functional Programming

This chapter introduces functional programming using OCaml[1)] There are three main reasons why this book covers functional programming. First, as we will be designing and implementing programming languages in the future, we will use OCaml—a functional programming language—as our implementation language. This is because functional languages are very convenient for implementing programs that treat other programs as data, such as programming language interpreters or type systems. The second reason is that functional languages, including OCaml, are well-designed from the perspective of programming language theory. Simply examining the characteristics of these languages provides valuable insight into programming languages. In particular, since the concepts of functional programming have recently been adopted by most programming languages, understanding them is essential for gaining a deep understanding of other languages as well. Finally, the language we will be developing is similar to OCaml. This will give you a preliminary sense of the features of the pro-

1) https://ocaml.org (available for download here).

gramming language we will design starting in the next chapter.

2.1 OCaml Basics

Let's learn the most basic concepts needed to write programs in OCaml.[2)]

Running the Program

There are three main ways to run an OCaml program. Use a text editor to write the program as shown below and save it as a file named `hello.ml`.

```
let _ = print_endline "Hello World"
```

This program uses the standard output function `print_endline` to print the string ``Hello World''.

To run the program above, you can first use the REPL (Read-Eval-Print Loop, an interactive program execution tool). If you type `ocaml` at the command prompt, the REPL will start as shown below (let's assume that `$` and `#` are the characters representing the shell and REPL prompts, respectively).

```
$ ocaml
        OCaml version 4.09.0
```

2) Some of the examples in this section are taken from the following book: Yaron Minsky, Anil Madhavapeddy, Jason Hickey. Real-World OCaml. O'Reilly

```
#
```

If you enter the following and press Enter, it will print the string.

```
# print_endline "Hello World";;
Hello World
- : unit = ()
```

Here, `;;` is a sequence used to instruct the REPL to execute a command; you do not need to include it when writing the program in a text file. Instead of entering the program directly into the REPL, you can also load the entire file where the program is stored.

```
# #use "hello.ml";;
Hello World
- : unit = ()
```

You can also use the OCaml interpreter directly instead of the REPL. Simply specify the name of the file to execute after the `ocaml` command, as shown below.

```
$ ocaml hello.ml
Hello World
```

In a Linux environment, here's how to use the compiler (`ocamlc`) to create and run an executable file:

```
$ ocamlc helloworld.ml
$ ls
```

```
a.out   hello.cmi  hello.cmo   hello.ml
$ ./a.out
Hello World
```

Compiling generates an executable file named `a.out`, and running it outputs the string ``‘‘Hello World’’``.

Basic Units of Program Structure

Compared to imperative languages such as C, Java, and Python, one of the characteristics of functional programming languages like OCaml is that the basic unit of a program is an expression, not a statement. In general, in programming languages, a statement refers to a construct that changes the program's state. For example, the assignment statement `x = x + 1` in C or Java is a statement that increments the value of the variable `x` stored in memory by 1. In contrast, an expression refers to a construct that produces a value as the result of a calculation without changing the program's state. For example, the expression `x + y` simply calculates a new value by referencing the values of `x` and `y`, without altering the memory state. In functional languages such as OCaml, Haskell, and Lisp, it is common to write programs that calculate values based on expressions without modifying the memory state.

Since programs are written around expressions rather than

statements, it is natural in functional programming to focus on describing the problem to be solved rather than the procedure for solving it. As such, the choice of programming language greatly influences the way of thinking used to approach a problem; we will explore this in more detail with examples later.

Primitive Values

Among the most basic expressions provided by OCaml are those that produce primitive values such as integers, real numbers, true/-false, characters, and strings. For example, let's enter the arithmetic expression `1 + 2 * 3` in the REPL.

```
# 1 + 2 * 3;;
- : int = 7
```

The OCaml interpreter evaluates the expression provided as input and reports that the result is 7, along with the result's type. In this case, `int` means that the result is of type integer. Floating-point expressions can be evaluated as follows.

```
# 1.1 +. 2.2 *. 3.3;;
- : float = 8.36
```

The result of the above expression is 8.36, which means it is of type `float`. OCaml is a language that clearly classifies values by type. In fact, the addition operators for floating-point numbers and integers are different. As shown in the example above, when

performing operations on floating-point numbers, you must prefix the operator with `.`. If you do not explicitly specify the type of a value, a type error occurs, as shown below.

```
# 3 + 2.0;;
Error: This expression has type float but an
expression was expected of type int
```

This means that the `+` operator expects an integer as an operand, but since the value on the right is a floating-point number, the calculation cannot be performed. To add an integer and a floating-point number, you must explicitly convert the types as follows.

```
# 3 + (int_of_float 2.0);;
- : int = 5
# (float_of_int 3) +. 2.0;;
- : float = 5.
```

`int_of_float` and `float_of_int` are functions that convert floating-point numbers to integers and integers to floating-point numbers, respectively.

A Boolean expression is an expression whose result is either true or false. True and false are represented by `true` and `false`, respectively, and their type is `bool`.

```
# true;;
- : bool = true
# false;;
- : bool = false
```

Comparison operations on arithmetic expressions also produce Boolean values.

```
# 1 = 2;;  (* equal to *)
- : bool = false
# 1 <> 2;;  (* not equal to *)
- : bool = true
# 2 <= (1+1);;  (* less than or equal to *)
- : bool = true
```

You can use logical operators to combine existing Boolean expressions to create new ones.

```
# (2 > 1) && (3 > 2 || 5 < 2);;
- : bool = true
# not (2 > 1);;
- : bool = false
```

In addition, OCaml provides character, string, and unit values as built-in types.

```
# 'c';;
- : char = 'c'
# "Objective " ^ "Caml";;
- : string = "Objective Caml"
# ();;
- : unit = ()
```

`unit` is a type that holds only one value, which is represented by `()`. You can think of it as serving a similar role to the `void` type in C. In the example above, `^` is the operator that concatenates two strings.

Conditional Expressions

One of the most basic constructs provided by a programming language is the conditional statement. Unlike C or Java, in OCaml, a conditional statement is also an expression and therefore evaluates to a value. A conditional expression looks like this:

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

Here, e_1, e_2, and e_3 represent arbitrary expressions. The conditional expression means that if the value of e_1 is true, e_2 is evaluated; if the value of e_2 is false, e_3 is evaluated. For example, the program below corresponds to the cases where e_1, e_2, and e_3 are `2 > 1`, `0`, and `1`, respectively.

```
# if 2 > 1 then 0 else 1;;
- : int = 0
```

Since the value of the expression `2 > 1` is `true`, the expression `0` corresponding to e_2 is evaluated, and its value becomes the value of the entire conditional expression. If the condition is changed as follows, the value of the entire conditional expression becomes the value of e_3.

```
# if 2 < 1 then 0 else 1;;
- : int = 1
```

When writing conditional expressions in OCaml, you must follow several rules. First, e_1 must be a Boolean expression. For

example,

```
if 1 then 2 else 3
```

will result in a type error at compile time.

```
# if 1 then 2 else 3;;
Error: This expression has type int but an expression
was expected of type bool
```

Furthermore, e_2 and e_3 must be expressions that compute values of the same type. For example, the expression below also results in a type error and will not execute.

```
# if true then 1 else true;;
Error: This expression has type bool but an expression
was expected of type int
```

The second rule is actually a requirement of OCaml's static type system. Even if the types of e_2 and e_3 are different, there is no problem executing the program; however, this constraint is necessary because the type is determined before execution. As such, languages with static type systems generally impose more rules on program writing than dynamic languages do. You will understand why once you study the type system in Chapter 8.

Variables

In OCaml, a variable is a name assigned to a value. To assign a name, use the keyword `let`. For example,

```
# let x = 3 + 4;;
val x : int = 7
# let y = x + x;;
val y : int = 14
```

the value of `3 + 4` is named `x`, and the value of `x + x` is named `y`. Names defined in this way become a kind of global variable that can be accessed anywhere in the program. To define local variables whose names are valid only within a limited scope, use the `let...in` expression.

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

This means to define the variable x to have the value of e_1, and then evaluate the expression e_2. In this case, the scope of the variable x is e_2. In other words, the variable x, which was just defined, is meaningful only within the expression e_2. Furthermore, since `let...in` is also an expression, its value is evaluated. That value is the result of the expression e_2. For example, consider the program below.

```
# let a = 1 in a * 2;;
- : int = 2
```

After defining the variable `a` as 1, `a * 2` was evaluated, and that value became the value of the entire expression. In this case, the variable `a` is valid only within `a * 2`, which corresponds to e_2.

Therefore, if you attempt to access `a` after evaluating the `let` expression above, an error occurs.

```
# a;;
Error: Unbound value a
```

This means you attempted to access the undefined variable `a`.

`let` x `=` e_1 `in` e_2, where e_1 and e_2 can contain any expression. You can also use nested '`let`' statements as follows.

```
#  let d =
    let a = 1 in
    let b = a + a in
    let c = b + b in
      c + c;;
val d : int = 8
# d;;
- : int = 8
# a;;
Error: Unbound value a
# b;;
Error: Unbound value b
# c;;
Error: Unbound value c
```

The value of the global variable `d` was defined using a nested `let`. Since the variables `a`, `b`, and `c` above were local variables used temporarily to define the value of `d`, they become inaccessible after `d` is defined.

In the example below, a nested '`let`' is used in place of the '`let`' in e_1.

```
# let x = (let y = 2 in y * 2) in x;;
- : int = 4
# y;;
Error: Unbound value y
```

In this case as well, since the variable `y` is a local variable used to define `x`, it cannot be accessed after the entire expression has been evaluated.

Functions

We also use `let` to define functions. The following defines the function `square`.

```
# let square x = x * x;;
 val square : int -> int = <fun>
```

After the function name, list the names of the function's arguments, and define the function body after `=`. The function `square` above takes an argument `x` and returns `x * x` as its result. When you define a function, the OCaml interpreter specifies its type; in this case, `int -> int` means it is a function that takes an integer as an argument and returns an integer. In the example above, `<fun>` indicates that the defined value is a function. For functions, the OCaml interpreter does not display a concrete value.

After defining a function, you can call and use it as follows.

```
# square 2;;
- : int = 4
# square (2 + 5);;
- : int = 49
# square (square 2);;
- : int = 16
```

The function `square` can also be defined as shown below.

```
# let square = fun x -> x * x;;
val square : int -> int = <fun>
```

Here, `fun x -> x * x` represents a function that takes an argument `x` and returns `x * x`, and the name `square` refers to that function value. This method of defining a function is exactly the same as the one defined earlier. In other words, you can think of the OCaml compiler as internally processing an expression like `let square x = x * x` as `let square = fun x -> x * x`. A function defined as `fun x -> x * x` is called an anonymous function.

As another example, let's define a function `neg` that checks whether the argument `x` is negative. This is a case where the body of the function is a conditional expression.

```
# let neg x = if x < 0 then true else false;;
val neg : int -> bool = <fun>
# neg 1;;
- : bool = false
# neg (-1);;
- : bool = true
```

In OCaml, a function call expression typically looks like e_1 e_2, while e_1 and e_2 can be arbitrary expressions. For example, in the three examples above that call `square`, e_1 is always `square`, while e_2 can be `2`, `2 + 5`, or `square 2`, respectively. As shown below, an unnamed function can also be passed directly to e_1.

```
# (fun x -> x * x) 3;;
- : int = 9
# (fun x -> if x > 0 then x + 1 else x * x) 1;;
- : int = 2
```

A function can have multiple arguments. For example, let's write a function that takes two integers, `x` and `y`, as arguments and returns the sum of their squares.

```
# let sum_of_squares x y = (square x) + (square y);;
val sum_of_squares : int -> int -> int = <fun>
# sum_of_squares 3 4;;
- : int = 25
```

OCaml tells us that the type of the above function is `int -> int -> int`, which denotes a function type that takes two integers and returns an integer. Alternatively, we can interpret this type as `int -> (int -> in` that is, a function that takes one integer argument and returns a function that maps from integers to integers. In fact, the function defined above can be interpreted and used in this way.

```
# let f = sum_of_squares 3;;
val f : int -> int = <fun>
```

```
# f 4;;
- : int = 25
```

The variable `f` is defined as the value of `sum_of_squares 3`, which is a function that takes an integer `y` and returns the value of `(square 3) + (square y)`. Therefore, `f` can be called with the value 4, and its result is 25.

The method for defining recursive functions is the same, but you must append the keyword `rec` after `let`. For example, the function `factorial` can be defined as follows.

```
# let rec factorial a =
    if a = 1 then 1 else a * factorial (a - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

OCaml is a language that treats functions as first-class. This means that functions can be defined and used with great flexibility. In a programming language, for a value to be first-class, it must be possible to store it in a variable, pass it as an argument to a function, and return it as the result of a function. For example, in the C language, values such as integers and characters are treated as first-class. In OCaml, functions are also treated as first-class, so a function can take another function as an argument, as shown below.

```
# let sum f a b =
```

```
    (if f a then a else 0) + (if f b then b else 0)
val sum : (int -> bool) -> int -> int -> int = <fun>
# let even x = x mod 2 = 0;;
val even : int -> bool = <fun>
# sum even 3 4;;
- : int = 4
# sum even 2 4;;
- : int = 6
```

The function `even` is a function that returns true if the integer `x` passed as an argument is even. That is, the body expression `even`'s `x mod 2 = 0` is a boolean expression that calculates true if the remainder of `x` when divided by 2 is 0. The function `sum`receives arguments `f`, `a`, and `b`, where `f` is a function from integers to boolean values. This function applies `a` and `b` and returns the sum only when true is calculated. For example, `sum even 3 4` has the value of 4 because the function `sum` was given the function `even` as its first argument.

A function can also return another function.

```
# let plus a = fun b -> a + b;;
val plus : int -> int -> int = <fun>
# let plus1 = plus 1;;
val plus1 : int -> int = <fun>
# plus1 1;;
- : int = 2
# plus1 2;;
- : int = 3
# let plus2 = plus 2;;
val plus2 : int -> int = <fun>
```

```
# plus2 1;;
- : int = 3
```

The function `plus` is a function that takes an argument `a` and returns the function `fun b -> a + b`. The type `int -> int -> int` can be interpreted as `int -> (int -> int)`, meaning it is a function type that takes an integer and returns a function that maps from an integer to an integer. Since the variable `plus1` is defined as `plus 1`, `plus1` refers to a function that takes an argument `b` and returns `1 + b`. Similarly, `plus2` is a function that takes an integer argument and adds 2 to it.

Functions that take other functions as arguments or return them—such as `sum` and `plus`—are called higher-order functions. When a programming language supports higher-order functions, it allows for more abstract thinking and programming. In other words, it becomes possible to abstract and name programming patterns, which is advantageous for writing concise and readable programs. We will explore higher-order functions in more detail in Section 2.3.

Static Type System

OCaml is a language with a static type system. Generally, programming languages can be classified into statically typed languages and dynamically typed languages; OCaml belongs to the

former category, along with C, C++, Java, Scala, and others. Since these languages perform type checking statically during the compilation phase, programs with type errors, such as the one shown below, will not pass the compiler.

```
# 1 + true;;
Error: This expression has type bool but an
expression was expected of type int
```

In contrast, languages with dynamic type systems, such as Python and Lisp, perform type checking during program execution. Rather than checking in advance whether a program contains type errors, they monitor for calculations with incorrect types as the program runs.

These two categories have contrasting advantages and disadvantages. For example, languages with static type systems are suitable for developing programs that require high stability, as they can detect type errors during program development. On the other hand, languages with dynamic type systems offer greater freedom of expression and flexibility than static languages, which allows for faster program development.

Languages with static type systems can be further divided into two categories. The first category consists of languages with sound type systems. These languages detect all possible type errors that could occur during program execution without ex-

ception at the compilation stage. Functional languages such as OCaml, Haskell, and Scala typically fall into this category. The second category consists of languages that have a static type system but are not type-safe, meaning type errors can still occur at runtime. C and C++ are prime examples.

OCaml statically checks a program's types while also automatically inferring them. For example, in C or Java, you always had to explicitly specify the types of variables and functions as shown below.

```
public static int f(int n) {
  int a = 2;
  return a * n;
}
```

In OCaml, you can define the function above without specifying type information, and the compiler automatically infers the types for you.

```
# let f n =
    let a = 2 in
      a * n;;
val f : int -> int = <fun>
```

OCaml can automatically infer types no matter how complex the program is. To see how the OCaml runtime automatically infers types, let's take the function `sum`, defined earlier, as an example.

```
# let sum f a b =
    (if f a then a else 0) + (if f b then b else 0)
val sum : (int -> bool) -> int -> int -> int = <fun>
```

OCaml automatically infers type information by analyzing the body of a function through the following process. First, in the conditional expression `if` e_1 `then` e_2 `else` e_3, the types of e_2 and e_3 must be the same, and since the type of `0` is an integer, we can infer that the types of the arguments `a` and `b` are `int`. Next, since `f` is used in the form of a function call (`f a` or `f b`), it must have a function type; furthermore, `a` or `b` is passed as an argument to `f`, and the result of `f a` or `f b` are used as the result of the conditional expression e_1, we infer that `f` must have the function type `int -> bool`. Finally, since `sum` returns the result of an addition, we infer that its type is `int`. Through this process, OCaml's automatic type inference algorithm automatically infers the types from the program code before execution. The working principles of automatic type inference will be examined in detail in Chapter 8.

Even though types are inferred automatically, you may want to specify them explicitly. You can do so as follows.

```
# let sum (f : int -> bool) (a : int) (b : int) : int
  = (if f a then a else 0) + (if f b then b else 0);;
val sum : (int -> bool) -> int -> int -> int = <fun>
```

In this case, OCaml verifies whether the type you specified is correct. For example, if the type you specified is incorrect, as shown below, it automatically detects the error.

```
# let sum (f : int -> int) (a : int) (b : int) : int =
    (if f a then a else 0) + (if f b then b else 0);;
Error: The expression (f a) has type int but an
expression was expected of type bool
```

This means that the user accidentally specified the type of `f` as `int -> int`, but since the return type of `f` must be `bool`, this is incorrect. Because OCaml's type system is sound, it will always detect errors in user-specified types.

In some cases, the type of an expression may not be uniquely determined. For example, the function `id` defined below can be used with any type.

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# id 1;;
- : int = 1
# id "abc";;
- : string = "abc"
# id true;;
- : bool = true
```

As shown above, when a function operates on arbitrary types, OCaml uses type variables such as `'a` to represent the type. Such types are called polymorphic types, and functions that accept

polymorphic types are called polymorphic functions. This means the function can be used with any type.

OCaml's polymorphic type system is not entirely unrestricted; it supports only polymorphic functions defined using `let`. This type system is called the let-polymorphic type system. For example, if you define the function `f` as shown below, it is recognized as a polymorphic function and executes without any issues.

```
# let f = fun x -> x in
    let x = f 1 in
      let y = f true in
        3;;
- : int = 3
```

However, if you write a program with the same meaning without the `let` expression, as shown below, a type error occurs.

```
# (fun f ->
    let x = f 1 in
      let y = f true in
        3) (fun x -> x);;
Error: The expression has type bool but an expression
was expected of type int
```

Pattern Matching

Pattern matching allows for a concise expression of nested conditions. For example, consider the program below.

```
let is123 s = if s = "1" then true
              else if s = "2" then true
              else if s = "3" then true
              else false
```

The function above can be written using pattern matching as shown below.

```
let is123 s =
  match s with
  "1" -> true
 |"2" -> true
 |"3" -> true
 | _  -> false
```

Alternatively, it can be defined even more simply as follows.

```
 let is123 s =
  match s with
  "1" | "2" | "3" -> true
  | _ -> false
```

This means that if the value of `s` is one of ‘‘`a`’’, ‘`b`’, or ‘‘`c`’’, then `true` is returned; otherwise, `false` is returned. Pattern matching is particularly useful when working with data types such as tuples and lists, or with values of user-defined types.

When using nested pattern matching, it is recommended to enclose the nested `match...with` statements in parentheses as shown below

```
match x with
| 1 -> true
| 2 ->
  (match y with
    | "a" -> true
    | "b" -> false)
| 3 -> false
```

Alternatively, you can enclose them in `begin ... end` statements instead of parentheses.

```
match x with
| 1 -> true
| 2 ->
  begin
    match y with
    | "a" -> true
    | "b" -> false
  end
| 3 -> false
```

If you do not follow the instructions above, the pattern on the last line will be interpreted as a case of `match y with`, resulting in a type error.

Tuples and Lists

Tuples and lists are the most commonly used data structures in functional programming languages.

A tuple is a collection of values. For example, the tuple `(1, ‘‘one’’)` is a collection of an integer and a string, and its type is ex-

pressed as `int * string`. The tuple `(2, ‘‘two’’, true)` is a collection of an integer, a string, and a boolean value, and its type is `int * string * bool`.

```
# let x = (1, "one");;
val x : int * string = (1, "one")
# let y = (2, "two", true);;
val y : int * string * bool = (2, "two", true)
```

To access each element of a tuple, you can use pattern matching. For example, you can define the functions `fst` and `snd` to retrieve the first and second elements of a tuple consisting of two elements, as follows.

```
# let fst p = match p with (x,_) -> x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd p = match p with (_,x) -> x;;
val snd : 'a * 'b -> 'b = <fun>
```

Alternatively, you can use tuple patterns directly as function arguments, as shown below.

```
# let fst (x,_) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (_,x) = x;;
val snd : 'a * 'b -> 'b = <fun>
```

The type `'a * 'b -> 'a` indicates that the function `fst` is a polymorphic function that takes a tuple of arbitrary types `'a` and `'b`

as input and returns a value of type `'a`. By looking at a function's type in this way, you can infer to some extent what the function does.

Tuple patterns can be used not only in function arguments but also in `let` statements. For example, consider the code below.

```
# let p = (1, true);;
val p : int * bool = (1, true)
# let (x,y) = p;;
val x : int = 1
val y : bool = true
```

`p` represents the tuple `(1, true)`, which is unpacked into `x` and `y`.

A list is a sequence of elements of the same type. For example, a list consisting of the numbers 1, 2, and 3 is expressed as `[1; 2; 3]`, and its type is `int list`.

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
```

In OCaml, each element of a list is separated by a semicolon (`;`). Note that a list such as `[1, 2, 3]`, where the elements are separated by commas (`,`), is interpreted as `[(1,2,3)]`—a list whose elements are the tuple `(1,2,3)`—so caution is required.

```
# [1,2,3];;
- : (int * int * int) list = [(1, 2, 3)]
```

```
# [(1,2,3)];;
- : (int * int * int) list = [(1, 2, 3)]
```

An empty list is represented as `[]` and has the polymorphic type `'a list`.

```
# [];;
- : 'a list = []
```

In OCaml, all elements of a list must be of the same type. For example, `[1; true]` is not a list in OCaml.

```
# [1; true];;
Error: This expression has type bool but an expression
was expected of type int
```

This constraint also stems from the static type system. For example, in languages with dynamic type systems, such as Python, values of different types can be used together in a list.

Furthermore, a list is an ordered sequence of elements. Therefore, the two lists below are distinct.

```
# [1;2;3] = [2;3;1];;
- : bool = false
```

The first element of a list is called the head, and the remaining list excluding the first element is called the tail. For example, in the list `[1; 2; 3]`, the head is `1`, and the tail is `[2; 3]`. Generally, when the type of a list is t `list`, the type of the head is t and the type of the tail is t `list`.

The elements of a list can be values of any type. Of course, the type of each element must be the same.

```
# [1;2;3;4;5];;
- : int list = [1; 2; 3; 4; 5]
# ["OCaml"; "Java"; "C"];;
- : string list = ["OCaml"; "Java"; "C"]
# [(1,"one"); (2,"two"); (3,"three")];;
- : (int * string) list =
    [(1, "one"); (2, "two"); (3, "three")]
# [[1;2;3];[2;3;4];[4;5;6]];;
- : int list list = [[1; 2; 3]; [2; 3; 4]; [4; 5; 6]]
```

The last example is a list of integer lists (`int list list`). In this case, the lengths of the lists corresponding to each element can all be different.

```
# [[1;2;3]; [4]; []];;
- : int list list = [[1; 2; 3]; [4]; []]
```

This is because the lists `[1;2;3]` and `[4]` are both integer lists even though they have different lengths, and the empty list `[]` is a polymorphic type and can therefore also be an integer list.

OCaml provides two basic list operators. First, `::` (pronounced "cons") creates a list by adding an element to the front of the list.

```
# 1::[2;3];;
- : int list = [1; 2; 3]
# 1::2::3::[];;
- : int list = [1; 2; 3]
```

To concatenate two lists, use `@` (pronounced "append").

```
# [1; 2] @ [3; 4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

Pattern matching is frequently used when writing functions that manipulate lists. For example, let's define the functions `hd` and `tl` to find the head and tail of a list.

```
# let hd l =
    match l with
    | [] -> raise (Failure "hd is undefined")
    | a::b -> a;;
val hd : 'a list -> 'a = <fun>
# let tl l =
    match l with
    | [] -> raise (Failure "tl is undefined")
    | a::b -> b;;
val tl : 'a list -> 'a list = <fun>
# hd [1;2;3];;
- : int = 1
# tl [1;2;3];;
- : int list = [2; 3]
```

The head and tail of a list are not defined for an empty list. If the list `l` is not empty, it can be decomposed into the head `a` and the tail `b`, and `hd` returns `a`, while `tl` returns `b` (if the list `l` contains only one element, the tail `tl` is an empty list). Note that the return types of the two functions differ in the definition above. If we omit exception handling, we can define them simply as follows.

```
# let hd (a::b) = a;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val hd : 'a list -> 'a = <fun>
# let tl (a::b) = b;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val tl : 'a list -> 'a list = <fun>
```

In this case, OCaml warns that the functions `hd` and `tl` do not account for empty lists. Although these warning messages are not compilation errors, they can cause unhandled exceptions at runtime, so it is best to eliminate them all in advance.

As another example, let's write a function to find the length of a list.

```
# let rec length l =
    match l with
     [] -> 0
    |h::t -> 1 + length t;;
val length : 'a list -> int = <fun>
# length [1;2;3];;
- : int = 3
```

We have defined the length as 0 if the list `l` passed as an argument is empty. If it is not empty, `l` can be split into a head `h` and a tail `t`, and in this case, the length of `l` must be equal to

the length of the tail `t` plus 1. Since `h` is not used in the above definition, it can be omitted using an underscore (`_`) as follows.

```
let rec length l =
  match l with
   [] -> 0
  |_::t -> 1 + length t
```

User-Defined Types

New types are defined using the keyword `type`. First, you can use `type` to assign a new name to an existing type.

```
type var = string
type vector = float list
type matrix = float list list
```

Alternatively, you can create a new set of values and define it as a type. For example, you can define the type `days`, which represents the set of "days of the week," as follows.

```
# type days = Mon | Tue | Wed | Thu | Fri | Sat | Sun;;
# Mon;;
- : days = Mon
# Tue;;
- : days = Tue
```

After the keyword `type`, write the name of the type you want to define (`days`), and after `=`, list the values belonging to that type, separated by `|`. `Mon`, ..., `Sun` are called constructors and each represents a distinct value of type `days`. In OCaml, type

names begin with a lowercase letter, and constructors begin with an uppercase letter. Once a new type has been defined, you can define functions that operate on values of that type. For example, you can define the function `nextday`, which takes a day of the week as input and returns the next day, as follows.

```
# let nextday d =
  match d with
  | Mon -> Tue | Tue -> Wed | Wed -> Thu | Thu -> Fri
  | Fri -> Sat  | Sat -> Sun | Sun -> Mon ;;
val nextday : days -> days = <fun>
# nextday Mon;;
- : days = Tue
```

You can also define constructors to take different values as arguments. For example, let's define the type `shape`, which can have a rectangle or a circle as its value.

```
# type shape = Rect of int * int | Circle of int;;
type shape = Rect of int * int | Circle of int
```

The constructor `Rect`, which represents a rectangle, is defined to take the width and height as arguments, while the constructor `Circle`, which represents a circle, is defined to take the radius as an argument. Examples of values belonging to this type are as follows:

```
# Rect (2,3);;
- : shape = Rect (2, 3)
# Circle 5;;
- : shape = Circle 5
```

A function that calculates the area of these shapes can be written as follows.

```
# let area s =
    match s with
      Rect (w,h) -> w * h
    | Circle r -> r * r * 3;;
val area : shape -> int = <fun>
# area (Rect (2,3));;
- : int = 6
# area (Circle 5);;
- : int = 75
```

In the example above, for convenience, the value of pi was approximated as 3.

It is also possible to define types inductively. For example, consider the set of lists of integers defined in Chapter 1.

$$\frac{}{\mathsf{nil}} \qquad \frac{l}{n \cdot l}\ n \in \mathbb{Z}$$

In OCaml, the set above can be defined as follows.

```
# type intlist = Nil | Cons of int * intlist;;
type intlist = Nil | Cons of int * intlist
```

We defined the name (type) of the set as `intlist` and defined two ways to create elements of that set. First, `Nil` is a constructor that represents an empty list. `Cons` is a constructor that creates a new list by adding an element to the front of a given list. For example, you can create a list as follows.

```
# Nil;;
- : intlist = Nil
# Cons (1, Nil);;
- : intlist = Cons (1, Nil)
# Cons (1, Cons (2, Nil));;
- : intlist = Cons (1, Cons (2, Nil))
```

For example, `Cons (1, Cons (2, Nil))` denotes the list `[1;2]`. Now we can write functions that operate on lists. The function that returns the length of a list is as follows.

```
# let rec length l =
    match l with
    | Nil -> 0
    | Cons (_, l') -> 1 + length l';;
val length : intlist -> int = <fun>
# length (Cons (1, Cons (2, Nil)));;
- : int = 2
```

Let's define the arithmetic expression we defined earlier as an OCaml data type.

$$\overline{n}\ n \in \mathbb{Z} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ - } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ + } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ * } E_2} \qquad \frac{E_1 \quad E_2}{E_1 \texttt{ / } E_2}$$

The syntax structure above can be defined as follows.

```
type exp =
    Int of int
  | Minus of exp * exp
  | Plus of exp * exp
  | Mult of exp * exp
  | Div of exp * exp
```

For example, `(1+2)*(3/3)` is expressed as follows.

```
# Mult(Plus(Int 1, Int 2), Div(Int 3, Int 3));;
- : exp = Mult (Plus (Int 1, Int 2), Div (Int 3, Int 3))
```

The inductive rules representing the meaning of arithmetic expressions can be implemented as a recursive function as follows.

```
# let rec eval exp =
    match exp with
    | Int n -> n
    | Plus (e1, e2) -> (eval e1) + (eval e2)
    | Mult (e1, e2) -> (eval e1) * (eval e2)
    | Minus (e1, e2) -> (eval e1) - (eval e2)
    | Div (e1, e2) ->
      let n1 = eval e1 in
      let n2 = eval e2 in
        if n2 <> 0 then n1 / n2
        else raise (Failure "division by 0");;
val eval : exp -> int = <fun>
# eval (Mult (Plus (Int 1, Int 2),
              Div (Int 3, Int 3)));;
- : int = 3
```

This is a direct translation of the semantic structure definition into a recursive function. Since division by zero is undefined, an exception is thrown.

Exception Handling

OCaml throws an exception when a runtime error occurs due to the evaluation of an expression whose meaning is undefined. For

example, dividing any number by zero causes a `Division_by_zero` exception, as shown below.

```
# let div a b = a / b;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
Exception: Division_by_zero.
```

To handle exceptions that occur during execution, use `try ... with`.

```
# let div a b =
    try
      a / b
    with Division_by_zero -> 0;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
- : int = 0
```

You can define and use new exceptions using the keyword `exception` as shown below.

```
# exception Fail;;
exception Fail
# let div a b =
    if b = 0 then raise Fail
    else a / b;;
val div : int -> int -> int = <fun>
# div 10 5;;
- : int = 2
# div 10 0;;
```

```
Exception: Fail.
# try
    div 10 0
  with Fail -> 0;;
- : int = 0
```

Modules

A module is a collection of types and values. It serves to group related functions in one place and hide its internal details. For example, let's define the queue data structure as a module as follows.

```
module IntQueue = struct
  type t = int list
  exception E
  let empty = []
  let enq q x = q @ [x]
  let is_empty q = q = []
  let deq q =
    match q with
    | [] -> raise E
    | h::t -> (h, t)
  let rec print q =
    match q with
    | [] -> print_string "\n"
    | h::t -> print_int h; print_string " "; print t
end
```

This is an implementation of a queue using a list. Users of the module can use it as shown below without needing to know the implementation details.

```
let q0 = IntQueue.empty
let q1 = IntQueue.enq q0 1
let q2 = IntQueue.enq q1 2
let (_,q3) = IntQueue.deq q2
let _  = IntQueue.print q1
let _  = IntQueue.print q2
let _  = IntQueue.print q3
```

We created a queue, added and removed elements, and then printed its state. The output is as follows.

```
1
1 2
2
```

So far, we have explained the basic features of OCaml. Knowing just this much should be sufficient to write basic programs without difficulty. In the next two sections, we will study recursive functions and higher-order functions—two programming styles commonly used in functional programming.

2.2 Recursive Functions

In functional programming, recursive functions are primarily used to express repetition instead of loops. This is because recursive functions are not only a general concept that encompasses the idea of loops, but also because thinking recursively often makes it easier to solve problems from a different perspective.

Recursive Function Practice

Let's practice solving problems recursively. All recursive functions have the following structure.

- If the problem to be solved is small enough, solve it directly.
- Otherwise,
    1. we break the original problem down into subproblems with the same structure.
    2. We then recursively find the solutions to these subproblems.
    3. We construct the solution to the original problem by combining the solutions to the subproblems.

Finding the Length of a List Let's apply the above method to the problem of finding the length of a list. First, for an empty list, its length is 0, so we can determine it immediately. If the list is not empty, it can be divided into a head and a tail, and the length of the tail is calculated recursively. Let's call that length n; then, the length of the original list is $n + 1$. This process can be expressed in OCaml as follows.

```
let rec length l =
  match l with
```

```
  | [] -> 0
  | hd::tl -> 1 + length tl
```

Concatenating Lists Let's write a function `append` that concatenates two lists. Although OCaml provides the built-in operator `@`, let's define it ourselves to practice recursive functions. For example, it should behave as follows.

```
# append [1; 2; 3] [4; 5; 6; 7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
# append [2; 4; 6] [8; 10];;
- : int list = [2; 4; 6; 8; 10]
```

`append` should take two lists, `l1` and `l2`, as arguments and return the list corresponding to `l1@l2`. Let's consider the first argument, `l1`, recursively. If `l1` is an empty list, `l2` becomes the list formed by concatenating the two lists. If `l1` is not an empty list, it can be split into a head and a tail; first, we recursively compute the list formed by concatenating the tail with the list `l2`. Adding the head of `l1` to the front of that list yields the answer to the original problem. In OCaml, this is expressed as follows:

```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | hd::tl -> hd::(append tl l2)
```

Reversing a List Let's write a function `reverse` that reverses a list.

```
# reverse [1; 2; 3];;
- : int list = [3; 2; 1]
# reverse ["C"; "Java"; "OCaml"];;
- : string list = ["OCaml"; "Java"; "C"]
```

If the list is empty, reversing it results in an empty list. If the list is not empty, we can reverse the tail first and then append the head to the end. Here's how to implement this in OCaml:

```
let rec reverse l =
  match l with
  | [] -> []
  | hd::tl -> (reverse tl)@[hd]
```

Finding the nth Element in a List Let's write a function `nth` that returns the nth element of a list (assuming n is a natural number). An exception must be thrown for any access that goes beyond the bounds of the given list.

```
# nth [1;2;3] 0;;
- : int = 1
# nth [1;2;3] 1;;
- : int = 2
# nth [1;2;3] 2;;
- : int = 3
# nth [1;2;3] 3;;
Exception: Failure "list is too short".
```

First, for an empty list, the nth element is undefined for any n, so an exception must be thrown. Let's consider the case where the list is not empty. In this case, if n is 0, the head of the list is the nth element. If n is not 0, the $n-1$th element at the tail corresponds to the nth element in the original list. In OCaml, this is expressed as follows:

```
let rec nth l n =
  match l with
  | [] -> raise (Failure "list is too short")
  | hd::tl -> if n = 0 then hd else nth tl (n-1)
```

Removing the First Occurrence of a Specific Element from a List Let's write a function that returns a list from which a given element has been removed.

```
# remove_first 2 [1; 2; 3];;
- : int list = [1; 3]
# remove_first 2 [1; 2; 3; 2];;
- : int list = [1; 3; 2]
# remove_first 4 [1;2;3];;
- : int list = [1; 2; 3]
# remove_first [1; 2] [[1; 2; 3]; [1; 2]; [2; 3]];;
- : int list list = [[1; 2; 3]; [2; 3]]
```

If the list passed as an argument is empty, there is no element to remove, so we simply return an empty list. If the list is not empty, we can split it into a head and a tail. If the element we want to remove matches the head, we simply return the tail as

is. If it does not match, we find the list with the element to be removed removed from the tail and prefix the head to it. In OCaml, this is expressed as follows:

```
let rec remove_first a l =
  match l with
  | [] -> []
  | hd::tl ->
    if a = hd then tl
    else hd::(remove_first a tl)
```

Insertion Sort Let's implement a function that sorts a list using insertion sort. First, let's write a function `insert` that, given a sorted list and an element, finds the appropriate position for that element and inserts it into the list. For example, it should behave as follows:

```
# insert 2 [1;3];;
- : int list = [1; 2; 3]
# insert 1 [2;3];;
- : int list = [1; 2; 3]
# insert 3 [1;2];;
- : int list = [1; 2; 3]
# insert 4 [];;
- : int list = [4]
```

It can be written as a recursive function as follows.

```
let rec insert a l =
  match l with
  | [] -> [a]
  | hd::tl ->
```

```
    if a <= hd then a::hd::tl
    else hd::(insert a tl)
```

Inserting the element `a` into an empty list results in `[a]`. The process of inserting the element `a` into a non-empty list `hd::tl` is as follows: First, if `a` is less than or equal to the first element of the list, `hd`, then `a` is placed before `hd`; otherwise, `a` is inserted recursively into `tl`, and `hd` is placed before it. Once the function `insert` is defined, the sort function `sort` can be defined simply as follows.

```
let rec sort l =
  match l with
  | [] -> []
  | hd::tl -> insert hd (sort tl)
```

An empty list remains empty even after sorting. To sort a non-empty list `hd::tl`, first sort `tl` recursively, and then use `insert` to place `hd` in its correct position.

Functional vs. Imperative Programming

Let's compare the insertion sort function defined above with the code below, which implements it using a loop in the C language.

```
void insert_sort(int arr[], int len) {
  int i, j;
  int tmp;
  for(i = 1; i < len; i++) {
    tmp = arr[i];
```

```
    j = i - 1;
    while(j >= 0 && arr[j] > tmp) {
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = tmp;
  }
}
```

The version written as a recursive function is more concise and readable. Functional programs that use recursive functions are easier to understand because they encourage you to describe the problem itself rather than the procedure for solving it. For example, if we look again at the definition of `sort` given above, it specifies the condition that the sorted list—in the case where the list is not empty, i.e., `hd::tl`—must match the list obtained by sorting `tl` and inserting `hd`. In other words, it describes the condition that the sorted list must satisfy. In other words, it specifies that the function `sort` to be implemented must satisfy the following equivalence relation.

```
sort (hd::tl) = insert hd (sort tl)
```

In the OCaml implementation, the condition that `sort` must satisfy, as described above, is simply implemented as a recursive function. As another example, the function `factorial`, which calculates factorials, must satisfy the following condition when the number given as an argument is greater than 0.

```
factorial n = n * factorial (n-1)
```

The above specification can be implemented as a recursive function, including the case where the argument is 0, as follows.

```
let rec factorial n =
  if n = 0 then 1 else n * factorial (n-1)
```

In contrast, imperative programming is a paradigm that encourages the description of problem-solving procedures. For example, if you were to write a factorial function in the C language, you would most likely implement it as follows.

```
int factorial (int n) {
  int i; int r = 1;
  for (i = 0; i < n; i++)
    r = r * i;
  return r;
}
```

This code describes one of several implementations that satisfy the conditions the factorial function must meet. In other words, in imperative programming, one must go a step further than simply describing the problem and also propose a specific implementation strategy. In contrast, functional programming focuses on describing the problem, and for this reason, it is sometimes referred to as declarative programming.

Cost of Recursive Functions

For reference, in functional programming languages such as OCaml, recursive functions are not more expensive than loops. For example, calling the following function in C causes a stack overflow, but

```
void f() { f(); }       /* stack overflow */
```

in OCaml, the function runs indefinitely without consuming additional memory when called.

```
let rec f () = f ()   (* infinite loop *)
```

In both cases, the function `f` represents an infinite loop, and OCaml is able to execute it exactly as intended. As this example shows, in OCaml, a function is not necessarily more expensive than a loop just because it is defined recursively. A function call is expensive only when the computation it describes is expensive; it is not expensive simply because the computation is described recursively.

To explain in more detail, loops such as `for` and `while` can always be converted into tail-recursive functions, and calling tail-recursive functions is not expensive. A tail-recursive function is a recursive function that has nothing left to do after calling itself. For example, consider the function `last`, which returns the last element of a list:

```
let rec last l =
  match l with
  | [a] -> a
  | _::tl -> last tl  (* tail-recursive call *)
```

Here, the function call `last tl` corresponds to a tail-recursive call. This means that after computing the result of `last tl`, there is no further work to be done using that value. Since there is no further work to be done with that result, when a function is called in the form of tail recursion, there is no need to return to the point of the call; therefore, no additional memory needs to be allocated when the function is called. Many functional language compilers support this tail-call optimization.

On the other hand, in cases where a function is not tail-recursive—such as the factorial function—additional memory is required when making a recursive call.

```
let rec factorial n =
  if n = 1 then 1 else n * factorial (n - 1)
```

After calculating the recursive call `factorial (a-1)`, there is still work to be done (multiplying by `a`) using the result, so this is not a tail-recursive call. In this case, the recursive call consumes memory, and calling `factorial` for large numbers may cause a stack overflow.

Note that a function not defined as tail-recursive can always be converted into a tail-recursive form. This follows the same

principle as the fact that every recursive function can be expressed as a loop. For example, `factorial` can be defined in tail-recursive form as follows:

```
let rec factorial n r =
  if n = 1 then r else factorial (n - 1) (r * n)
```

For example, 5 factorial is calculated as `factorial 5 1`.

2.3 Higher-Order Functions

In functional programming, higher-order functions are used extensively in addition to recursive functions. A higher-order function is a function that takes other functions as arguments or returns them. Higher-order functions allow programs to be written at a higher level by abstracting programming patterns and making them reusable. Let's take a closer look at the commonly used higher-order functions: `map`, `filter`, and `fold`.

map

The three functions defined below—`inc_all`, `sqaure_all`, and `cube_all`—increment, square, and cube each element of a given list, respectively.

```
let rec inc_all l =
  match l with
  | [] -> []
```

```
  | hd::tl -> (hd+1)::(inc_all tl)

let rec square_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)

let rec cube_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd*hd*hd)::(cube_all tl)
```

These functions are all defined according to a common pattern. The only difference lies in the operation applied to each element of the list. If we generally refer to this operation as `f`, the programming pattern shared by the three functions above can be expressed using the following higher-order function, `map`.

```
let rec map f l =
  match l with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

`map` takes as arguments a list `l` and a function `f` that specifies the operation to apply to each element. It then generates a list in which each element `a` of the list `l` is replaced with the value `f a`. In other words, when `l` is a list `[a1; a2; ...; ak]`, `map f l` produces a new list `[f a1; f a2; ...; f ak]`. Examin-

ing the type of `map` provides some insight into this meaning.

```
map : ('a -> 'b) -> 'a list -> 'b list
```

This means that the type of the function to be applied is `'a -> 'b`, and it takes a list of type `'a` as input and produces a list of type `'b` as output. In this way, the type of a function can generally be considered an abstraction of what the function does.

Using the higher-order function `map`, we can define the three functions above concisely as follows.

```
let inc_all l = map (fun x -> x + 1) l
let square_all l = map (fun x -> x * x) l
let cub_all l = map (fun x -> x * x * x) l
```

Alternatively, the common factor `l` can be omitted, and the definition can be written as follows.

```
let inc_all = map (fun x -> x + 1)
let square_all = map (fun x -> x * x)
let cub_all = map (fun x -> x * x * x)
```

Let's compare this with the previous definition. We can see that in the definition using `map`, the meaning of each function is expressed more clearly and directly at the top level. For example, the meaning of `square_all`—applying the function `fun x -> x * x` to each element of the list `l`—is directly expressed. In contrast,

when written recursively without using `map`, this meaning is hidden within the code and is not immediately apparent.

A well-structured program is one that others can easily understand. This is because it is written in such a way that it can be understood at a high level, even without knowing the implementation details. From this perspective, a good programming language is one that allows ideas to be expressed at a higher level. In functional programming, higher-order functions play a major role in raising the level of abstraction in a program.

filter

Although the two functions below perform different tasks, they are defined according to the same pattern.

```
let rec even l =
  match l with
  | [] -> []
  | hd::tl ->
    if hd mod 2 = 0 then hd::(even tl)
    else even tl

let rec greater_than_five l =
  match l with
  | [] -> []
  | hd::tl ->
    if hd > 5 then hd::(greater_than_five tl)
    else greater_than_five tl
```

`even` selects only the even numbers from the list `l`, and `greater_than_five` selects only the numbers greater than 5. In other words, both are functions that select elements from a given list that satisfy specific conditions; the only difference is the selection criteria. This common pattern can be abstracted into the following higher-order function.

```
let rec filter p l =
  match l with
  | [] -> []
  | hd::tl ->
    if p hd then hd::(filter p tl)
    else filter p tl
```

`filter` takes a function `p` and a list `l`, and selects only those elements from `l` that satisfy `p`. The function `p` must be a predicate that returns a Boolean value. In other words, the type of `filter` is as follows.

```
filter : ('a -> bool) -> 'a list -> 'a list
```

Using the higher-order function `filter`, the two functions above can be defined as follows.

```
let even l = filter (fun x -> x mod 2 = 0) l
let greater_than_five l = filter (fun x -> x > 5) l
```

fold

One of the most commonly used higher-order functions in functional programming is `fold`. Let's compare the following two functions.

```
let rec sum l =
  match l with
  | [] -> 0
  | hd::tl -> hd + (sum tl)

let rec prod l =
  match l with
  | [] -> 1
  | hd::tl -> hd * (prod tl)
```

Both functions follow the pattern of iterating over each element of a given list and applying a certain operation recursively. For example, the process of applying the two functions above to the list `[1; 2; 3]` is as follows.

```
sum [1; 2; 3] = 1 + (2 + (3 + 0))
prod [1; 2; 3] = 1 * (2 * (3 * 1))
```

The difference between the two functions lies in the operation to be applied cumulatively and the initial value. For `sum`, the operator is `+` and the initial value is `0`, while for `prod`, the operator is `*` and the initial value is `1`. We can define a higher-order func-

tion `fold_right` that takes these two as additional arguments as follows.

```
let rec fold_right f l a =
  match l with
  | [] -> a
  | hd::tl -> f hd (fold_right f tl a)
```

`f` is the operation to be applied recursively, `l` is the list, and `a` is the initial value. Using `fold_right`, we can define the functions `sum` and `prod` as follows.

```
let sum  lst = fold_right (fun x y -> x + y) lst 0
let prod lst = fold_right (fun x y -> x * y) lst 1
```

The meaning of each function is directly expressed at a higher level. `sum` is a function that starts with an initial value of 0 and applies addition recursively to each element of the list, while `prod` is a function that starts with an initial value of 1 and applies multiplication recursively to each element of the list.

Consider a case where `sum` and `prod` are implemented as tail-recursive functions, as shown below, but have the same meaning as in the example above.

```
let rec sum a l =
  match l with
  | [] -> a
  | hd::tl -> sum (a + hd) tl

let rec prod a l =
```

```
  match l with
  | [] -> a
  | hd::tl -> prod (a * hd) tl
```

The higher-order function that defines the common pattern of the two functions above is called `fold_left`.

```
let rec fold_left f a l =
  match l with
  | [] -> a
  | hd::tl -> fold_left f (f a hd) tl
```

Using `fold_left`, `sum` and `prod` can be defined as follows.

```
let sum a l = fold_left (fun x y -> x + y) a l
let prod a l = fold_left (fun x y -> x * y) a l
```

Let's take a closer look at the difference between `fold_right` and `fold_left`. First, their types differ as shown below.

```
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

An even more important difference is that they accumulate operations in opposite directions. `fold_right` starts with the last element of the list and accumulates from right to left.

```
fold_right f [x;y;z] init = f x (f y (f z init))
```

In contrast, `fold_left` accumulates from the leftmost element of the list, moving toward the right.

```
fold_left f init [x;y;z] = f (f (f init x) y) z
```

Therefore, if the operation `f` does not satisfy the associative law, the two results may differ.

```
# fold_right (fun x y -> x - y) [1;2;3] 0;;
- : int 2
# fold_left (fun x y -> x - y) 0 [1;2;3];;
- : int = -6
```

Finally, `fold_left` is a tail-recursive function. For long lists, it is generally better to use `fold_left` whenever possible.

2.4 Exercises

Problem 1 Write a function `range` that takes two numbers n, m $(n \leq m)$ and returns a list consisting of numbers greater than or equal to n and less than or equal to m.

```
range : int -> int -> int list
```

For example, `range 3 7` produces `[3;4;5;6;7]`.

Problem 2 Write a function `concat` that takes a list of lists and returns a single list containing all elements in order.

```
concat: 'a list list -> 'a list
```

For example, `concat [[1;2];[3;4;5]` produces `[1;2;3;4;5]`.

Problem 3 Write a function `zipper` that sequentially concatenates two lists, `a` and `b`.

```
zipper: int list -> int list -> int list
```

Sequential concatenation means that the ith element of list `a` comes before the ith element of list `b`. Elements that do not have a match are appended in order at the end. For example,

```
# zipper [1;3;5] [2;4;6];;
- : int list = [1; 2; 3; 4; 5; 6]
# zipper [1;3] [2;4;6;8];;
- : int list = [1; 2; 3; 4; 6; 8]
# zipper [1;3;5;7] [2;4];;
- : int list = [1; 2; 3; 4; 5; 7]
```

Problem 4 Write a function `unzip` that splits a list of tuples, each consisting of two elements, into two lists.

```
unzip: ('a * 'b) list -> 'a list * 'b list
```

For example,

```
unzip [(1,"one");(2,"two");(3,"three")]
```

evaluates to `([1;2;3],[‘‘one’’;‘two’;‘‘three’’])`.

Problem 5 Given a list l and an integer n, write a function `drop` that returns the list remaining after removing the first n elements from l.

```
drop : 'a list -> int -> 'a list
```

For example,

```
drop [1;2;3;4;5] 2 = [3; 4; 5]
drop [1;2] 3 = []
drop ["C"; "Java"; "OCaml"] 2 = ["OCaml"]
```

Problem 6 Write a higher-order function `sigma`.

```
sigma : (int -> int) -> int -> int -> int
```

`sigma f a b` computes the following:

$$\sum_{i=a}^{b} f(i).$$

For example,

```
sigma (fun x -> x) 1 10
```

evaluates to 55,

```
sigma (fun x -> x*x) 1 7
```

evaluates to 140.

Problem 7 Write a higher-order function `iter`.

```
iter : int * (int -> int) -> (int -> int)
```

It must satisfy the following conditions.

$$\texttt{iter}(n, f) = \underbrace{f \circ \cdots \circ f}_{n}$$

If $n = 0$, then `iter`(n, f) is the identity function that returns the argument unchanged. If $n > 0$, then `iter`(n, f) is a function that applies f n times. For example,

```
iter(n, fun x -> 2+x) 0
```

computes $2 \times n$.

Problem 8 Use `fold_right` to implement the higher-order function `all`.

```
all : ('a -> bool) -> 'a list -> bool
```

`all p l` indicates whether all elements of the list `l` make the value of the function `p` true. For example,

```
all (fun x -> x > 5) [7;8;9]
```

computes *true*.

Problem 9 Write a function that converts a list of integers into a number using `fold_left`.

```
lst2int : int list -> int
```

For example, `lst2int [1;2;3]` evaluates to 123. Assume that the elements of the list are numbers between 0 and 9, inclusive.

Problem 10 Redefine the following functions using `fold_right` and `fold_left`.

1. ```
   let rec length l =
       match l with
       [] -> 0
       |h::t -> 1 + length t
   ```
   ```

2.
```
let rec reverse l =
    match l with
    | [] -> []
    | hd::tl -> (reverse tl)@[hd]
```

3.
```
let rec is_all_pos l =
  match l with
  | [] -> true
  | hd::tl -> (hd > 0) && (is_all_pos tl)
```

4.
```
let rec map f l =
    match l with
    | [] -> []
    | hd::tl -> (f hd)::(map f tl)
```

5.
```
let rec filter p l =
    match l with
    | [] -> []
    | hd::tl ->
      if p hd then hd::(filter p tl)
      else filter p tl
```

Problem 11 Consider the set of natural numbers defined inductively.

$$\frac{}{0} \qquad \frac{n}{n+1}$$

In OCaml, it can be defined as follows.

```
type nat = ZERO | SUCC of nat
```

For example, `SUCC ZERO` denotes 1, and `SUCC (SUCC ZERO)` denotes 2. Define addition and multiplication for the natural num-

bers defined in this way.

```
natadd : nat -> nat -> nat

natmul : nat -> nat -> nat
```

For example,

```
# let two = SUCC (SUCC ZERO);;
val two : nat = SUCC (SUCC ZERO)
# let three = SUCC (SUCC (SUCC ZERO));;
val three : nat = SUCC (SUCC (SUCC ZERO))
# natmul two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO)))))
# natadd two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC ZERO))))
```

Problem 12 Let us define the arithmetic expression `aexp` as follows.

```
type aexp =
 | Const of int
 | Var of string
 | Power of string * int
 | Times of aexp list
 | Sum of aexp list
```

Write a function that takes an arithmetic expression and a string representing a variable as input, and returns the arithmetic ex-

pression differentiated with respect to that variable.

```
diff : aexp * string -> aexp
```

For example, the arithmetic expression $x^2 + 2x + 1$ is expressed as follows.

```
Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1]
```

If we denote this expression as `e`, then `diff(e, ‘‘x’’)` must return $2x + 2$, which is the derivative of `e` with respect to `x`.

```
Sum [Times [Const 2; Var "x"]; Const 2]
```

3

Variables and the Environment

Now let's begin designing and implementing a programming language in earnest. First, let's extend the integer expression language defined in Chapter 1 to allow the use of variables and conditional expressions.

3.1 Syntactic Structure

The grammatical structure of the extended language is shown in Figure 3.1. Four new constructs have been added to the integer expression language.

- The program allows the variable x to be used at any location where an expression can appear.[1)]
- `let` x `=` E_1 `in` E_2 is an expression that declares the variable x. Let the value of E_1 be x, and then calculate the value of E_2. In this case, the scope in which the variable x can be used is E_2.

1) Here, x is a metavariable that refers to any variable, not a specific one.

E	$\rightarrow$	n	
	$\mid$	E_1 `+` E_2	
	$\mid$	E_1 `-` E_2	
	$\mid$	E_1 `*` E_2	
	$\mid$	E_1 `/` E_2	
	$\mid$	x	variable
	$\mid$	`let` x `=` E_1 `in` E_2	`let` expression
	$\mid$	`if` E_1 `then` E_2 `else` E_3	conditional expression
	$\mid$	`iszero` E	Boolean expression

Figure 3.1: Syntax Structure

- `if` E_1 `then` E_2 `else` E_3 is a conditional expression and is evaluated as follows. First, the expression E_1 is evaluated; it must return either true (*true*) or false (*false*). If the value of E_1 is *true*, then E_2 is evaluated, and if the value of E_1 is *false*, then calculate E_3.

- Since using a conditional expression requires an expression that produces a true/false value, the statement `iszero` E was added. Compute the value of E; if the value is 0, compute *true*; otherwise, compute *false*.

Since it is similar to the OCaml syntax, you will likely get a good sense of what kinds of programs you can write. Let's look at a few examples.

Example 1 The following program defines the variable `x` as 1 and then calculates the value of `x+2`. The result is 3.

```
let x = 1 in x + 2
```

Example 2 As shown below, another `let` expression can appear in place of the E_2 in the `let` expression.

```
let x = 1
in let y = 2
   in x + y
```

Since `x` and `y` in the expression `x + y` represent 1 and 2, respectively, the result of the program's calculation is 3.

Example 3 As shown below, another `let` expression can appear in the E_1 position of the `let` expression.

```
let x = let y = 2
        in y + 1
in x + 3
```

The value of the expression `let y = 2 in y + 1` is 3. If we let this be `x`, then the result of evaluating `x + 3` is 6. Note that the variable `y` cannot be used when evaluating the expression `x + 3`. The scope of the variable `y` is limited to the expression `y + 1`, and it cannot be used elsewhere. In other words, the program below is syntactically correct but will not execute properly.

```
let x = let y = 2
        in y + 1
in x + y
```

Example 4 It is also possible to define the same name multiple times, as shown below.

```
let x = 1
in let y = 2
   in let x = 3
      in x + y
```

When evaluating `x + y` on the last line, `x` refers to the value 3 defined last; therefore, the value of `x + y` is 5.

Example 5 Redefining a variable with the same name does not change the value of the previous variable. This is because defining a variable using `let` creates a new name rather than modifying the value of an existing variable, and that name is meaningful only within its own scope. For example, consider the following program.

```
let x = 1
in let y = let x = 2
           in x + x
   in x + y
```

The result of evaluating the program above is 5. In the first line, the variable `x` is defined as 1. In the second line, the value of `x` is

2; therefore, the value of `x + x` in the third line is 4. As a result, the value of the variable `y` is 4. In the expression `x + y` on the last line, the variable `x` refers to the `x` defined on the first line; therefore, the value of `x + y` is 5.

Example 6 Conditional expressions are used as follows. This is a program that calculates the integer 1.

```
let x = 1
in let y = 2
   in if iszero (x - 1) then y - 1 else  y + 1
```

Example 7 Now that a program can evaluate not only integers but also true/false values, there are programs that cannot be executed due to type mismatches—that is, programs with type errors. For example, consider the program below.

```
let x = 1
in let y = iszero x
   in x + y
```

Addition is an operation defined only for two integers, but since `y` is a Boolean value, a type error occurs when evaluating the expression `x + y`. Since our language does not yet have a static type system, we cannot detect such type errors before execution; instead, a type error occurs during execution, causing the program to terminate abnormally.

3.2 Semantic Structure

Now let's define the semantic structure of the language. These are the rules for executing a program written in the language shown in Figure 3.1.

Notation For the two sets X and Y, take the union of $X+Y$, $X \times Y$ as the product [2], $X \setminus Y$ be the difference. $X \to Y$ denotes the set of all finite functions from X to Y. [3] For a function $f : X \to Y$ to be finite means that its domain is a finite set. This means that the function is defined for a finite number of elements. When $f : X \to Y$ is a finite function, $\mathrm{Dom}(f)$ represents its domain. In other words, $\mathrm{Dom}(f)$ is a finite subset of the set X. A finite function can be represented as a table. For example, if the domain of the function $f(x) = x+1$ is $\{1, 2, 3\}$, then f can be represented as a table, as shown in $\{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4\}$.

3.2.1 Environment

When a program contains variables, its meaning cannot be defined by the program alone; it requires an environment—a mechanism that specifies the current values of the variables. For ex-

2) $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y \ \}$

3) Generally, $X \to Y$ denotes the set of all functions from X to Y, but in this book, we will consider only finite functions.

ample, the meaning of the expression `x+y` depends on the current values of the variables `x` and `y`. In an environment where the value of `x` is 1 and the value of `y` is 2,

$$\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$$

If calculated there, the value of `x+y` is 3, but in other contexts, such as

$$\{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 3\}$$

When calculated there, the same expression `x+y` takes on a different value of 5.

The environment is a finite function that maps variables to values.

$$\mathit{Env} = \mathit{Var} \to \mathit{Val}$$

The set of all possible environments is defined as *Env*. Here, *Var* represents the set of variables available in the program.

$$\mathit{Var} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots\}$$

Val denotes the set of values handled by the program. Since the current language only has integers ($\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$) or true/false ($\mathbb{B} = \{\mathit{true}, \mathit{false}\}$) as values, the set of values *Val* can

be defined as follows.

$$Val = \mathbb{Z} + \mathbb{B}$$

The collection of sets (*Env*, *Var*, *Val*) required to define the meaning of a program in this way is called the semantic domain.

From now on, we will use the following notation for the environment.

- We will use the symbol ρ to represent the environment ($\rho \in Env$). As needed, we will also use $\rho', \rho'', \rho_1, \rho_2$ and so on.

- $\{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\}$ is an environment in which the variable $x_1, \ldots, x_k$ is mapped to the value $v_1, \ldots, v_k$. For example, $\rho = \{x \mapsto 1, y \mapsto 2\}$ refers to an environment where the value of x is defined as 1 and the value of y is defined as 2. For all other names, no values are defined in this environment.

- In the environment ρ, the value denoted by the name x is denoted as $\rho(x)$. Since the environment ρ is a function, ρ has been applied to x. For example, when $\rho = \{x \mapsto 1, y \mapsto 2\}$, $\rho(x)$ is 1 and $\rho(y)$ is 2.

- $\emptyset$ denotes an empty environment. It is an environment in which no variables are defined. In other words, if $\rho = \emptyset$,

then $\rho(x)$ is not defined for any variable $x \in \mathit{Var}$.

- $\{x \mapsto v\}\rho$ denotes an environment obtained by extending the given environment ρ such that the variable x takes the value v. For example, when $\rho = \{x \mapsto 1, y \mapsto 2\}$, $\{x \mapsto 2\}\rho$ denotes the environment $\{x \mapsto 2, y \mapsto 2\}$. Mathematically, $\{x \mapsto v\}\rho$ is a function defined as follows:

$$(\{x \mapsto v\}\rho)(y) = \begin{cases} v & \text{if } x = y \\ \rho(y) & \text{if } x \neq y \end{cases}$$

  In the environment $\{x \mapsto v\}\rho$, the value of the variable y is v if $x = y$, and if it is $x \neq y$, it is the value of y as defined in the environment ρ prior to the expansion. For example, when $\rho = \{x \mapsto 1, y \mapsto 2\}$, it is $(\{x \mapsto 2\}\rho)(y) = \rho(y) = 2$.

- When extending an environment with respect to multiple variables, notation such as $\{x_1 \mapsto v_1, x_2 \mapsto v_2\}\rho$ is used. Extending the environment ρ to x_1 results in v_1, and x_2 is extended to have v_2. The environment ρ is first extended with respect to (x_2, v_2), and the result is then extended once more with respect to (x_1, v_1). It is defined as follows.

$$\{x_1 \mapsto v_1, x_2 \mapsto v_2\}\rho = \{x_1 \mapsto v_1\}(\{x_2 \mapsto v_2\}\rho)$$

  For example, when $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$, $\{x \mapsto$

$2, y \mapsto 3\}\rho$ refers to the environment $\{x \mapsto 2, y \mapsto 3, z \mapsto 3\}$.

3.2.2 Inference Rules

Given the environment ρ, let's define the meaning of the expression E. Let us express "The result of evaluating expression E in environment ρ is v" as follows.

$$\rho \vdash E \Rightarrow v$$

In the case of arithmetic expressions, the semantic structure was defined as a binary relation ($E \Rightarrow v$) between a program and a value; however, since the meaning of a program now depends on the environment, the semantic structure is defined as a ternary relation between the environment, the program, and the value. Such a ternary relation is denoted as $\rho \vdash E \Rightarrow v$. For example,

- $\emptyset \vdash$ `1` $\Rightarrow 1$: This means that the result of evaluating equation `1` in the empty environment is 1.
- $\{$`x` $\mapsto 1\} \vdash$ `x+1` $\Rightarrow 2$: In the environment $\{$`x` $\mapsto 1\}$, the value of the expression `x+1` is 2.
- $\{$`x` $\mapsto 1\} \vdash$ `let y = 2 in x + y` $\Rightarrow 3$: In the $\{$`x` $\mapsto 1\}$ environment, the result of evaluating `let y = 2 in x + y` is 3.

$$\frac{}{\rho \vdash n \Rightarrow n} \quad \text{E-Num}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2} \quad \text{E-Plus}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ - } E_2 \Rightarrow n_1 - n_2} \quad \text{E-Minus}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ * } E_2 \Rightarrow n_1 * n_2} \quad \text{E-Mult}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ / } E_2 \Rightarrow n_1/n_2} \; n_2 \neq 0 \quad \text{E-Div}$$

$$\frac{}{\rho \vdash x \Rightarrow \rho(x)} \quad \text{E-Var}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad \{x \mapsto v_1\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v} \quad \text{E-Let}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v} \quad \text{E-If-T}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v} \quad \text{E-If-F}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{true}} \quad \text{E-Zero-T}$$

$$\frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{false}} \; n \neq 0 \quad \text{E-Zero-F}$$

Figure 3.2: Semantic Structure

The inference rules that define the semantic structure are shown in Figure 3.2. The first five rules are simply extensions of the semantic structure of the integer expression language to include

the environment. The newly added rules concerning variables are E-VAR and E-LET. Rule E-VAR means that, in a given environment ρ, the meaning of the variable x is the value of x defined in that environment $\rho(x)$. Rule E-VAR applies only if the variable x is defined in the current environment ($x \in \mathrm{Dom}(\rho)$). Rule E-LET defines the meaning of the `let` expression. In environment ρ, expression `let` x `=` E_1 `in` E_2 is evaluated as follows:

1. First, the value of the expression E_1 is evaluated in the given environment ρ, yielding the value v_1 ($\rho \vdash E_1 \Rightarrow v_1$).

2. Next, the current environment is extended so that the variable x takes the value v_1 ($\{x \mapsto v_1\}\rho$).

3. In the expanded environment $\{x \mapsto v_1\}\rho$, the expression E_2 is evaluated to yield the value v.

4. The result of evaluating the full expression `let` x `=` E_1 `in` E_2 is v.

The key to evaluating the expression E_2 is to perform the evaluation within the environment ($\{x \mapsto v_1\}\rho$) expanded from ρ. As a result, using the variable x within E_2 not only refers to the value v_1 but also even if variables are redefined within E_1, this will not affect the calculation of E_2.

The condition `if` E_1 `then` E_2 `else` E_3 means that if the result of calculating E_1 is *true*, then calculate E_2 (E-IF-T),

and if it is *false*, then E_3 is computed (E-If-F). The value of `iszero` E is *true* if E evaluates to 0 (E-Zero-T), if the value of E is not 0, then the value of *false* is E-Zero-F.

In the above definition, n represents an integer value, while v is a symbol denoting any value. For example, in the rules for the four basic arithmetic operations, the results of the expressions E_1 and E_2 are expressed as n_1 and n_2, which implies that E_1 and E_2 must evaluate to integer values for their meanings to be defined. Therefore, according to the above semantic structure, an expression such as `1 + true` becomes a program that has no meaning. On the other hand, in the semantic structure definition of the `let` expression, the results of evaluating E_1 and E_2 are represented as v_1 and v, respectively, which means that values of any type can be evaluated. Going forward, we will define semantic structures using symbols that denote the type of the value: n for integer values, b for true/false values, and v for arbitrary values.

Let's run the example programs above according to the semantic structure defined so far. Depending on the situation, when applying inference rules, let's omit rules that correspond to axioms or represent them simply, showing only the key steps.

Example 1 In the empty environment, `let x = 1 in x + 2` is executed as follows.

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{1} \Rightarrow 1} \quad \dfrac{\dfrac{}{\{\texttt{x} \mapsto 1\} \vdash \texttt{x} \Rightarrow 1} \quad \dfrac{}{\{\texttt{x} \mapsto 1\} \vdash \texttt{2} \Rightarrow 2}\ \text{E-Plus}}{\{\texttt{x} \mapsto 1\} \vdash \texttt{x + 2} \Rightarrow 3}\ \text{E-Let}}{\emptyset \vdash \texttt{let x = 1 in x + 2} \Rightarrow 3}$$

Since the given program is a `let` expression, we first apply execution rule E-Let. According to rule E-Let, expression `1` is evaluated in the current environment $\emptyset$ to yield the value 1 (E-Num), then the current environment is expanded with respect to the variable `x` ($\{\texttt{x} \mapsto 1\}$), and `x + 2` is computed. To compute `x + 2`, the execution rule E-Plus is applied, which in turn is executed by applying rules E-Var and E-Num.

Example 2 In an empty environment, the program `let x = 1 in let y = 2 i` is evaluated as follows.

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{1} \Rightarrow 1} \quad \dfrac{\dfrac{}{\{\texttt{x} \mapsto 1\} \vdash \texttt{2} \Rightarrow 2} \quad \dfrac{\dfrac{}{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{x} \Rightarrow 1} \quad \dfrac{}{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{y} \Rightarrow 2}}{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{x + y} \Rightarrow 3}}{\{\texttt{x} \mapsto 1\} \vdash \texttt{let y = 2 in x + y} \Rightarrow 3}}{\emptyset \vdash \texttt{let x = 1 in let y = 2 in x + y} \Rightarrow 3}$$

Example 3 In an empty environment, the program

```
let x = let y = 2
```

```
        in y + 1
in x + 3
```

is evaluated as follows.

$$\dfrac{\dfrac{\emptyset \vdash \texttt{2} \Rightarrow 2 \quad \dfrac{\{\texttt{y} \mapsto 2\} \vdash \texttt{y} \Rightarrow 2 \qquad \{\texttt{y} \mapsto 2\} \vdash \texttt{1} \Rightarrow 1}{\{\texttt{y} \mapsto 2\} \vdash \texttt{y + 1} \Rightarrow 3}}{\emptyset \vdash \texttt{let y = 2 in y + 1} \Rightarrow 3} \quad \dfrac{\{\texttt{x} \mapsto 3\} \vdash \texttt{x} \Rightarrow 3 \qquad \{\texttt{x} \mapsto 3\} \vdash \texttt{3} \Rightarrow 3}{\{\texttt{x} \mapsto 3\} \vdash \texttt{x + 3} \Rightarrow 6}}{\emptyset \vdash \texttt{let x = (let y = 2 in y + 1) in x + 3} \Rightarrow 6}$$

Note that the environment ($\{\texttt{x} \mapsto 3\}$) used to evaluate the expression `x + 3` contains no information about the variable `y`. Therefore, if the program had evaluated `x + y` instead of `x + 3` on the last line, the execution meaning would be undefined. According to the definition of the semantic structure, the variable `y` is naturally valid only within the `let` expression that defines the value of `x`.

Example 4 In an empty environment, the program

```
let x = 1
in let y = 2
   in let x = 3
      in x + y
```

is evaluated as follows.

$$\dfrac{\emptyset \vdash \texttt{1} \Rightarrow 1 \quad \dfrac{\{\texttt{x} \mapsto 1\} \vdash \texttt{2} \Rightarrow 2 \quad A}{\{\texttt{x} \mapsto 1\} \vdash \texttt{let y = 2 in let x = 3 in x + y} \Rightarrow 5}}{\emptyset \vdash \texttt{let x = 1 in let y = 2 in let x = 3 in x + y} \Rightarrow 5}$$

The omitted section above (A) is as follows.

$$\dfrac{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{3} \Rightarrow 3 \quad \dfrac{\begin{array}{c}\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 3\} \vdash \texttt{x} \Rightarrow 3 \\ \{\texttt{y} \mapsto 2, \texttt{x} \mapsto 3\} \vdash \texttt{y} \Rightarrow 2\end{array}}{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 3\} \vdash \texttt{x + y} \Rightarrow 5}}{\{\texttt{y} \mapsto 2, \texttt{x} \mapsto 1\} \vdash \texttt{let x = 3 in x + y} \Rightarrow 5}$$

It is important to understand how the environment changes each time a variable is defined via `let` in the process above.

Example 5 In an empty environment, the program

```
let x = 1
in let y = let x = 2
           in x + x
   in x + y
```

is evaluated as follows.

$$\dfrac{\dfrac{\dfrac{\{\texttt{x} \mapsto 2\} \vdash \texttt{x} \Rightarrow 2}{\{\texttt{x} \mapsto 2\} \vdash \texttt{x} \Rightarrow 2} \quad \dfrac{\{\texttt{y} \mapsto 4, \texttt{x} \mapsto 1\} \vdash \texttt{x} \Rightarrow 1}{\{\texttt{y} \mapsto 4, \texttt{x} \mapsto 1\} \vdash \texttt{y} \Rightarrow 4}}{\{\texttt{x} \mapsto 2\} \vdash \texttt{x + x} \Rightarrow 4} \quad \dfrac{}{\{\texttt{y} \mapsto 4, \texttt{x} \mapsto 1\} \vdash \texttt{x + y} \Rightarrow 5}}{\{\texttt{x} \mapsto 1\} \vdash \texttt{let x = 2 in x + x} \Rightarrow 4} \quad \dfrac{}{\{\texttt{x} \mapsto 1\} \vdash \texttt{let y = (let x = 2 in x + x) in x + y} \Rightarrow 5}}{\emptyset \vdash \texttt{let x = 1 in let y = (let x = 2 in x + x) in x + y} \Rightarrow 5}$$

It is important to understand that the environment $\{\texttt{x} \mapsto 2\}$, created when evaluating the expression `let x = 2 in x + x`, is used only when evaluating `x + x` and does not propagate to the point where the final expression `x + y` is evaluated. As a result, the `x` in `x + y` refers to the `x` defined in the first line.

Example 6 In the environment $\rho = \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 2\}$, the execution process of the program

```
if iszero (x - 1) then y - 1 else y + 1
```

is as follows.

$$\dfrac{\dfrac{\dfrac{\rho \vdash \texttt{x} \Rightarrow 1 \quad \rho \vdash \texttt{1} \Rightarrow 1}{\rho \vdash \texttt{x - 1} \Rightarrow 0}}{\rho \vdash \texttt{iszero (x - 1)} \Rightarrow \mathit{true}} \quad \dfrac{\rho \vdash \texttt{y} \Rightarrow 2 \quad \rho \vdash \texttt{1} \Rightarrow 1}{\rho \vdash \texttt{y - 1} \Rightarrow 1}}{\rho \vdash \texttt{if iszero (x - 1) then y - 1 else y + 1} \Rightarrow 1}$$

3.3 Implementation

Let's implement the interpreter for the language we have designed so far. First, we can define the language's grammatical structure using OCaml data types as follows (for convenience, we have considered only addition and subtraction among the four arithmetic operations).

```
type exp =
  | CONST of int
  | ADD of exp * exp
  | SUB of exp * exp
  | VAR of var
  | LET of var * exp * exp
  | IF of exp * exp * exp
  | ISZERO of exp
and var = string
```

For example, the following program

```
let x = 1
in let y = 2
   in let  y = let x = x + 1
               in x + y
      in x - y
```

It is expressed as follows.

```
LET ("x", CONST 1,
   LET ("y", CONST 2,
      LET ("y", LET ("x", ADD(VAR "x", CONST 1),
                  ADD (VAR "x", VAR "y")),
        SUB (VAR "x", VAR "y"))))
```

To define the semantic structure of a program, we need to define a semantic space. First, since values are either integers or Booleans, let's define them as follows.

```
type value = Int of int | Bool of bool
```

For example, the integer value 1 is represented as `Int 1`, and the Boolean value *true* is represented as `Bool true`. An environment can be implemented as a list of variables and values, as shown below.

```
type env = (var * value) list
let empty_env = []
let extend_env (x,v) e = (x,v)::e
let rec apply_env x e =
  match e with
  | [] -> raise (Failure (x ^ " is not found"))
  | (y,v)::tl -> if x = y then v else apply_env x tl
```

We defined an empty environment as an empty list and defined the function `extend_env` to extend the environment and the function `apply_env` to retrieve a variable's value from the environment.

The semantic structure defined by the inference rules can be implemented using the recursive function `eval` shown below.

```
let rec eval : exp -> env -> value
=fun exp env ->
  match exp with
  | CONST n -> Int n
```

```
| VAR x -> apply_env x env
| ADD (e1,e2) ->
  let v1 = eval e1 env in
  let v2 = eval e2 env in
    (match v1,v2 with
    | Int n1, Int n2 -> Int (n1 + n2)
    | _ -> raise (Failure "Type Error"))
| SUB (e1,e2) ->
  let v1 = eval e1 env in
  let v2 = eval e2 env in
    (match v1,v2 with
    | Int n1, Int n2 -> Int (n1 - n2)
    | _ -> raise (Failure "Type Error"))
| ISZERO e ->
  (match eval e env with
  | Int n when n = 0 -> Bool true
  | Int n when n <> 0 -> Bool false
  | _ -> raise (Failure "Type Error"))
| IF (e1,e2,e3) ->
  (match eval e1 env with
  | Bool true -> eval e2 env
  | Bool false -> eval e3 env
  | _ -> raise (Failure "Type Error"))
| LET (x,e1,e2) ->
  let v1 = eval e1 env in
    eval e2 (extend_env (x,v1) env)
```

This is a direct implementation of the inference rules in the form of a recursive function. We have explicitly added exception handling for cases where type errors might occur during execution—cases that were implicitly handled in the definition of the inference rules.

The function that executes the program can be defined as follows. It calls `eval` with the given program and an empty environment.

```
let run : exp -> value
=fun pgm -> eval pgm empty_env
```

For example, you can run the program as follows.

```
# let e1 = LET ("x", CONST 1, ADD (VAR "x", CONST 2));;
val e1 : exp = LET ("x", CONST 1, ADD (VAR "x", CONST 2))
# run e1;;
- : value = Int 3
# let e2 = LET ("x", CONST 1,
   LET ("y", CONST 2,
      LET ("y", LET ("x", ADD(VAR "x", CONST 1),
                  ADD (VAR "x", VAR "y")),
        SUB (VAR "x", VAR "y"))));;
val e2 : exp =
  LET ("x", CONST 1,
   LET ("y", CONST 2,
    LET ("y", LET ("x", ADD (VAR "x", CONST 1),
                  ADD (VAR "x", VAR "y")),
     SUB (VAR "x", VAR "y"))))
# run e2;;
- : value = Int (-3)
```

4 Defining and Calling Functions

In this chapter, we will extend the language introduced in the previous chapter to allow us to define and use functions.

4.1 Syntactic Structure

To use functions in a program, the language must support both function creation and function calls. To achieve this, let's extend the language's syntactic structure as shown in Figure 4.1. We have added the last two cases (function declaration and function call) to the existing language. `fun` x E is a syntax that defines a function that takes x as an argument and returns the result of evaluating E. In this case, x is called the formal parameter, and E is called the body expression of the function. The value of the argument x can only be used within the body E. A function call generally takes the form E_1 E_2. E_1 is an expression that computes the function to be called, and E_2 is an expression that computes the function's argument. The value of E_2 is called the actual parameter.

The following is an example of a program that can be written

$$
\begin{array}{lll}
E & \rightarrow & n \\
& | & x \\
& | & E_1 \texttt{ + } E_2 \\
& | & E_1 \texttt{ - } E_2 \\
& | & E_1 \texttt{ * } E_2 \\
& | & E_1 \texttt{ / } E_2 \\
& | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{iszero } E \\
& | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\
& | & \texttt{fun } x \ E \qquad\qquad \text{Function definition} \\
& | & E_1 \ E_2 \qquad\qquad \text{Function call}
\end{array}
$$

Figure 4.1: Syntax

in this language.

- `fun x (x+1)`

  This denotes a function that takes the argument `x` and returns the value of `x+1`.

- `fun x (let y = x+1 in if iszero y then x else y)`

  The function body can contain any expression. In this case, it refers to a function that uses a `let` expression to assign `x+1` to `y` and then returns the result of the conditional expression.

- `fun x (fun y (x+y))`

  This is a case where a function creation expression is used

in the function body. It refers to a function that takes an argument `x` and returns the function `fun y (x+y)`.

- `fun f (f 1)`

  This is a case where a function takes another function as an argument. It refers to a function that takes the function `f` as an argument and returns the result of calling the passed function with the first argument.

- `let f = fun x (x+1) in (f 2)`

  Just like other values, functions can be given names. This program names the function `fun x (x+1)` as `f` and then calls `f` with the actual argument `2`. The result is 3.

- `let f = fun x (x+1) in (f (f 2))`

  In the function call expression (E_1 E_2), E_1 and E_2 can be any expressions. In this example, the function call expression `(f (f 2))` occurs when E_2 is another function call expression, `(f 2)`. Since `(f 2)` is evaluated first, and the function `f` is then called again with that result as an argument, the final result is 4.

- `(fun f (f (f 2)) fun x (x+1))`

  In this example, the function expression `fun f (f (f 2))` was used directly in place of a name at the E_1 position in the function call expression. The position of E_2 also uses

the function-generating expression `fun x (x+1)`; this function is passed as the argument `f` and called twice, resulting in a value of 4.

- `let f = fun x (fun y (x+y)) in ((f 3) 4)`

  In this example, the function `f` can be interpreted as a function that takes arguments `x` and `y` and returns `x+y`. In `((f 3) 4)`, `f` is called by passing 3 and 4 to `x` and `y`, respectively, and the result is 7.

As seen in the example above, functions can be used very freely in the language shown in Figure 4.1. Functions can be stored in variables, passed as arguments to other functions, and returned as the result of other functions. It is a language that supports higher-order functions, similar to OCaml.

Note that if a programming language supports functions, the `let` expression is not required. This is because any `let` expression can be converted into a function call expression with the same meaning. For any E_1 or E_2, one can use the following equivalence relation.

$$\texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2\ \equiv\ (\texttt{fun}\ x\ E_2)\ E_1$$

For example, the program

```
let f = fun x (x+1)
```

```
in (f (f 2))
```

has the same meaning as the program below, which uses function calls.

```
(fun f (f (f 2)) fun x (x+1))
```

Although it can be expressed using other syntax, such as the `let` expression, the syntax provided by a programming language for the convenience of writing programs is called "syntactic sugar."

The function definition syntax provided by OCaml is also syntactic sugar. For example, consider the following OCaml program.

```
let square x = x * x in
  let add x y = x + y in
    (add 1 (square 2))
```

As shown below, this can be expressed solely using the syntactic structure in Figure 4.1.

```
let square = fun x (x * x) in
  let add = fun x (fun y (x + y)) in
    (add 1 (square 2))
```

4.2 Semantic Structure

Now, let's define the execution semantics of function declarations and calls.

Free Variables To define the semantics of a function, we must first classify the variables appearing in the program into free variables and bound variables. A free variable is a variable whose definition cannot be found within a given expression. Conversely, a variable whose definition can be found within a given expression is called a bound variable. The set $FV(E)$ of free variables appearing in expression E is defined inductively as follows (Let $VAR(E)$ denote the set of all variables appearing in expression E; then the set of bound variables is $VAR(E) \setminus FV(E)$).

$$
\begin{aligned}
FV(n) &= \emptyset \\
FV(x) &= \{x\} \\
FV(E_1 \texttt{ + } E_2) &= FV(E_1) \cup FV(E_2) \\
FV(\texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2) &= FV(E_1) \cup (FV(E_2) \setminus \{x\}) \\
FV(\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3) &= FV(E_1) \cup FV(E_2) \cup FV(E_3) \\
FV(\texttt{fun } x\ E) &= FV(E) \setminus \{x\} \\
FV(E_1\ E_2) &= FV(E_1) \cup FV(E_2)
\end{aligned}
$$

If a given expression consists of a single variable (x), then that variable is a free variable within expression x. The free variables in expression E_1 `+` E_2 are the set of free variables in expressions E_1 and E_2. When determining the free variables of expression `let` x `=` E_1 `in` E_2, we also combine the free variables from E_1 and E_2. However, since the variable x has been defined, x must

be excluded from the free variables of E_2. Even within expression `fun` x E, x is defined as a function argument and is therefore not a free variable. Let's look at a few examples.

- $FV($`fun y (x+y)`$) = \{$`x`$\}$
- $FV($`fun x (let y = 1 in x+y+z)`$) = \{$`z`$\}$
- $FV($`fun x (fun y (x+y))`$) = \emptyset$
- $FV($`let x = y in fun y (x+y+z)`$) = \{$`y`$,$`z`$\}$
- $FV($`let x = x in x`$) = \{x\}$

Static/Dynamic Scope The meaning of a function depends on how the values of the free variables appearing in the function are interpreted. For example, consider the following program.

```
let x = 1
in let f = fun y (x+y)
   in let x = 2
      in let g = fun y (x+y)
         in (f 1) + (g 1)
```

The meaning of the above program can be interpreted in two ways, depending on how the values of the free variables are determined.

- If we interpret the value of the free variable (`x`) appearing in the function expression `fun y (x+y)` as the value

at the time the function is defined, the result of executing the program above is 5. Since the value of `x` at the time the function `f` is defined is 1, `f` is interpreted as a function that takes `y` as an argument and increments it by 1. Therefore, the result of the function call `(f 1)` is 2. On the other hand, since the value of `x` at the time the function `g` is defined is 2, `g` is interpreted as a function that increments the value of its argument by 2. Therefore, the result of the function call `(g 1)` is 3.

- If the values of the free variables in the function are determined based on their values at the time the function is called, the result of executing the above program will be 6. Since the value of `x` is 2 at the time the final expression `(f 1) + (g 1)` is evaluated, both functions `f` and `g` are interpreted as functions that increment their argument by 2. Therefore, the results of both function calls, `(f 1)` and `(g 1)`, are 3.

If a programming language handles function calls in the first way, it supports static scoping. The second approach is called dynamic scoping.[1)]

1) Generally, in programming languages, "static" means "determined before execution." "Dynamic" means "determined after execution."

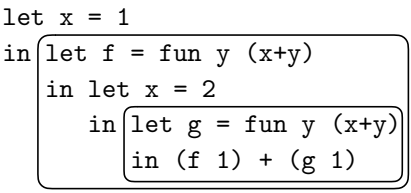


Figure 4.2: Static scope of variable x

Most programming languages support static scoping. This is because, in this approach, the scope of variables is statically determined before program execution, making the program much easier to understand. For example, when running the sample program above, the scope of variable `x` is determined before execution, regardless of the program's execution, as shown in Figure 4.2. The outer rectangle denotes the scope of `x` as defined in the first line. In other words, the free variable `x` appearing in this region refers to the `x` defined in the first line—that is, 1. The inner rectangle represents the scope of `x` defined on the third line. In other words, the `x` used in this region refers to the `x` on the third line—that is, 2—rather than the one on the first line. The scopes of other variables are also determined in this way before execution, based on the program's structure. Since the scope of every variable can be determined without running the program, it is easy to interpret the program.

On the other hand, when a program is executed using dy-

namic scope, the scope of a variable is determined at runtime. For example, when executing a program in this manner, the variable `x` defined on the third line becomes valid in the expression `x+y` on the second line; however, it is generally impossible (undecidable) to determine this fact before execution. Since it is difficult to predict a variable's scope before execution—and one must actually run the program to find out—interpreting the program becomes difficult.

4.2.1 Static Scope

Let's define the semantics of functions to support static scope. First, since functions can now be used as values in the program, let's extend the semantic space as follows.

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \mathit{Env} &= \mathit{Var} \to \mathit{Val} \end{aligned}$$

Now, in addition to integers and Boolean values, function values (*Procedure*) have been added to the set of values that can be handled by the program. The function value $(x, E, \rho) \in \mathit{Procedure}$ consists of three elements. x represents the function's formal parameters, E represents the function's body, and ρ represents the environment at the time the function is defined. The last ele-

ment, the environment ρ, is necessary to evaluate the function's free variables using the values they had at the time the function was defined. A function value defined in this way is also called a closure[2)].

Now let's define the semantics of function creation and the execution of function calls. The semantics of a function creation expression are as follows.

$$\frac{}{\rho \vdash \texttt{fun}\ x\ E \Rightarrow (x, E, \rho)}$$

This means that, given the environment ρ, evaluating the expression `fun` x E produces the function value (x, E, ρ). For static scope, the environment ρ at the time the function is defined is stored as the last element of the function value. For example, evaluating the function-creation expression `fun x x` in an empty environment ($\emptyset$) produces the function value $(\texttt{x}, \texttt{x}, \emptyset)$.

The meaning of a function call expression is as follows.

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

To execute the function call E_1 E_2, first compute E_1 and E_2, respectively, in the current environment ρ. Computing E_1 should

2) This refers to a value that contains all the information necessary to call the function.

yield the function value (x, E, ρ'). If the result of E_1 is a value other than a function value—for example, an integer—then the function call is impossible and the program's behavior is undefined (a type error occurs). In the function value (x, E, ρ'), ρ' represents the environment saved at the time the function was defined. Since it may differ from ρ—the environment at the time of the function call—it is denoted as ρ'. Next, E_2 is computed to determine the value to be passed as an argument to the function. Let's call that value v (here, v indicates that the value of E_2 can be an integer, true/false, or a function). Once E_1 and E_2 have been evaluated, the final step is to evaluate E, which is the body of the function. The key to static scope is using ρ', which represents the environment at the time the function is defined. In ρ', after expanding the formal parameter x to have the actual parameter v, evaluating the body expression E is evaluated, yielding the value v', which becomes the result of the function call expression $E_1\ E_2$.

Let's look at a few examples of execution.

Example 1 In environment $\rho = \{\texttt{y} \mapsto 2\}$, the process of evaluating the function call `(fun x (x+y)) 1` is as follows.

$$\dfrac{\rho \vdash \texttt{fun x (x+y)} \Rightarrow (\texttt{x}, \texttt{x+y}, \rho) \quad \rho \vdash \texttt{1} \Rightarrow 1 \quad \dfrac{\begin{array}{c}\{\texttt{x} \mapsto 1\}\rho \vdash \texttt{x} \Rightarrow 1 \\ \{\texttt{x} \mapsto 1\}\rho \vdash \texttt{y} \Rightarrow 2\end{array}}{\{\texttt{x} \mapsto 1\}\rho \vdash \texttt{x+y} \Rightarrow 3}}{\rho \vdash \texttt{(fun x (x+y)) 1} \Rightarrow 3}$$

Example 2 Let's examine the execution process of the program below in an empty environment.

```
let x = 1
in let f = fun y (x+y)
    in let x = 2
        in (f 3)
```

1. Executing `let x = 1 in ...` creates the following environment:

$$\rho_1 = \{\texttt{x} \mapsto 1\}$$

2. When `let f = fun y (x+y) in ...` is executed in the environment ρ_1, the current environment ρ_1 is expanded so that `f` has the function value $(\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})$. Let's call

the expanded environment ρ_2.

$$\rho_2 = \{\texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\}$$

3. If we execute `let x = 2 in ...` in the environment ρ_2, `x` now refers to the integer 2:

$$\rho_3 = \{\texttt{x} \mapsto 2, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\}$$

   Note that in the environment contained within the function value that `f` represents, `x` still refers to 1.

4. Now, using the environment ρ_3, we evaluate the function call `(f 3)`. According to the semantic structure rules, it is executed through the following steps.

   a) First, we compute the value that `f` denotes in the current environment ρ_3. We obtain the function value $(\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})$.

   b) The actual argument `3` is evaluated to obtain the integer value 3.

   c) From the function value $(\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})$, retrieve the environment $\{\texttt{x} \mapsto 1\}$ at the time the function was defined and extend it to include the passed argu-

ment. Let this be ρ_4.

$$\rho_4 = \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 3\}$$

d) In environment ρ_4, the function's body `x+y` is evaluated, yielding 4.

The entire process described above can be represented as a proof tree using inference rules as follows.

$$\dfrac{\emptyset \vdash \texttt{1} \Rightarrow 1 \qquad \dfrac{\{\texttt{x} \mapsto 1\} \vdash \texttt{fun y (x+y)} \Rightarrow (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\}) \quad \vdots}{\{\texttt{x} \mapsto 1\} \vdash \begin{array}{l}\texttt{let f = fun y (x+y)} \\ \texttt{in let x = 2} \\ \texttt{in (f 3)}\end{array} \Rightarrow 4}}{\emptyset \vdash \begin{array}{l}\texttt{let x = 1} \\ \texttt{in let f = fun y (x+y)} \\ \texttt{in let x = 2} \\ \texttt{in (f 3)}\end{array} \Rightarrow 4}$$

The portion omitted from the above process ($\vdots$) is as follows.

$$\dfrac{\{\ \texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\ \} \vdash \texttt{2} \Rightarrow 2 \quad \vdots}{\{\ \texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\ \} \vdash \texttt{let x = 2 in (f 3)} \Rightarrow 4}$$

The portion omitted from the above process ($\dot{\vdots}$) is as follows.

$$\dfrac{\left\{\begin{array}{l}\texttt{x} \mapsto 2,\\ \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\end{array}\right\} \vdash \texttt{f} \Rightarrow (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\}) \quad \left\{\begin{array}{l}\texttt{x} \mapsto 2,\\ \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\end{array}\right\} \vdash \texttt{3} \Rightarrow 3 \quad \left\{\begin{array}{l}\texttt{x} \mapsto 1\\ \texttt{y} \mapsto 3\end{array}\right\} \vdash \texttt{x+y} \Rightarrow 4}{\left\{\begin{array}{l}\texttt{x} \mapsto 2,\\ \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}, \{\texttt{x} \mapsto 1\})\end{array}\right\} \vdash \texttt{(f 3)} \Rightarrow 4}$$

4.2.2 Dynamic Scope

Supporting dynamic scope is simpler than supporting static scope. Instead of using the environment at the time the function is created, we simply use the environment provided at the time the function is called. We can modify the meaning of the function call expression that supports static scope, as defined earlier, as follows.

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

Unlike the rules for supporting static scope, when evaluating the function body, we use ρ—the environment provided at the time of the function call—instead of ρ'.

While simply changing the meaning of the function call expression in this way would suffice, it results in the function value unnecessarily storing the environment from the time the function was created. Let's simplify the definition of the meaning as follows. First, let's remove the environment from the function value and redefine it as shown below.

$$\mathit{Procedure} = \mathit{Var} \times \mathit{Exp}$$

In dynamic scope, since the environment at the time of the function call is used for the body of the function, there is no longer a need to carry the environment from when the function was created. The semantic structure of function creation and the call expression is simplified as follows.

$$\frac{}{\rho \vdash \texttt{fun}\ x\ E \Rightarrow (x, E)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E) \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

From the perspective of defining semantic structures, dynamic scope is simpler than static scope. However, understanding the execution semantics of a program has become more complex.

For example, the program from the previous example exe-

cutes as follows.

$$\dfrac{\emptyset \vdash \texttt{1} \Rightarrow 1 \qquad \dfrac{\{\texttt{x} \mapsto 1\} \vdash \texttt{fun y (x+y)} \Rightarrow (\texttt{y}, \texttt{x+y}) \quad \vdots}{\{\texttt{x} \mapsto 1\} \vdash \begin{array}{l}\texttt{let f = fun y (x+y)}\\ \texttt{in let x = 2}\\ \texttt{in (f 3)}\end{array} \Rightarrow 5}}{\emptyset \vdash \begin{array}{l}\texttt{let x = 1}\\ \texttt{in let f = fun y (x+y)}\\ \texttt{in let x = 2}\\ \texttt{in (f 3)}\end{array} \Rightarrow 5}$$

The omitted portion ($\vdots$) in the process above is as follows.

$$\dfrac{\{\ \texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y})\ \} \vdash \texttt{2} \Rightarrow 2 \quad \vdots}{\{\ \texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y})\ \} \vdash \begin{array}{l}\texttt{let x = 2}\\ \texttt{in (f 3)}\end{array} \Rightarrow 5}$$

The omitted portion ($\vdots$) in the process above is as follows.

$$\dfrac{\begin{array}{c}\{\ \texttt{x} \mapsto 2, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y})\ \} \vdash \texttt{f} \Rightarrow (\texttt{y}, \texttt{x+y})\\ \{\ \texttt{x} \mapsto 2, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y})\ \} \vdash \texttt{3} \Rightarrow 3\\ \{\ \texttt{x} \mapsto 2, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y}), \texttt{y} \mapsto 3\ \} \vdash \texttt{x+y} \Rightarrow 5\end{array}}{\left\{\ \texttt{x} \mapsto 2, \texttt{f} \mapsto (\texttt{y}, \texttt{x+y})\ \right\} \vdash \texttt{(f 3)} \Rightarrow 5}$$

This differs from the execution process based on static scope in that the environment used when evaluating the function call `(f 3)` is propagated and used until the body expression `x+y` is evaluated.

4.2.3 Recursive Functions

Consider the following program.

```
let f = fun x (f x) in (f 1)
```

Although this program is intended to be a recursive function, it does not run correctly according to the semantic structure supporting static scope defined earlier. The environment immediately after defining `f` with a `let` expression is as follows:

$$\{\texttt{f} \mapsto (\texttt{x}, \texttt{(f x)}, \emptyset)\}$$

When the function call `(f 1)` is executed in this environment, it retrieves the body expression `(f x)`—which represents the function `f`—and the saved environment $\emptyset$. Expanding the environment with respect to the argument yields $\{\texttt{x} \mapsto 1\}$, and the body expression `(f x)` is evaluated; however, since the current environment contains no information about `f`, execution cannot

proceed any further.

$$\dfrac{\{\texttt{f} \mapsto (\texttt{x}, \texttt{f x}, \emptyset)\} \vdash \texttt{f} \Rightarrow (\texttt{x}, \texttt{f x}, \emptyset) \qquad \dfrac{\{\texttt{x} \mapsto 1\} \vdash \texttt{f} \Rightarrow ? \quad \{\texttt{x} \mapsto 1\} \vdash \texttt{x} \Rightarrow 1}{\{\texttt{x} \mapsto 1\} \vdash \texttt{f x} \Rightarrow ?}}{\{\texttt{f} \mapsto (\texttt{x}, \texttt{f x}, \emptyset)\} \vdash \texttt{f 1} \Rightarrow ?}$$

Therefore, recursive functions cannot be used in the current language. Let's extend the language to support recursive functions. First, let's extend the grammar structure as follows.

$$\begin{array}{rcl} E & \rightarrow & \cdots \\ & | & \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \end{array}$$

The expression `letrec` $f(x)$ `=` E_1 `in` E_2 has been added to define recursive functions. Here, f is the function name, x is the argument, and E_1 is the function body. Just like with `let`, the defined function f can only be used within E_2. For example, the previous example can be written as follows.

```
letrec f(x) = (f x) in (f 1)
```

To define a semantic structure, let's extend the values handled by the program to include recursive functions as follows.

$$\begin{array}{rcl} \mathit{Val} & = & \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{RecProcedure} \\ \mathit{Procedure} & = & \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \mathit{RecProcedure} & = & \mathit{Var} \times \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Val} \end{array}$$

The difference between the value of a recursive function and that of a regular function is that the former additionally remembers the function name. Recursive functions also have static scope and include the environment at the time of definition in the function value.

The meaning of a recursive function expression is as follows.

$$\frac{\{f \mapsto (f, x, E_1, \rho)\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec}\ f(x)\ \texttt{=}\ E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

In the environment ρ, the function value (f, x, E_1, ρ) of the recursive function $f(x)$ `=` E_1 is computed, and then the current environment (ρ) is expanded to calculate E_2. Except for the fact that a recursive function is being defined, this is identical in meaning to a standard `let` expression.

The rules for calling a recursive function are as follows.

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \\ \{x \mapsto v, f \mapsto (f, x, E, \rho')\}\rho' \vdash E \Rightarrow v'\end{array}}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

Unlike non-recursive function calls, when a function is called, the environment ρ'—as it existed at the time the function was defined—is expanded to include both the function arguments and the function being called.

For example, the program above executes as follows. You can see that the function is called recursively an infinite number of times (only the main steps are shown).

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{\texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{f}, \texttt{x}, \texttt{f x}, \emptyset)\} \vdash \texttt{f x} \Rightarrow}}{\{\texttt{x} \mapsto 1, \texttt{f} \mapsto (\texttt{f}, \texttt{x}, \texttt{f x}, \emptyset)\} \vdash \texttt{f x} \Rightarrow}}{\{\texttt{f} \mapsto (\texttt{f}, \texttt{x}, \texttt{f x}, \emptyset)\} \vdash \texttt{f 1} \Rightarrow}}{\emptyset \vdash \texttt{letrec f(x) = (f x) in (f 1)} \Rightarrow}$$

We can now write a program that runs forever using recursive functions. The execution of a never-ending program corresponds to the process of constructing a proof tree of infinite size, as shown above.

The syntactic and semantic structures of a language that includes recursive functions are summarized in Figures 4.3 and 4.4. Now, based on the definition of the semantic structure, there are two execution rules for the function call expression $E_1\ E_2$. The rule to be applied is selected based on the result of evaluating E_1.

For reference, recursive functions work well in dynamic scope without any special extensions. Let's consider the program below again.

```
let f = fun x (f x) in (f 1)
```

$$
\begin{array}{rcl}
E & \rightarrow & n \\
& | & x \\
& | & E_1 \texttt{ + } E_2 \\
& | & E_1 \texttt{ - } E_2 \\
& | & E_1 \texttt{ * } E_2 \\
& | & E_1 \texttt{ / } E_2 \\
& | & \texttt{iszero } E \\
& | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\
& | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{fun } x\ E \\
& | & E_1\ E_2
\end{array}
$$

Figure 4.3: Syntactic structure of a language that includes recursive functions

Execution semantics of function calls using dynamic scope

$$\frac{\rho \vdash E_1 \Rightarrow (x, E) \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

If we run the program according to this, it will run indefinitely as follows.

$$\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{\{\texttt{f} \mapsto (\texttt{x}, \texttt{f x}), \texttt{x} \mapsto 1\} \vdash \texttt{f x} \Rightarrow}}{\{\texttt{f} \mapsto (\texttt{x}, \texttt{f x}), \texttt{x} \mapsto 1\} \vdash \texttt{f x} \Rightarrow}}{\{\texttt{f} \mapsto (\texttt{x}, \texttt{f x})\} \vdash \texttt{f 1} \Rightarrow}}{\emptyset \vdash \texttt{let f = fun (x) (f x) in (f 1)} \Rightarrow}$$

In dynamic scope, the body of a function is evaluated in the en-

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{true}} \qquad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{false}}\ n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \{x \mapsto v_1\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

$$\frac{\{f \mapsto (f, x, E_1, \rho)\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

$$\frac{}{\rho \vdash \texttt{fun } x\ E \Rightarrow (x, E, \rho)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \\ \{x \mapsto v, f \mapsto (f, x, E, \rho')\}\rho' \vdash E \Rightarrow v'\end{array}}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

Figure 4.4: Semantic structure of a language that includes recursive functions

vironment at the time the function call occurs, and that environment already contains information about the function `f`.

4.3 Implementation

Let's implement the interpreter for the language defined in this chapter. The target language defined in OCaml is as follows.

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string
```

For example, the program below

```
let f = fun x (x+1)
in f (f 1)
```

is represented as follows.

```
  LET ("f", PROC ("x", ADD (VAR "x", CONST 1)),
    CALL (VAR "f", CALL (VAR "f", CONST 1)))
```

Values and environments can be defined as follows.

```
type value = Int of int | Bool of bool
           | Procedure of var * exp * env
           | RecProcedure of var * var * exp * env
and env = var -> value

let empty_env =
  fun x -> raise (Failure ("env is empty"))
let extend_env (x,v) e =
  fun y -> if x = y then v else (e y)
let apply_env e x = e x
```

Above, we defined the environment as a function, exactly as in its mathematical definition. Since an empty environment contains no information, it is defined as a function that raises an exception for any variable. The value of variable `x` in environment `e` can be obtained through a function call (`e x`). The environment obtained by extending environment `e` so that variable `x` takes the value `v` is given by the following function.

```
  fun y -> if x = y then v else (e y)
```

This means that it takes a variable `y` and, if `y` matches `x`, returns `v`; otherwise, it retrieves the value from the previous environment `e`. If you are not familiar with this implementation, you can use the environment implemented with lists from the previous chapter.

Let's implement the interpreter

```
eval : exp -> env -> value
```

for the programming language defined in this chapter. Let's implement and test both cases: one using static scope and one using dynamic scope.

5

Functional Language Fun

Let's extend the language we've developed so far to make it more robust. Using the language from the previous chapter as a basis, let's design and implement a functional language, Fun, that supports units (`()`), true/false (`true`, `false`), lists, and mutually recursive functions.

5.1 Syntactic Structure

The syntax of Fun is shown in Figure 5.1. For example, the function that reverses a list, as expressed in OCaml,

```
let rec reverse l =
  match l with
  | [] -> []
  | hd::tl -> (reverse tl)@[hd]
in reverse [1;2;3]
```

can be written in Fun as follows.

```
letrec reverse(l) =
  if (isnil l) then nil
  else (reverse (tail l)) @ ((head l)::nil)
in reverse 1::2::3::nil
```

We can define and use mutual recursive functions as follows.

```
letrec even(x) = if (x = 0) then true  else odd(x-1)
    and odd(x) = if (x = 0) then false else even(x-1)
in (even 9)
```

The function `even` returns true if its argument `x` is even, and false if it is odd. The function `odd` returns true if its argument is odd. The two functions are defined recursively so that each calls the other. For simplicity, we have restricted the syntax structure shown in Figure 5.1 so that only these two functions can be defined recursively.

5.2 Semantic Structure

The domain is as shown in Figure 5.2. The values are units ($\cdot$), integers ($\mathbb{Z}$), Boolean values (*Bool*), lists (*List*), functions (*Procedure*), recursive functions (*RecProcedure*), and mutual recursive functions (*MRecProcedure*). In the definition of the semantic space, Val^* denotes the set of lists whose elements are values. Let us denote a list as $[a_1, a_2, \ldots, a_n]$ and an empty list as []. For the value v and the list s, let $v :: s$ denote the list obtained by appending v to s. Given the two lists s_1 and s_2, $s_1 @ s_2$ denotes the list formed by concatenating s_1 and s_2. The environment (*Env*) is, as before, a function that maps variables to values.

$$\rho \in Env = Var \to Val$$

$$
\begin{array}{lcl}
P & \rightarrow & E \\
E & \rightarrow & \texttt{()} \\
 & | & \texttt{true} \mid \texttt{false} \\
 & | & n \\
 & | & x \\
 & | & E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \\
 & | & E \texttt{ = } E \mid E \texttt{ < } E \\
 & | & \texttt{not } E \\
 & | & \texttt{nil} \\
 & | & E \texttt{ :: } E \\
 & | & E \texttt{ @ } E \\
 & | & \texttt{head } E \\
 & | & \texttt{tail } E \\
 & | & \texttt{isnil } E \\
 & | & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & | & \texttt{let } x \texttt{ = } E \texttt{ in } E \\
 & | & \texttt{letrec } f(x) \texttt{ = } E \texttt{ in } E \\
 & | & \texttt{letrec } f(x_1) = E_1 \texttt{ and } g(x_2) = E_2 \texttt{ in } E \\
 & | & \texttt{fun } x \ E \\
 & | & E \ E \\
 & | & \texttt{print } E \\
 & | & E; E
\end{array}
$$

Figure 5.1: Fun's Syntax Structure

Since, by the syntax, mutually recursive functions are always defined together, their values are defined as follows.

$$(\mathit{Var} \times \mathit{Var} \times \mathit{Exp}) \times (\mathit{Var} \times \mathit{Var} \times \mathit{Exp}) \times \mathit{Env}$$

$$
\begin{array}{rcl}
v \in \mathit{Val} & = & \{\cdot\} + \mathbb{Z} + \mathit{Bool} + \mathit{List} + \\
 & & \mathit{Procedure} + \mathit{RecProcedure} + \mathit{MRecProcedure} \\
n \in \mathbb{Z} & = & \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
b \in \mathit{Bool} & = & \{\mathit{true}, \mathit{false}\} \\
s \in \mathit{List} & = & \mathit{Val}^* \\
\mathit{Procedure} & = & \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\
\mathit{RecProcedure} & = & \mathit{Var} \times \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\
\mathit{MRecProcedure} & = & (\mathit{Var} \times \mathit{Var} \times \mathit{Exp}) \times (\mathit{Var} \times \mathit{Var} \times \mathit{Exp}) \times \mathit{Env}
\end{array}
$$

Figure 5.2: Fun's Semantic Space

It includes the names, arguments, and body expressions of both functions and is designed to remember the environment at the time of definition (the environment is shared by both functions). For example, when `even` and `odd` are defined as above, the following values are produced.

$$
\begin{array}{l}
((\texttt{even}, \texttt{x}, \texttt{if (x=0) then true else odd(x-1)}), \\
\ (\texttt{odd}, \texttt{x}, \texttt{if (x=0) then false else even(x-1)}), \emptyset)
\end{array}
$$

Let's define the semantic structure rules one by one. Expressions denoting constants produce default values as follows.

$$\frac{}{\rho \vdash \texttt{()} \Rightarrow \cdot}$$

$$\frac{}{\rho \vdash \texttt{true} \Rightarrow \mathit{true}}$$

$$\frac{}{\rho \vdash \texttt{false} \Rightarrow \mathit{false}}$$

$$\frac{}{\rho \vdash n \Rightarrow n}$$

A variable refers to the value that variable holds in the environment.

$$\frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

The meanings of arithmetic operators are as follows.

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ - } E_2 \Rightarrow n_1 - n_2}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ * } E_2 \Rightarrow n_1 * n_2}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ / } E_2 \Rightarrow n_1/n_2} \; n_2 \neq 0$$

Let's define the meanings of comparison operators (`=`, `<`) as follows.

$$\frac{\rho \vdash E_1 \Rightarrow b_1 \qquad \rho \vdash E_2 \Rightarrow b_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{true}} \; b_1 = b_2$$

$$\frac{\rho \vdash E_1 \Rightarrow b_1 \qquad \rho \vdash E_2 \Rightarrow b_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{false}} \; b_1 \neq b_2$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{true}} \; n_1 = n_2$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{false}} \; n_1 \neq n_2$$

$$\frac{\rho \vdash E_1 \Rightarrow [a_1, \ldots, a_n] \quad \rho \vdash E_2 \Rightarrow [b_1, \ldots, b_m]}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{true}} \; n = m \wedge \forall i.a_i = b_i$$

$$\frac{\rho \vdash E_1 \Rightarrow [a_1, \ldots, a_n] \quad \rho \vdash E_2 \Rightarrow [b_1, \ldots, b_m]}{\rho \vdash E_1 \texttt{ = } E_2 \Rightarrow \mathit{false}} \; n \neq m \vee \exists i.a_i \neq b_i$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ < } E_2 \Rightarrow \mathit{true}} \; n_1 < n_2$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ < } E_2 \Rightarrow \mathit{false}} \; n_1 \geq n_2$$

In the definition of the semantic structure above, when comparing whether two values are equal (E_1 `=` E_2), only values that are integers, Booleans, or lists were considered. In the case of lists, they were defined as equal if their lengths are the same and all their elements are identical.[1)] Comparing function values is impossible, so it was not considered. This is because a function value contains a program (the body of the function), and determining whether two programs have the same meaning is generally an undecidable problem. In OCaml as well, comparing two functions as shown below causes an exception.

```
# (fun x -> x) = (fun y -> y);;
Exception: Invalid_argument "compare: functional value".
```

1) In fact, for lists, the condition under which two values are equal has not been strictly defined, leaving room for interpretation. For example, the above semantic definition does not clearly specify how to interpret cases where a list's elements contain other lists. In implementation, the meaning will depend on how the operator = is defined.

The meaning of the logical operator `not` is as follows.

$$\frac{\rho \vdash E \Rightarrow \mathit{true}}{\rho \vdash \texttt{not}\ E \Rightarrow \mathit{false}}$$

$$\frac{\rho \vdash E \Rightarrow \mathit{false}}{\rho \vdash \texttt{not}\ E \Rightarrow \mathit{true}}$$

Lists are created in the following three ways.

$$\frac{}{\rho \vdash \texttt{nil} \Rightarrow []}$$

$$\frac{\rho \vdash E_1 \Rightarrow v \qquad \rho \vdash E_2 \Rightarrow s}{\rho \vdash E_1\ \texttt{::}\ E_2 \Rightarrow v :: s}$$

$$\frac{\rho \vdash E_1 \Rightarrow s_1 \qquad \rho \vdash E_2 \Rightarrow s_2}{\rho \vdash E_1\ \texttt{@}\ E_2 \Rightarrow s_1 @ s_2}$$

In `Fun`, an empty list is represented by `nil`. The operators `::` and `@` are identical to those in OCaml. In the above definitions, v and s represent arbitrary values and list values, respectively. The meanings of other list operators are as follows.

$$\frac{\rho \vdash E \Rightarrow v :: s}{\rho \vdash \texttt{head}\ E \Rightarrow v} \qquad \frac{\rho \vdash E \Rightarrow v :: s}{\rho \vdash \texttt{tail}\ E \Rightarrow s}$$

$$\frac{\rho \vdash E \Rightarrow []}{\rho \vdash \texttt{isnil}\ E \Rightarrow \mathit{true}} \qquad \frac{\rho \vdash E \Rightarrow v :: s}{\rho \vdash \texttt{isnil}\ E \Rightarrow \mathit{false}}$$

The meanings of other expressions are as defined in the previous

chapter. The meaning of a conditional expression is as follows.

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v}$$

The meaning of a 'let' expression is as follows.

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \{x \mapsto v_1\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

$$\frac{\{f \mapsto (f, x, E_1, \rho)\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

The meanings of function definitions and calls are as follows.

$$\frac{}{\rho \vdash \texttt{fun}\ x\ E \Rightarrow (x, E, \rho)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

$$\frac{\begin{array}{c}\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \\ \{x \mapsto v, f \mapsto (f, x, E, \rho')\}\rho' \vdash E \Rightarrow v'\end{array}}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

A semantic structure is also needed to define and call mutually recursive functions. First, a mutually recursive function is con-

structed as follows.

$$\frac{\{f \mapsto (f, x, E_1, g, y, E_2, \rho), g \mapsto (g, y, E_2, f, x, E_1, \rho)\}\rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{and}\ g(y) = E_2\ \texttt{in}\ E_3 \Rightarrow v}$$

The function value $(f, x, E_1, g, y, E_2, \rho)$ denotes the value of the function f and implies that it is defined mutually recursively with respect to the function g. The value of the function g is denoted as $(g, y, E_2, f, x, E_1, \rho)$. The rule for calling a mutually recursive function is as follows.

$$\frac{\rho \vdash E_1 \Rightarrow (f, x, E_f, g, y, E_g, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \left\{\begin{array}{l} x \mapsto v, \\ f \mapsto (f, x, E_f, g, y, E_g, \rho'), \\ g \mapsto (g, y, E_g, f, x, E_f, \rho') \end{array}\right\} \rho' \vdash E_f \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

The key point is that when evaluating the body expression E_f of the called function (f), the environment must also be extended for the mutually defined function g. Only then can g be called from within E_f.

The expression `print` E calculates the value of E and outputs it to the screen.

$$\frac{}{\rho \vdash \texttt{print}\ E \Rightarrow \cdot}$$

Typically, semantic structure rules do not express interactions with the external environment, such as input and output. Therefore, we have only expressed that calculating the expression `print` E generates a unit value. Finally, the expression $E_1; E_2$ instructs the system to calculate the two expressions E_1 and E_2 in sequence.

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash E_1; E_2 \Rightarrow v_2}$$

Although expression E_1 is evaluated, its result v_1 is ignored.

5.3 Implementation

Let's implement the grammatical and semantic structures defined so far in OCaml. First, the syntactic structure can be defined using the following data type.

```
type program = exp
and exp =
  | UNIT
  | TRUE
  | FALSE
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | EQUAL of exp * exp
  | LESS of exp * exp
```

```
  | NOT of exp
  | NIL
  | CONS of exp * exp
  | APPEND of exp * exp
  | HEAD of exp
  | TAIL of exp
  | ISNIL of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | LETMREC of (var * var * exp) *
                (var * var * exp) * exp
  | PROC of var * exp
  | CALL of exp * exp
  | PRINT of exp
  | SEQ of exp * exp
and var = string
```

Let's define the value and the environment as follows.

```
type value =
  | Unit
  | Int of int
  | Bool of bool
  | List of value list
  | Procedure of var * exp * env
  | RecProcedure of var * var * exp * env
  | MRecProcedure of var * var * exp *
                     var * var * exp * env
and env = (var * value) list
```

Let's implement the following function that takes a program as

input and computes its value.

```
run : program -> value
```

Here are a few examples of execution. The mutual recursive function defined earlier

```
letrec even(x) = if (x = 0) then true  else odd(x-1)
    and odd(x) = if (x = 0) then false else even(x-1)
in (even 8)
```

is represented by the following OCaml data type.

```
  LETMREC
   (("even", "x",
     IF (EQUAL (VAR "x", CONST 0), TRUE,
      CALL (VAR "odd", SUB (VAR "x", CONST 1)))),
   ("odd", "x",
    IF (EQUAL (VAR "x", CONST 0), FALSE,
     CALL (VAR "even", SUB (VAR "x", CONST 1)))),
   CALL (VAR "even", CONST 8))
```

When the program above is executed with `run`, `Bool true` should be computed.

In the program below, we define a factorial function and have it output the factorials of numbers from 1 to 10 in reverse order.

```
letrec factorial(x) =
         if (x = 0) then 1
         else factorial(x-1) * x
in letrec loop n =
```

```
    if (n = 0) then ()
    else (print (factorial n); loop (n-1))
  in (loop 10)
```

The program written in OCaml is as follows:

```
LETREC ("factorial", "x",
 IF (EQUAL (VAR "x", CONST 0), CONST 1,
  MUL (CALL (VAR "factorial",
           SUB (VAR "x", CONST 1)), VAR "x")),
 LETREC ("loop", "n",
  IF (EQUAL (VAR "n", CONST 0), UNIT,
   SEQ (PRINT (CALL (VAR "factorial", VAR "n")),
    CALL (VAR "loop", SUB (VAR "n", CONST 1)))),
  CALL (VAR "loop", CONST 10)))
```

When executed, the following output appears on the screen. The final value calculated is `Unit`.

```
3628800
362880
40320
5040
720
120
24
6
2
1
```

A program that creates a list consisting of integers from 1 to 10 can be written as follows.

```
letrec range(n) =
      if (n = 1) then (1::nil)
      else n::(range (n-1))
in (range 10)
```

In OCaml, it can be expressed as follows:

```
  LETREC ("range", "n",
   IF (EQUAL (VAR "n", CONST 1), CONS (CONST 1, NIL),
    CONS (VAR "n",
      CALL (VAR "range", SUB (VAR "n", CONST 1)))),
   CALL (VAR "range", CONST 10))
```

When executed, the list value `List [Int 10; Int 9; ...; Int 1]` is generated.

A function that reverses a list

```
letrec reverse(l) =
  if (isnil l) then nil
  else (reverse (tail l)) @ ((head l)::nil)
in reverse 1::2::3::nil
```

can be defined in OCaml as follows:

```
  LETREC ("reverse", "l",
   IF (ISNIL (VAR "l"), NIL,
    APPEND (CALL (VAR "reverse", TAIL (VAR "l")),
     CONS (HEAD (VAR "l"), NIL))),
   CALL (VAR "reverse",
    CONS (CONST 1, CONS (CONST 2, CONS (CONST 3, NIL)))))
```

When executed, it produces the list value `List [Int 3; Int 2; Int 1]`.

6

State Changes

The language we have designed so far was a purely functional language that does not support state changes. In such a language, program execution was purely a process of evaluating values. In this chapter, let's extend the language to allow for changes in values—a key feature of imperative languages.

Until now, once a variable was defined, it was impossible to change its value. For example, suppose we want to count the number of times the function `f` is called while executing the program below.

```
let f = fun x (x)
in (f (f 1))
```

We could try the following, but it won't work.

```
let counter = 0
in let f = fun x (let counter = counter + 1 in x)
   in let a = (f (f 1))
      in counter
```

To count the number of times the function is called, we introduced the variable `counter` and redefined the function so that

the value of `counter` is incremented by one each time the function body is evaluated. This program calls the function twice on the third line and then calculates the value of `counter` at the end. Although the intended final value is 2, the actual result is 0 (regardless of static or dynamic scope). This is because redefining the value of `counter` within the function body does not change the value of the variable `counter` defined on the first line.

To make something like this possible, in addition to the environment, we need memory—a device where values can be changed. Let's extend the language to support memory and allow values to change. The ways in which programming languages support changes in values can be broadly classified into two categories based on how they access and use memory.

6.1 First Approach

The first category consists of languages that support memory allocation, access, and modification through explicit syntax. This is the approach used in languages such as OCaml and F#.[1)]

1) Although not introduced earlier, OCaml also supports changes in memory state using the same syntax as in this section.

6.1.1 Syntactic Structure

Let's extend the language from the previous chapter as shown in Figure 6.1. We have added four new syntaxes.

- `ref` E is an instruction that assigns a new address in memory. If the newly allocated address is l, it calculates the value of E, stores it in l, and then returns the address l.
- `!`E refers to the value stored at the memory address denoted by E.
- $E_1 := E_2$ changes the value stored at the memory address pointed to by E_1 to the value represented by E_2.
- $E_1; E_2$ is a statement that executes E_1 and E_2 in sequence.

For example, in this language, you can write a program to count the number of function calls as shown below.

```
let cnt = ref 0
in let f = fun x (cnt := !cnt + 1; !cnt)
   in let a = (f 0)
      in let b = (f 0)
         in a + b
```

To store the number of calls, memory is allocated and initialized to 0, and the variable `cnt` is set to point to that memory address. This variable, `cnt`, is called a reference variable. Each time the function `f` is called, the value stored at the memory

E	$\rightarrow$	n	
	$\mid$	x	
	$\mid$	E_1 `+` E_2	
	$\mid$	`let` x `=` E_1 `in` E_2	
	$\mid$	`iszero` E	
	$\mid$	`if` E_1 `then` E_2 `else` E_3	
	$\mid$	`fun` x E	
	$\mid$	E_1 E_2	
	$\mid$	`ref` E	Memory allocation
	$\mid$	`!`E	Memory access
	$\mid$	$E_1 := E_2$	Memory modification
	$\mid$	$E_1; E_2$	Sequential statement

Figure 6.1: Syntax

address pointed to by `cnt` is incremented by 1, and the current value (`!cnt`) is returned as the function's result. Therefore, `a` and `b` become 1 and 2, respectively, and ultimately, the value of `a+b` is calculated as 3.

6.1.2 Semantic Structure

To define the semantic structure, let's extend the semantic space as follows.

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{Loc} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Val} \\ \sigma \in \mathit{Mem} &= \mathit{Loc} \to \mathit{Val} \end{aligned}$$

Now, memory addresses can be created during program execution, and memory addresses can be used as values. Let *Loc* be the set of all memory addresses. We will use notations such as Memory is defined as a finite function that maps addresses to values. Characters such as $\sigma, \sigma', \sigma_1$ will be used to refer to memory. The definitions of functions and environments remain unchanged from before. In this language, a variable still refers to a name attached to a value. What has changed is that a memory address can also be a value, and a modifiable value is stored at that address in memory.

Now, executing a program involves not only calculating values but also changes in memory state. When a program is executed in the current environment and memory, the result is not only the calculated values but also the modified memory. Let's

express this as follows.

$$\rho, \sigma \vdash E \Rightarrow v, \sigma'$$

Given environment ρ and memory σ, executing program E results in the calculation of value v and means that the input memory σ changes to the output memory σ'.

Except for the four newly added constructs, the semantic structure of the previous language can be simply extended to account for memory changes (Figure 6.2). For each expression, the memory changes that may occur during computation are indicated, and these changes are continuously accumulated. For example, consider the case of the four basic arithmetic operations.

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2, \sigma_2}$$

To compute expression E_1 `+` E_2, we first use the given memory σ_0 to compute expression E_1 and record the memory changes that occurred during that process in σ_1. Next, expression E_2 is computed using memory σ_1, and the changes that occur while computing expression E_2 are stored in σ_2. As a result, σ_2 contains all the changes that occurred while calculating expression E_1 `+` E_2.

Let's define the meanings of the newly added constructs.

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \rho(x), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero } E \Rightarrow \mathit{true}, \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow n, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero } E \Rightarrow \mathit{false}, \sigma_1} \; n \neq 0$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{true}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{false}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_3 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \{x \mapsto v_1\}\rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v, \sigma_2}$$

$$\frac{}{\rho, \sigma \vdash \texttt{fun } x \; E \Rightarrow (x, E, \rho), \sigma}$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ \{x \mapsto v\}\rho', \sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\rho, \sigma_0 \vdash E_1 \; E_2 \Rightarrow v', \sigma_3}$$

Figure 6.2: Semantic structure extended to include memory changes

- First, the meaning of a memory allocation statement is as follows.

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{ref } E \Rightarrow l, \{l \mapsto v\}\sigma_1} \; l \notin \mathrm{Dom}(\sigma_1)$$

To execute the assignment statement `ref` E, first calcu-

late the value of E using the given environment (ρ) and memory (σ_0). Let's say the value of E is v, and let's call the modified memory resulting from the calculation of E σ_1. Next, a new address is allocated at memory location σ_1. Here, the condition $l \notin \mathrm{Dom}(\sigma_1)$ means that address l is a new address that has not been assigned to memory σ_1. This means that the new address l is newly assigned to memory σ_1. Next, `ref` E returns the assigned address value l and modifies the value at address l within memory σ_1 ($\{l \mapsto v\}\sigma_1$). The initially given memory location σ_0 is changed to E while calculating the expression σ_1, and is finally changed to $\{l \mapsto v\}\sigma_1$.

- The second is a statement that accesses the value stored at a memory address.

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma_0 \vdash \texttt{!}E \Rightarrow \sigma_1(l), \sigma_1}$$

First, the value of expression E is calculated. At this point, the result of E must be an address value (l). If calculating expression E yields a value of a different type, the result is undefined. For example, `!(1+2)` or `!(fun` x $(x+1)$) are well-defined programs according to the syntax, but they have no meaning. Suppose the expression E computes the

address value l, and the memory change is reflected in σ_1. Then, $!E$ outputs $\sigma_1(l)$—the value held at memory address l, which is $\sigma_1(l)$. Since the operation of reading a memory address does not itself cause a memory change, the final output memory address is σ_1.

- The meaning of an assignment statement, which changes the value at a memory address, is as follows.

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow l, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash E_1 := E_2 \Rightarrow v, \{l \mapsto v\}\sigma_2}$$

  First, we must be able to calculate E_1 to obtain the address value l. Assume that the memory change is reflected in σ_1. Next, execute E_2 using the current environment ρ and the memory state σ_1 resulting from the execution of E_1. Let's call the result v and the resulting memory σ_2. The assignment statement returns the value v and sets the address l of memory σ_2 to the value v.

- $E_1; E_2$ means to evaluate the two expressions in sequence.

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

  The expression E_1 is evaluated first, and then E_2 is evaluated, accumulating changes in memory. At this point, the

value v_1 of E_1 is ignored. The role of E_1 is simply to cause a change in memory.

Let's examine how the program below executes according to the semantic structure. The initial environment and memory are both empty ($\emptyset$ and $\emptyset$, respectively).

```
let cnt = ref 0
in let f = fun x (cnt := !cnt + 1; !cnt)
   in let a = (f 0)
      in let b = (f 0)
         in a + b
```

1. First, to process the first `let`, the expression `ref 0` is evaluated. According to the semantic definition of memory allocation, it executes as follows.

$$\frac{\emptyset, \emptyset \vdash 0 \Rightarrow 0, \emptyset}{\emptyset, \emptyset \vdash \texttt{ref 0} \Rightarrow l_1, \{l_1 \mapsto 0\}} \; l_1 \notin \mathrm{Dom}(\emptyset)$$

   A new address l_1 was allocated, and 0 was stored at that address. The input memory $\emptyset$ was updated to the output memory $\{l_1 \mapsto 0\}$, and the address l_1 was returned as the value. Based on this result, the environment and memory after executing `let cnt = ...` are as follows.

$$\begin{aligned} \rho_1 &: \{\texttt{cnt} \mapsto l_1\} \\ \sigma_1 &: \{l_1 \mapsto 0\} \end{aligned}$$

After the first `let`, the variable `cnt` refers to the address l_1, which means that the value 0 is stored at that address in memory.

2. Now, using the environment ρ_1 and memory σ_1, we evaluate the second `let` expression. To do this, we first evaluate the function expression

   ```
   fun x (cnt := !cnt + 1; !cnt)
   ```

   By evaluating this, we obtain the following function value:

   $$(\texttt{x}, \texttt{(cnt:=!cnt+1;!cnt)}, \{\texttt{cnt} \mapsto l_1\})$$

   We assign the name `f` to the resulting function value and update the environment. Let's call the environment and memory at this point ρ_2 and σ_2, respectively.

   $$\rho_2 : \{\texttt{f} \mapsto (\texttt{x}, \texttt{(cnt:=!cnt+1;!cnt)}, \{\texttt{cnt} \mapsto l_1\}), \texttt{cnt} \mapsto l_1\}$$
   $$\sigma_2 : \{l_1 \mapsto 0\}$$

3. The third `let` expression is evaluated using the environments ρ_2 and σ_2. To do this, the function call `(f 0)` is first evaluated, which results in the calculation of the function

body

```
(cnt:=!cnt+1;!cnt)
```

. The environment used at this point is $\{\texttt{x} \mapsto 0, \texttt{cnt} \mapsto l_1\}$—the environment obtained by expanding the function arguments from $\{\texttt{cnt} \mapsto l_1\}$, which is the environment stored in the function value—while the memory $\{l_1 \mapsto 0\}$, which is the memory at the time of the call, is used as-is. Let's refer to these as ρ_3, σ_3, respectively.

$$\begin{aligned} \rho_3 &: \{\texttt{x} \mapsto 0, \texttt{cnt} \mapsto l_1\} \\ \sigma_3 &: \{l_1 \mapsto 0\} \end{aligned}$$

Let's evaluate the function body in this environment and memory. First, the assignment statement `cnt:=!cnt+1` is evaluated as follows.

$$\dfrac{\dfrac{}{\rho_3, \sigma_3 \vdash \texttt{cnt} \Rightarrow l_1, \sigma_3} \quad \dfrac{\dfrac{\dfrac{}{\rho_3, \sigma_3 \vdash \texttt{cnt} \Rightarrow l_1, \sigma_3}}{\rho_3, \sigma_3 \vdash \texttt{!cnt} \Rightarrow 0, \sigma_3} \quad \dfrac{}{\rho_3, \sigma_3 \vdash \texttt{1} \Rightarrow 1, \sigma_3}}{\rho_3, \sigma_3 \vdash \texttt{!cnt+1} \Rightarrow 1, \sigma_3}}{\rho_3, \sigma_3 \vdash \texttt{cnt:=!cnt+1} \Rightarrow 1, \{l_1 \mapsto 1\}\sigma_3}$$

To compute `cnt:=!cnt+1`, we first look up the value of `cnt` in the environment ρ_3. Its address is l_1. Next, `!cnt` is evaluated. It is 0, the value currently stored at memory address l_1. Next, `!cnt+1` is calculated, resulting in 1. This

value is stored in memory at the address l_1, which corresponds to `cnt`. The memory state changes as follows:

$$\sigma_4 \quad : \quad \{l_1 \mapsto 1\}\sigma_3 = \{l_1 \mapsto 1\}$$

Next, in environment ρ_3 and memory σ_4, the result of evaluating `!cnt` is 1. Finally, the environment is updated so that the variable `a` in the current environment holds the value 1.

$$\{\texttt{a} \mapsto 1, \texttt{f} \mapsto (\texttt{x}, \texttt{(cnt:=!cnt+1;!cnt)}, \{\texttt{cnt} \mapsto l_1\}), \texttt{cnt} \mapsto l_1\}$$

Let's call this environment ρ_4.

4. The function call on the fourth line is evaluated in the environment ρ_4 and memory σ_4. If you follow a process similar to the previous step, the memory will change as follows.

$$\sigma_5 \quad : \quad \{l_1 \mapsto 2\}$$

The environment becomes $\{\texttt{b} \mapsto 2\}\rho_4$, containing the variable `b`.

5. The value of the expression `a+b`, calculated in the final environment and memory, is 3.

$$
\begin{array}{rcll}
E & \to & n & \\
 & | & x & \\
 & | & E_1 \texttt{ + } E_2 & \\
 & | & \texttt{iszero } E & \\
 & | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 & \\
 & | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 & \\
 & | & \texttt{fun } x\ E & \\
 & | & E_1\ E_2 & \\
 & | & x \texttt{ := } E & \text{Assignment statement} \\
 & | & E_1; E_2 & \text{Sequential expression}
\end{array}
$$

Figure 6.3: Syntax

6.2 Second Approach

In languages such as C, Java, and Python, memory allocation and access for value changes are not explicitly revealed. Let's support state changes using the methods employed in these languages.

6.2.1 Syntactic Structure

The syntax structure is as shown in Figure 6.3. Only assignment statements and sequence expressions have been newly added to the existing language. The assignment statement x `:=` E changes the value of the variable x to the value of E. The sequence expression $E_1; E_2$ means that the two expressions E_1 and E_2 are evaluated in sequence.

$$\frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash x \texttt{ := } E \Rightarrow v, \{\rho(x) \mapsto v\}\sigma_1}$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \\ \{x \mapsto l\}\rho, \{l \mapsto v_1\}\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2\end{array}}{\rho, \sigma_0 \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v, \sigma_2} \; l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\rho, \sigma_0 \vdash E_1 \; E_2 \Rightarrow v', \sigma_3} \; l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

Figure 6.4: Semantic Structure

6.2.2 Semantic Structure

Let's define the semantic space as follows:

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Loc} \\ \sigma \in \mathit{Mem} &= \mathit{Loc} \to \mathit{Val} \end{aligned}$$

In this case as well, memory is a function that maps addresses to values. Let's call the set of memory addresses $\mathit{Loc} = \{l_1, l_2, \ldots\}$ and the set of all possible memory locations Mem ($\mathit{Mem} = \mathit{Loc} \to$

Val). Unlike before, the environment is defined as a function that maps variables to memory addresses ($Env = Var \to Loc$). Therefore, what a variable signifies in a program is no longer a value but a memory address. The value of a variable refers to the value stored at the memory address that the variable denotes. Since the values stored at each address in memory are modifiable, the values of all variables ultimately became modifiable. The values that can be stored in memory are integers, booleans, and function values. In this language, memory addresses are not included in the values themselves. This is because, unlike the previous language, memory addresses cannot be created directly, nor can they be accessed directly.

Just as in the first approach, program execution not only calculates values but also alters the state of memory. In other words, when a program is executed in the current environment and memory, the result is not only the calculated value but also the modified memory.

$$\rho, \sigma \vdash E \Rightarrow v, \sigma'$$

The definition of semantics is as shown in Figure 6.4. When we omit the same syntactic elements that were used in previous languages to explicitly express state changes, the differences are

the following five points.

- Variables:

$$\frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

  Now, to read the value of a variable, we must refer to both the environment and memory. Since the environment is a function that maps variables to addresses, $\rho(x)$ refers to the memory address denoted by the variable x. Since memory is a function that maps addresses to values, $\sigma(\rho(x))$ refers to the value stored at the memory address denoted by the variable x.

- Introduction to College Admissions:

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash x \texttt{ := } E \Rightarrow v, \{\rho(x) \mapsto v\}\sigma_1}$$

  The value stored at the memory address denoted by the variable x is changed to the value of the expression E. Let the value of the expression E, the value calculated in the current environment and memory, and the result memory be denoted by v, σ_1, respectively. The assignment statement x `:=` E is defined to calculate the value v and to change the value at the address $(\rho(x))$ denoted by x in memory σ_1 to v $(\{\rho(x) \mapsto v\}\sigma_1)$.

Note that when a variable appears on the left-hand side of an assignment statement, it refers to the address it points to ($\rho(x)$). This differs from the previous rule, where a variable in an expression referred to the value stored at that address. Thus, in languages that follow the second approach (e.g., C, Java), the meaning of a variable changes depending on where it is used. The value when a variable appears on the left side of an assignment statement (l-value) differs from the value when it appears on the right side (r-value). For example, in `x := x`, the `x` on the left refers to the address, while the `x` on the right refers to the value stored at that address.

- `let` expression:

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \{x \mapsto l\}\rho, \{l \mapsto v_1\}\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}\ l \notin \mathrm{Dom}(\sigma_1)$$

As in the previous language, the role of the `let` expression is to define a new variable and extend the current scope. The difference is that variables now represent memory addresses rather than values. First, we calculate the value of E_1 to obtain the value v_1 and the memory address σ_1.

Next, a new memory address l is allocated from the current memory σ_1 for the variable x ($l \notin \mathrm{Dom}(\sigma_1)$). Next, the current environment is expanded so that the variable x refers to address l ($\{x \mapsto l\}\rho$). Expand the memory so that v_1, the value of E_1, is stored at l, the memory address referred to by the variable x ($\{l \mapsto v_1\}\sigma_1$). Finally, calculate the expression E_2 in the expanded environment and memory.

- Function call:

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

Compute expression E_1 to calculate the closure (x, E, ρ') of the function to be called and the memory σ_1. Next, σ_1 is used to compute E_2, which in turn computes the value v to be passed as an argument to the function, and updates the memory to σ_2. Now, the function body E is evaluated; before that, a new memory address l is allocated for the formal parameter x. In the environment ρ', the variable x is extended to have the address l, and the current memory σ_2 is also extended so that the argument value v is stored

at address l. The result of evaluating the body E in this extended environment and memory becomes the result of the function call.

Let's execute the program below according to its semantic structure.

```
let f = let cnt = 0
        in fun (x) (cnt := cnt + 1; cnt)
in let a = (f 0)
   in let b = (f 0)
      in a + b
```

1. First, starting from an empty environment ($\emptyset$) and empty memory ($\emptyset$), we execute the first `let f = ...`. To do this, we must first compute the value of the expression below.

   ```
   let cnt = 0 in fun (x) (cnt := cnt + 1; cnt)
   ```

   After defining the value of the variable `cnt`, the environment and memory are as follows.

   $$\begin{aligned} \rho_1 &= \{\texttt{cnt} \mapsto l_1\} \\ \sigma_1 &= \{l_1 \mapsto 0\} \end{aligned}$$

   The address l_1 was allocated in memory for the variable `cnt`, and the value 0 was stored at that address. In this environment and memory state, evaluating the function ex-

pression

```
fun (x) (cnt := cnt + 1; cnt)
```

produces the function value

$$(\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \{\texttt{cnt} \mapsto l_1\})$$

without any changes to memory. Now, to process `let f = ...` , a new memory address l_2 is allocated. Then, in the environment, `f` is set to point to l_2, and l_2 is set to hold the function value. As a result, the environment and memory are updated as follows.

$$\begin{aligned} \rho_2 &= \{\texttt{f} \mapsto l_2\} \\ \sigma_2 &= \{l_1 \mapsto 0, l_2 \mapsto (\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \{\texttt{cnt} \mapsto l_1\})\} \end{aligned}$$

Note that the current environment ρ_2 contains no information about the variable `cnt`. Since `cnt` exists only in the environment stored in the function value, it has become a variable that is accessible only within the function.

2. Now, we evaluate `let a = (f 0)` in environment ρ_2 and memory σ_2. To evaluate the function call, we first allocate a new address l_3 in memory σ_2 and store the argument value 0 at that address. Therefore, the environment and

memory state when evaluating the function body expression `(cnt:=cnt+1;cnt)` are as follows.

$$\begin{aligned}\rho_3 &= \{\texttt{x} \mapsto l_3, \texttt{cnt} \mapsto l_1\} \\ \sigma_3 &= \{l_1 \mapsto 0, l_2 \mapsto (\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \rho_1), l_3 \mapsto 0\}\end{aligned}$$

When the first expression in the function body—the assignment statement (`cnt:=cnt+1`)—is evaluated, the memory changes as shown below.

$$\{l_1 \mapsto 1, l_2 \mapsto (\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \rho_1), l_3 \mapsto 0\}$$

Since the second expression in the body is `cnt`, the environment retrieves the address l_1, which the variable `cnt` refers to, and fetches the value 1 stored at address l_1 in memory. The environment after the function call is as follows (l_4 is the new address allocated to define the variable `a`).

$$\{\texttt{f} \mapsto l_3, \texttt{a} \mapsto l_4\}$$

Memory is as follows.

$$\{l_1 \mapsto 1, l_2 \mapsto (\texttt{x}, \texttt{(cnt:=cnt+1;cnt)}, \rho_1), l_3 \mapsto 0, l_4 \mapsto 1\}$$

3. Similarly, the second `let` expression can be evaluated, and

as a result, the value of `b` becomes 2. Therefore, the value of `a+b` becomes 3.

6.2.3 Function Call Method

As programming languages allow changes to memory state, it becomes possible to call functions by reference (call-by-reference).

Call-by-Value

Up to this point, we have been calling functions by value (call-by-value). Let's revisit the meaning of a function call as defined in Section 6.2.

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

When a function is called to evaluate the body expression E, the value (v) of the expression (E_2) provided as the actual argument is stored at the memory address (l) denoted by the function's formal parameter (x). This method of passing function arguments is called "call-by-value." The key point here is that when passing a function argument, new memory is allocated to store the argument's value, and the value is copied and stored there. Because a new address is created to pass the value, even if

the value of the argument is modified within the function body, the value of the argument at the point of the function call remains unchanged. For example, consider the program below.

```
let f = fun (x) (x := 1; 1)
in let a = 2
   in let b = (f a)
        in a + b
```

The function `f` is a function that changes the value of the argument `x` to 1 and then returns 1. The function `f` is being called with the argument `a`; the value of `a` is passed, and within the function, `x` points to a new memory address where the passed value is stored. Even if the value of `x` is changed to 1 within the function body, only the value stored at the newly allocated memory address is modified; the value of `a` at the call site remains unchanged. Therefore, the value of `a + b` is 3.

Call-by-Reference

In an address-passing call, instead of allocating a new address and copying the argument value, the address where the argument value is stored is passed. For example, if the function `f` in the example above is called using an address-passing call, the address of `a` is passed as an argument, and the argument `x` refers to the same address as `a`. Therefore, modifying `x` within the function body changes the value of `a`.

To support address-passing calls in addition to value-passing calls, let's extend the language's syntax as follows.

$$
\begin{array}{rcll}
E & \rightarrow & \ldots & \\
 & | & E_1\ E_2 & \text{Value-passing call} \\
 & | & E\ \langle y \rangle & \text{Address-passing call}
\end{array}
$$

To distinguish it from a value-passing call ($E_1\ E_2$), let's use a different syntax ($E\ \langle y \rangle$) when referring to an address-passing call. An address-passing call works only when a variable is passed as an argument. It must be possible to pass an address as an argument, because in our current language, only variables represent addresses. Therefore, within the language's syntactic structure, we have enforced that only variables—and not arbitrary expressions—can be passed as function arguments when making an address-passing call. For example, to call the function `f` using an address-passing call in the example program above, write the program as follows.

```
let f = fun (x) (x := 1; 1)
in let a = 2
   in let b = (f <a>)
       in a + b
```

The meaning of a call by reference is as follows.

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \{x \mapsto \rho(y)\}\rho', \sigma_1 \vdash E \Rightarrow v', \sigma_2}{\rho, \sigma_0 \vdash E_1\ \langle y \rangle \Rightarrow v', \sigma_2}$$

It differs from a call by value in that, at the time of the call, the formal parameter x of the function is modified to point to the address represented by the variable passed as an argument. At the call site, only the address of the variable is passed, without the need to calculate the actual value of the argument.

When a programming language supports address-passing calls, it becomes possible for two different variables to refer to the same address. For example, consider the following program.

```
let a = 1
in let f = fun (x) (fun (y) (y := 2; x + y))
   in ((f <a>) <a>)
```

Within the function body, the variables `x`, `y`, and `a` all refer to the same memory address. Therefore, if the value of `y` is changed by the assignment statement `y := 2`, the values referred to by `x` and `a` also become 2. Due to the call by reference, the variables `x`, `y`, and `a` have effectively become aliases that all refer to the same memory address. When aliases are introduced into a program, it becomes difficult to predict the program's execution. Even if it appears that the value of a certain variable is be-

ing changed, the value of another variable that seems completely unrelated may actually be altered; therefore, it is necessary to examine all possible scenarios. When considering which assignment statements affect the values of which variables, it used to be sufficient to consider only the variables being defined, but now it has become much more complex because we must also consider that the values of variables that appear unrelated may also change.

Lazy Evaluation

For reference, on the other end of the spectrum of function calling styles is a method called lazy evaluation, which defers the calculation of function arguments as much as possible. The two calling conventions examined earlier were both "eager evaluation," in which arguments are evaluated before the function is called. In contrast, with lazy evaluation, function arguments are evaluated only when they are actually used. Therefore, if an argument is not used within the function body, its value is not evaluated. For example, consider the program below.

```
letrec forever(x) = (forever x)  (* infinite loop *)
in (fun x (1)) (forever 0)
```

The function, which returns 1 regardless of the argument's value, was called with the argument `(forever 0)`. If the argument value

is evaluated before calling the function `f`, the program above will never terminate. However, when the program is executed using lazy evaluation, the body of the called function—`1`—is executed without first evaluating the argument `(forever 0)`. Since the function body does not use the argument `x`, the expression `(forever 0)` is not evaluated, and the program terminates. Thus, a program that does not terminate under eager evaluation may terminate under lazy evaluation.

Lazy evaluation is primarily used in functional languages such as Haskell but is rarely used in imperative languages. This is because the timing of when argument values are evaluated varies dynamically, making it difficult to predict the order in which program statements will be executed. In particular, in imperative languages where the meaning of a program depends on the order of evaluation, this can increase the program's complexity.

6.3 Implementation

Let's implement the interpreter for the language defined in this chapter.

First Approach

First, we can define the language for the first approach in OCaml as follows.

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
  | NEWREF of exp (* memory allocation *)
  | DEREF of exp  (* dereference *)
  | SETREF of exp * exp (* assignment *)
  | SEQ of exp * exp (* sequence *)
and var = string
```

Let's implement values, the environment, and memory as follows.

```
type value =
    Int of int
  | Bool of bool
  | Loc of loc
  | Closure of var * exp * env
  | RecClosure of var * var * exp * env
and env = var -> value
and loc = int
and mem = loc -> value

let empty_env =
```

```
  fun x -> raise (Failure "Environment is empty")
let extend_env (x,v) e =
  fun y -> if x = y then v else (e y)
let apply_env e x = e x

let empty_mem =
  fun _ -> raise (Failure "Memory is empty")
let extend_mem (l,v) m =
  fun y -> if l = y then v else (m y)
let apply_mem m l = m l

let counter = ref 0
let new_location () = counter:=!counter+1;!counter
```

We've represented memory addresses as integers and designed the function so that each call to `new_location()` returns a new memory address. Let's implement an interpreter for the language described above, which has the function type shown below.

```
eval : exp -> env -> mem -> value * mem
```

The Second Approach

The language in the second approach can be defined as follows.

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
```

```
  | DIV of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp    (* call-by-value *)
  | CALLREF of exp * var (* call-by-reference *)
  | SET of var * exp
  | SEQ of exp * exp
and var = string
```

Let's implement values, the environment, and memory as follows.

```
type value =
    Int of int
  | Bool of bool
  | Closure of var * exp * env
  | RecClosure of var * var * exp * env
and env = var -> loc
and loc = int
and mem = loc -> value

let empty_env =
  fun x -> raise (Failure "Environment is empty")
let extend_env (x,v) e =
  fun y -> if x = y then v else (e y)
let apply_env e x = e x

let empty_mem =
  fun _ -> raise (Failure "Memory is empty")
let extend_mem (l,v) m =
  fun y -> if l = y then v else (m y)
```

```
let apply_mem m l = m l

let counter = ref 0
let new_location () = counter:=!counter+1;!counter
```

Let's implement an interpreter for the above language that has the function types shown below.

```
eval : exp -> env -> mem -> value * mem
```

7

Pointers and Memory Management

In this chapter, we will extend the language to support pointers and records and to enable memory management. Let's consider the language shown in Figure 7.1. It is a language that allows the modification of variable values and supports function calls based on both value passing and address passing. The semantic space is as follows.

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} \\ \mathit{Procedure} &= \mathit{Var} \times E \times \mathit{Env} \\ \rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Loc} \\ \sigma \in \mathit{Mem} &= \mathit{Loc} \to \mathit{Val} \end{aligned}$$

The main execution rules are as shown in Figure 7.2.

E	$\rightarrow$	n	integer
	$\mid$	x	variable
	$\mid$	E `+` E	arithmetic operation
	$\mid$	`iszero` E	test
	$\mid$	`if` E_1 `then` E_2 `else` E_3	conditional expression
	$\mid$	`let` $x = E_1$ `in` E_2	variable definition
	$\mid$	`fun` x E	Function definition
	$\mid$	E_1 E_2	Value-passing function call
	$\mid$	E $\langle y \rangle$	Address-passing function call
	$\mid$	x `:=` E	Assignment statement
	$\mid$	$E_1; E_2$	Sequential statement

Figure 7.1: Grammatical structure of the target language

7.1 Records

In imperative languages, a record[1)] refers to a collection of named memory addresses. For example, consider the following program.

```
let person = { age := 2, birth := 201803 }
in  person.age + person.birth
```

A record `person` is defined as a collection of two memory addresses named `age` and `birth`. Record fields can be accessed as `person.age`.

For reference, in an imperative language, an array can be understood as a record in which memory addresses are labeled with natural numbers. For example, consider the following example.

1) In C, this is called a "structure."

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \{x \mapsto l\}\rho, \{l \mapsto v_1\}\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}\ l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{}{\rho, \sigma \vdash \texttt{fun}\ x\ E \Rightarrow (x, E, \rho), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash x\ \texttt{:=}\ E \Rightarrow v, \{\rho(x) \mapsto v\}\sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \quad \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \{x \mapsto \rho(y)\}\rho', \sigma_1 \vdash E \Rightarrow v', \sigma_2}{\rho, \sigma_0 \vdash E_1\ \langle y \rangle \Rightarrow v', \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

Figure 7.2: Semantic structure of the target language

```
let arr[3] = { 1, 2, 3 }
in  arr[0] + arr[1] + arr[2]
```

The array `arr` is a collection of three memory addresses, and each address is accessed using a natural number index, such as `arr[0]`, `arr[1]`, and `arr[2]`.

$$
\begin{array}{rcll}
E & \rightarrow & \vdots & \\
& | & \{\} & \text{Empty record} \\
& | & \{x \texttt{ := } E_1, y \texttt{ := } E_2\} & \text{Record} \\
& | & E.x & \text{Field access} \\
& | & E_1.x \texttt{ := } E_2 & \text{Field value modification}
\end{array}
$$

Figure 7.3: Extending the Syntax Structure for Records

Syntax

To add records, let's extend the language's grammatical structure as shown in Figure 7.3. We have added the grammar for creating and using records. Although a record can have any number of fields, for the sake of simplicity in defining the semantic structure, let's consider only cases where there are either no fields or exactly two fields. $E.x$ denotes the value of the record field x, which is defined by E. $E_1.x$:= E_2 is a record assignment statement that changes the value stored at the memory address denoted by the record field $E_1.x$ to the value of E_2.

Semantic Structure

Let's extend the semantic space as follows. Since records can now serve as values, we have included records in the set of val-

ues.

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{Record} \\ \mathit{Procedure} &= \mathit{Var} \times E \times \mathit{Env} \\ r \in \mathit{Record} &= \mathit{Field} \to \mathit{Loc} \\ \rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Loc} \\ \sigma \in \mathit{Mem} &= \mathit{Loc} \to \mathit{Val} \end{aligned}$$

Since a record (r) is a collection of named memory addresses, we defined it as a function that maps field names to addresses. In other words, a record value looks like this:

$$\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}$$

It is a function that maps each record field x_i to the memory address l_i.

The definition of the semantic structure is shown in Figure 7.4. In the first rule, the empty record {} produces a function with no definitions—that is, the empty record value $\emptyset$.

The second rule generates the value of a record $\{x := E_1, y := E_2\}$ that contains fields. First, the values of the expressions E_1 and E_2 are calculated to produce the values v_1 and v_2. If the subsequent memory is denoted as σ_2, l_1, l_2—which contains the memory addresses where the values of each field in σ_2 will be stored—

$$\frac{}{\rho, \sigma \vdash \{\} \Rightarrow \emptyset, \sigma}$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2 \quad l_1, l_2 \notin \mathrm{Dom}(\sigma_2)}{\rho, \sigma \vdash \{x \texttt{ := } E_1, y \texttt{ := } E_2\} \Rightarrow \{ \begin{array}{l} x \mapsto l_1, \\ y \mapsto l_2 \end{array} \}, \{l_1 \mapsto v_1, l_2 \mapsto v_2\}\sigma_2}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow r, \sigma_1}{\rho, \sigma \vdash E.x \Rightarrow \sigma_1(r(x)), \sigma_1}$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow r, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma \vdash E_1.x \texttt{ := } E_2 \Rightarrow v, \{r(x) \mapsto v\}\sigma_2}$$

Figure 7.4: Extension of the semantic structure for records

is newly allocated ($l_1, l_2 \notin \mathrm{Dom}(\sigma_2)$). Finally, the record value $\{x \mapsto l_1, y \mapsto l_2\}$ is generated, and the value v_1, v_2 is stored at address l_1, l_2 in memory σ_2.

The third rule defines how to retrieve the value represented by a record's field ($E.x$). First, E is calculated to obtain the record value r. To retrieve the value of field x, first locate $r(x)$, which is the address represented by field x. Next, the value stored at that address in current memory is retrieved ($\sigma_1(r(x))$).

The last rule is for updating record field values. First, calculating E_1 should yield the record value (r). Next, calculate the value v of E_2, and store the address $r(x)$ of the record field x in the current memory σ_2 so that the address $r(x)$ of the record field x points to the value v. The value of the expression $E_1.x$:= E_2 is defined as v.

For example, let's look at how the program below runs according to semantic structure rules. It is a program that uses records to represent a tree structure.

```
let tree = { left := {}, v := 1, right := {} }
in  tree.right := { left := {}, v := 2, right := 3 }
```

First, the record constructor in an empty environment and with empty memory

```
{ left := {}, v := 1, right := {} }
```

When this is calculated, the following record values are generated, and

$$\{\texttt{left} \mapsto l_1, \texttt{v} \mapsto l_2, \texttt{right} \mapsto l_3\}$$

The memory state changes as follows.

$$\{l_1 \mapsto \emptyset, l_2 \mapsto 1, l_3 \mapsto \emptyset\}$$

A new memory address l_1, l_2, l_3 was assigned, and record values were generated so that each field points to it. The memory addresses l_1 and l_3 store empty record values ($\emptyset$), and the address l_2, which corresponds to the field `v`, stores the integer value 1. Since the variable `tree` was defined as the record value r, after executing `let tree = ...`, the environment and memory are as

follows.

$$\begin{array}{rcl} \rho & = & \{\texttt{tree} \mapsto l_4\} \\ \sigma & = & \{l_1 \mapsto \emptyset, l_2 \mapsto 1, l_3 \mapsto \emptyset, l_4 \mapsto \left\{ \begin{array}{rcl} \texttt{left} & \mapsto & l_1, \\ \texttt{v} & \mapsto & l_2, \\ \texttt{right} & \mapsto & l_3 \end{array} \right\} \end{array}$$

In this environment, the variable `tree` refers to the memory address l_4, and the memory address l_4 contains a record value. Each field of the record corresponds to the address l_1, l_2, l_3, and each of those addresses contains the value $\emptyset, 1, \emptyset$. In this environment and memory, the record assignment statement

```
tree.right := { left := {}, v := 2, right := 3 }
```

Let's calculate this. To do so, we must first calculate the memory address represented by `tree.right`. When we look up the value represented by the variable `tree` in the current environment and memory, the record $\{\texttt{left} \mapsto l_1, \texttt{v} \mapsto l_2, \texttt{right} \mapsto l_3\}$ is calculated. At the address l_3—which is the address represented by the field `right` in this record—the record creation expression

```
{ left := {}, v := 2, right := 3 }
```

Simply store the value calculated from this expression. When you evaluate the record creation expression above, a record value

is generated along with a new memory address, as shown below.

$$\{\texttt{left} \mapsto l_5, \texttt{v} \mapsto l_6, \texttt{right} \mapsto l_7\}$$

This value is returned as the result of the entire program, and the environment and memory state after execution are as follows.

$$\begin{aligned} \rho &= \{\texttt{tree} \mapsto l_4\} \\ \sigma &= \left\{ \begin{array}{l} l_1 \mapsto \emptyset, \\ l_2 \mapsto 1, \\ l_3 \mapsto \{\texttt{left} \mapsto l_5, \texttt{v} \mapsto l_6, \texttt{right} \mapsto l_7\}, \\ l_4 \mapsto \{\texttt{left} \mapsto l_1, \texttt{v} \mapsto l_2, \texttt{right} \mapsto l_3\}, \\ l_5 \mapsto \emptyset, \\ l_6 \mapsto 2, \\ l_7 \mapsto 3 \end{array} \right\} \end{aligned}$$

A tree structure, created by following the links between memory addresses, was constructed in memory and can now be accessed via the variable `tree`.

7.2 Pointers

In an imperative language, a pointer variable is a variable whose value is a memory address. To support pointers, a programming

language must provide syntax for generating address values and for referencing the values of pointer variables.

For example, let's look at the programs below.

- ```
  let x = 1 in
    let y = &x in
      *y := *y + 2
  ```

  The variable `y` is defined as a pointer to the address of the variable `x` (`&x`). When the pointer variable is referenced (`*y`), it refers to the value stored at the memory address pointed to by `y`—that is, the value stored in the variable `x`. Therefore, the value of `*y + 2` is 3. The assignment statement `*y := *y + 2` changes the value stored at the address pointed to by the pointer `y` to 3. Therefore, after executing the program above, the value of `x` will change to 3.

- ```
  let x = { left := {}, v := 1, right := {} } in
    let y = &x.v
      *y := *y + 2
  ```

  The variable `y` is defined as a pointer to the address of the structure field `x.v`. When the last statement (`*y := *y + 2`) is executed, the value of `x.v` changes to 3.

- ```
  let f = fun (x) (*x := *x + 1) in
    let a = 1 in
      (f &a); a
  ```

  The argument `x` of function `f` is a pointer variable, and the function body increments the value stored at the address
  ```

pointed to by `x` by 1. Since the initial value of variable `a` is 1 and the address of `a` is passed as an argument to function `f`, the final value of `a` becomes 2.

- ```
  let f = fun (x) (&x) in
    let p = (f 1) in
      *p + 2
  ```

  The function `f` returns the address of the argument `x`. When the function call `f 1` is executed, the address of the argument `x` is returned, and the pointer `p` comes to point to that address. Therefore, the result of executing the program is 3.[2)]

When a programming language supports pointers in this way, memory addresses become first-class values that can be used freely within the program. In other words, memory addresses can be stored in variables, passed as arguments to functions, and returned from functions.

2) We have assumed that inaccessible memory is not automatically released when the function returns.
  ```

Syntax

To support pointers, let's extend the language as follows.

$$
\begin{array}{rcl}
E & \rightarrow & \cdots \\
& | & \texttt{new}\ E \\
& | & \texttt{\&}x \\
& | & \texttt{\&}E\texttt{.}x \\
& | & \texttt{*}E \\
& | & \texttt{*}E_1\ \texttt{:=}\ E_2
\end{array}
$$

The first three statements are expressions that create memory addresses. `new` E allocates a new memory address and stores E at that address. `&`x and `&`E`.`x retrieve the memory address from its name. `*`E and `*`E_1 `:=` E_2 retrieve the value stored at a memory address or modify the value stored there.

Semantic Structure

The semantic space is as follows. The address value (*Loc*) is included in the value.

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{Record} + \mathit{Loc} \\ \mathit{Procedure} &= \mathit{Var} \times E \times \mathit{Env} \\ r \in \mathit{Record} &= \mathit{Field} \to \mathit{Loc} \\ \rho \in \mathit{Env} &= \mathit{Var} \to \mathit{Loc} \\ \sigma \in \mathit{Mem} &= \mathit{Loc} \to \mathit{Val} \end{aligned}$$

The definition of the semantic structure is as follows.

$$\frac{\rho, \sigma \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma \vdash \texttt{new}\ E \Rightarrow l, \{l \mapsto v\}\sigma_1}\ l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{}{\rho, \sigma \vdash \texttt{\&}x \Rightarrow \rho(x), \sigma}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow r, \sigma_1}{\rho, \sigma \vdash \texttt{\&}E.x \Rightarrow r(x), \sigma_1}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma \vdash \texttt{*}E \Rightarrow \sigma_1(l), \sigma_1}$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow l, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma \vdash \texttt{*}E_1\ \texttt{:=}\ E_2 \Rightarrow v, \{l \mapsto v\}\sigma_2}$$

The first rule allocates a new address l in memory and stores the value of the expression E at that address. The value of the memory assignment expression `new` E is the newly allocated address

l. The second and third rules retrieve the memory addresses represented by variables (x) and record fields ($E.x$). For variables, these addresses are found in the environment; for record fields, they are found in the record value. The fourth rule defines how to retrieve the value stored at a memory address when E represents that address. First, calculating E should yield the address l, and then the value of l should be retrieved from memory. The final rule specifies that the value stored at the memory address represented by E_1 is changed to the value of E_2. When E_1 is evaluated, it should yield the address l, and and it changes the value at memory location σ_2 from l to v, which is the value of E_2.

In the semantic structure above, the meaning of the pointer reference $*E$ varies depending on where it is used. The final rule applies when the reference expression $*E$ is used on the left-hand side of an assignment statement. In this case, $*E$ refers to the address value denoted by E. The fourth rule applies when $*E$ itself is used as an expression. In this case, $*E$ refers to the value stored at the address denoted by E. Even if an expression looks identical ($*E$), its meaning may differ depending on where it is used.

Example 1 Let's examine the execution process of the following program.

```
let x = 1 in
  let y = &x in
    *y := *y + 2
```

If we denote the environment and memory after executing the two `let` expressions as $\rho = \{\texttt{x} \mapsto l_1, \texttt{y} \mapsto l_2\}$ and $\sigma = \{l_1 \mapsto 1, l_2 \mapsto l_1\}$, respectively, the process of executing the final assignment statement is as follows.

$$\dfrac{\rho, \sigma \vdash \texttt{y} \Rightarrow l_1, \sigma \qquad \dfrac{\dfrac{\rho, \sigma \vdash \texttt{y} \Rightarrow l_1, \sigma}{\rho, \sigma \vdash \texttt{*y} \Rightarrow 1, \sigma} \quad \rho, \sigma \vdash 2 \Rightarrow 2, \sigma}{\rho, \sigma \vdash \texttt{*y + 2} \Rightarrow 3, \sigma}}{\rho, \sigma \vdash \texttt{*y := *y + 2} \Rightarrow 3, \{l_1 \mapsto 3\}\sigma}$$

Example 2 Let's examine the execution process of the following program.

```
let f = proc (x) (*x := *x + 1) in
  let a = 1 in
    (f &a)
```

Let's denote the environment and memory after executing the two `let` expressions as $\rho = \{\texttt{f} \mapsto l_1, \texttt{a} \mapsto l_2\}$ and $\sigma = \{l_1 \mapsto (\texttt{x}, \texttt{*x:=*x+1}, \emptyset), l_2 \mapsto 1\}$. If we make a function call `(f &a)` here, the function value and arguments are computed as follows:

$$\dfrac{}{\rho, \sigma \vdash \texttt{f} \Rightarrow (\texttt{x}, \texttt{*x:=*x+1}, \emptyset), \sigma} \qquad \dfrac{}{\rho, \sigma \vdash \texttt{\&a} \Rightarrow l_2, \sigma}$$

After expanding the environment $\emptyset$ and memory σ with respect to the arguments as follows,

$$\rho = \{\texttt{x} \mapsto l_3\}, \qquad \sigma = \{l_3 \mapsto l_2, l_1 \mapsto (\texttt{x}, \texttt{*x:=*x+1}, \emptyset), l_2 \mapsto 1\}$$

the function body is evaluated as follows.

$$\dfrac{\dfrac{\rho, \sigma \vdash \texttt{x} \Rightarrow l_2, \sigma}{\rho, \sigma \vdash \texttt{*x} \Rightarrow 1, \sigma} \quad \rho, \sigma \vdash \texttt{1} \Rightarrow 1, \sigma}{\rho, \sigma \vdash \texttt{x} \Rightarrow l_2, \sigma \qquad \rho, \sigma \vdash \texttt{*x+1} \Rightarrow 2, \sigma}$$
$$\rho, \sigma \vdash \texttt{*x:=*x+1} \Rightarrow 2, \{l_2 \mapsto 2\}\sigma$$

7.3 Memory Management

Looking at the semantic structure defined so far, there are four cases where new memory is allocated. A new memory address is allocated each time a `let` expression, a function call, a record creation, or a memory allocation expression is evaluated (Figure 7.5).

However, based on the semantic structure defined so far, there are no cases where memory is allocated and then deallocated during program execution. Therefore, there will come a point when memory becomes full and the program can no longer run. For example, if you write a recursive function as shown below, the program will not run indefinitely but will instead terminate

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \{x \mapsto l\}\rho, \{l \mapsto v_1\}\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v, \sigma_2} \; l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \quad \{x \mapsto l\}\rho', \{l \mapsto v\}\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1 \; E_2 \Rightarrow v', \sigma_3} \; l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\rho, \sigma \vdash E_1 \Rightarrow v_1, \sigma_1 \quad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2 \quad l_1, l_2 \notin \mathrm{Dom}(\sigma_2)}{\rho, \sigma \vdash \{x \texttt{ := } E_1, y \texttt{ := } E_2\} \Rightarrow \{ \begin{array}{c} x \mapsto l_1, \\ y \mapsto l_2 \end{array} \}, \{l_1 \mapsto v_1, l_2 \mapsto v_2\}\sigma_2}$$

$$\frac{\rho, \sigma \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma \vdash \texttt{new } E \Rightarrow l, \{l \mapsto v\}\sigma_1} \; l \notin \mathrm{Dom}(\sigma_1)$$

Figure 7.5: Execution rules that consume memory

abnormally due to a memory shortage.[3)]

```
letrec forever(x) = (forever x) in (forever (new 0))
```

Let's extend the language to allow memory addresses to be recycled. Most programming languages adopt one of the two methods described below.

1. Manual memory recycling: This leaves memory management to the programmer. This is typical of system programming languages (C, C++). Since programmers manage memory directly in these languages, they can manage

3) Although recursive functions were not considered in the language definition in this chapter, they can be easily extended.

it more precisely according to their needs. However, manually managing memory is very difficult and can cause various other problems.

2. Automatic Memory Recycling: Languages such as Java automatically support memory recycling. The programmer simply allocates memory, and the programming language automatically deallocates it. An advantage is that errors caused by memory management mistakes do not occur. However, it can be less efficient than manual management, and since memory recycling must be performed during program execution, it imposes a burden.

7.3.1 Manual Memory Reclamation

Manual memory reclamation is simple. The language simply needs to support a memory deallocation statement.

$$\begin{array}{rcl} E & \rightarrow & \cdots \\ & | & \texttt{free}\ E \end{array}$$

`free` E is an instruction that deletes the address represented by E from current memory. Its meaning can be defined as follows.

$$\frac{\rho, \sigma \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma \vdash \texttt{free}\ E \Rightarrow l, \sigma_1 \setminus l}\ l \in \mathrm{Dom}(\sigma_1)$$

Here, $\sigma \setminus l$ refers to the memory resulting from deleting the entry at address l from memory σ. If we call the result σ', then σ' is the memory that satisfies the following two conditions: 1) $\mathrm{Dom}(\sigma') = \mathrm{Dom}(\sigma) \setminus \{l\}$ (where $A \setminus B$ denotes the set difference between A and B), 2) $\forall l \in \mathrm{Dom}(\sigma') : \sigma'(l) = \sigma(l)$. The value of the memory deallocation statement was defined as address l; however, since this value will no longer be used, it can actually be defined as any value.

Manual memory recycling is very simple, as shown here. However, it is difficult for a programmer to handle all memory management directly. Three problems may arise.

- Memory may be freed too late or not at all. This is called a memory leak.
- Memory may be freed before it is actually used (a "use-after-free" or "dangling pointer" error).
- The same block of memory may be freed multiple times (double-free).

These three errors are very common in programs written in C/C++—languages that support manual memory management—and are a major cause of software security vulnerabilities.

7.3.2 Automatic Memory Recycling

Let's consider how a programming language could support automatic memory recycling. The following approach comes to mind.

1. When memory runs low, the program pauses briefly.
2. It collects all the memory addresses in the current memory that will no longer be used.
3. It deletes all the addresses collected in this way from the current memory.

For example, consider the program below.

```
let f = fun (x) (x+1) in
  let a = f 0 in
    a + 1
```

Suppose the environment and memory are as follows immediately before evaluating the body of the second `let` (`a + 1`).

$$\rho = \{\texttt{f} \mapsto l_1, \texttt{a} \mapsto l_3\}, \sigma = \{l_1 \mapsto (\texttt{x}, \texttt{x+1}, \emptyset), l_2 \mapsto 0, l_3 \mapsto 1\}$$

At this point, among the addresses in the current memory, the only one that will be used in the future is l_3, which is referenced by the variable `a`. Therefore, we can reuse memory by removing l_1, l_2 from σ to obtain $\sigma' = \{l_3 \mapsto 1\}$, and then evaluate the expression `a + 1`.

Although the principle of automatic memory reuse is this simple, it is impossible to implement the above method in general-purpose programming languages. This is because the second step—accurately identifying only those addresses that will not be used in the future—is undecidable. If such an algorithm existed, it could be used to solve the halting problem. The halting problem is the problem of determining whether a given program, when executed, will terminate in finite time or run indefinitely; it is a classic example of a problem for which no algorithm can exist and which therefore cannot be solved exactly by a computer. Suppose there exists an algorithm that can identify, without omission, all memory addresses that will not be used from the current point onward, and let's call it G. Using algorithm G, we can construct algorithm H to solve the halting problem as follows.

1. H takes the input program P and constructs a new program as follows.

   `let x = new() in` P`; x`

   This program allocates memory for the variable `x`, executes the program P, and then returns `x`. Assume that the variable `x` is a new variable that does not appear in P.

2. While executing the above program, just before executing P, execute G to collect all memory addresses that will not be used in the future. Let's call that set S.

3. Now, we can determine whether P has terminated as follows.

    - If S does not contain the address of `x`, then P has ended. This is because `x` is used only after P has ended.
    - If S contains the address of `x`, it means that P does not end. Only then will `x` not be used.

Since the algorithm H for solving the halting problem should not exist, we can see that the assumption that algorithm G exists was incorrect.

For problems that are computationally intractable, the only option is to use approximation methods. Therefore, automatic memory reclamation techniques give up on perfectly identifying memory addresses that will not be used in the future and instead resort to approximation. The most commonly used approach is to approximate the set of memory addresses that will not be used in the future as the set of memory addresses that are inaccessible from the current environment. Let's look at the previous example again.

```
let f = fun (x) (x+1) in
```

```
let a = f 0 in
  a + 1
```

Immediately before computing the expression (`a + 1`), the environment and memory are as follows:

$$\rho = \{\texttt{f} \mapsto l_1, \texttt{a} \mapsto l_3\}, \sigma = \{l_1 \mapsto (\texttt{x}, \texttt{x+1}, \emptyset), l_2 \mapsto 0, l_3 \mapsto 1\}$$

In the current environment, the variables `f` and `a` are defined and point to the memory addresses l_1 and l_3, respectively. Therefore, the accessible address in the current environment is l_1, l_3, and the inaccessible address is l_2. We remove the inaccessible address l_2 from the current memory, reuse the memory as $\sigma' = \{l_1 \mapsto (\texttt{x}, \texttt{x+1}, \emptyset), l_3 \mapsto 1\}$, and then compute the expression `a + 1`.

The above approximation method is sound. This is because the set of addresses that are inaccessible from the current environment ($\{l_2\}$ in the example above) is a subset of the set of addresses that will not be used in the future ($\{l_1, l_2\}$ in the example above). It is sound in the sense that addresses to be used in the future will never be reused. However, this method is not complete. There may be addresses that are accessible from the current environment but will not be used in the future. In the example above, the address l_1 is such a case.

When identifying memory addresses accessible from the current environment, all link structures created within memory must be taken into account. For example, let's try reusing memory using the method described above in the following environment (ρ) and memory (σ).

$$\rho = \left\{ \begin{array}{l} \mathtt{x} \mapsto l_1 \\ \mathtt{y} \mapsto l_2 \end{array} \right\} \qquad \sigma = \left\{ \begin{array}{l} l_1 \mapsto 0 \\ l_2 \mapsto \{\mathtt{a} \mapsto l_3, \mathtt{b} \mapsto l_1\} \\ l_3 \mapsto l_4 \\ l_4 \mapsto (\mathtt{x}, E, \{\mathtt{z} \mapsto l_5\}) \\ l_5 \mapsto 0 \\ l_6 \mapsto l_7 \\ l_7 \mapsto l_6 \end{array} \right\}$$

First, collect the addresses $\{l_1, l_2\}$ that are directly accessible in the environment. The value stored at l_2 in memory is record $\{\mathtt{a} \mapsto l_3, \mathtt{b} \mapsto l_1\}$, so through the current environment and the address l_2, we can additionally access l_3 and l_1. When these addresses are combined, the set of accessible addresses becomes $\{l_1, l_2, l_3\}$. Since l_3 has been newly added compared to the previous set, we check whether there are any additional accessible addresses in memory via l_3. Since l_3 contains the address value l_4, we expand the set of accessible memory to $\{l_1, l_2, l_3, l_4\}$. Let's now look at the value stored at the newly added address l_4. It

contains the function value $(\mathtt{x}, E, \{\mathtt{z} \mapsto l_5\})$; if this function is called, the address l_5 can be accessed via the saved environment $\{\mathtt{z} \mapsto l_5\}$. Therefore, we also add l_5 to the set: $\{l_1, l_2, l_3, l_4, l_5\}$. We repeat the above process for l_5. In this case, since l_5 contains an integer value, there are no further accessible addresses; consequently, we can conclude that $\{l_1, l_2, l_3, l_4, l_5\}$ represents the complete set of addresses accessible (reachable) in the current environment. If we retain only these addresses in memory and reuse the remaining ones, the result is as follows.

$$\sigma' = \left\{ \begin{array}{l} l_1 \mapsto 0 \\ l_2 \mapsto \{\mathtt{a} \mapsto l_3, \mathtt{b} \mapsto l_4\} \\ l_3 \mapsto l_4 \\ l_4 \mapsto (\mathtt{x}, E, \{\mathtt{z} \mapsto l_5\}) \\ l_5 \mapsto 0 \end{array} \right\}$$

Let's precisely define the memory addresses accessible through the current environment. Suppose $\mathsf{Reach}(\rho, \sigma)$ is the set of addresses in memory σ that are accessible through environment ρ. It can be defined as the smallest set that satisfies the rules in Figure 7.6.

The first rule means that all addresses directly accessible in the current environment must be included in the set $\mathsf{Reach}(\rho, \sigma)$. The second rule states that if l is an accessible address, and the

$$\frac{}{\rho(x) \in \mathsf{Reach}(\rho, \sigma)}\ x \in \mathrm{Dom}(\rho)$$

$$\frac{l \in \mathsf{Reach}(\rho, \sigma) \qquad \sigma(l) = l'}{l' \in \mathsf{Reach}(\rho, \sigma)}$$

$$\frac{l \in \mathsf{Reach}(\rho, \sigma) \qquad \sigma(l) = \{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}}{\{l_1, \ldots, l_n\} \subseteq \mathsf{Reach}(\rho, \sigma)}$$

$$\frac{l \in \mathsf{Reach}(\rho, \sigma) \qquad \sigma(l) = (x, E, \rho')}{\mathsf{Reach}(\rho', \sigma) \subseteq \mathsf{Reach}(\rho, \sigma)}$$

Figure 7.6: Memory addresses accessible from the environment

value stored at l in memory is the same as that at another address, l', l' must also be included in the set of accessible addresses. Similarly, the third rule states that when l is a record—a collection of addresses—the addresses represented by each field must all belong to the set of accessible addresses. The final rule means that when the value stored in l is a function value, all memory addresses ($\mathsf{Reach}(\rho', \sigma)$) accessible from the environment in which the function's body is evaluated must also be collected.

Let GC be a function that performs automatic memory recycling (garbage collection). It can be defined as follows.

$$\mathsf{GC}(\rho, \sigma) = \sigma|_{\mathsf{Reach}(\rho, \sigma)}$$

Here, $\sigma|_X$ refers to memory σ from which all addresses except

those in the set X have been deleted. To use GC, simply call it before evaluating any expression E that has no continuation after evaluation.

$$\rho, \mathsf{GC}(\rho, \sigma) \vdash E \Rightarrow v, \sigma'$$

For example, in the previous sample program

```
let f = fun (x) (x+1) in
  let a = f 0 in
    a + 1
```

, since the expression `a + 1` has no remaining tasks after evaluation, GC can be called before execution. However, when evaluating the function call `f 0` or the function body `x+1`, there is still work to be done, so GC is not called.

7.4 Implementation

The language described so far can be defined in OCaml as follows.

```
type program = exp
and exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
```

```
  | DIV of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
  | CALLREF of exp * var
  | ASSIGN of var * exp
  | SEQ of exp * exp
  | EMPTYREC                        (* {} *)
  | REC of var * exp * var * exp    (* { x:=E1, y:=E2 *)
  | FIELD of exp * var              (* E.x *)
  | FIELDASSIGN of exp * var * exp (* E1.x := E2 *)
  | NEW of exp                      (* new E *)
  | ADDROF of var                   (* &x *)
  | DEREF of exp                    (* *E *)
  | STORE of exp * exp              (* *E1 := E2 *)
and var = string
```

1. Let's implement the value (`value`), environment (`env`), and memory (`mem`) for the language above.

2. Let's implement an execution function for the type below.

```
eval: program -> env -> mem -> mem
```

3. Let's implement an automatic garbage collector `gc` for the type below.

```
gc: env -> mem -> mem
```

4. Let's integrate the garbage collector `gc` into the evaluator `eval` defined above.

8

Static Type Systems

So far, we have defined the execution semantics of a programming language and implemented an executor. In this chapter, let's explore ways to predict a program's execution semantics without actually executing it. This technique is generally referred to as static program analysis or simply static analysis. Among the various branches of static analysis, let's understand the principles of static type systems—which are widely used in modern programming languages—and implement them in our language.

The goal of a static type system is to filter out all programs containing type errors before they are executed. For example, the programs below all contain type errors and are filtered out by the type system.

- `let x = iszero 0 in (x+3)`

  According to the semantic rules for addition $E_1 + E_2$, the values of E_1 and E_2 must both be of type integer. In this example, since the value of `x` is a Boolean value *true*, the meaning of `x+3` is undefined.

- `if 3 then 88 else 99`

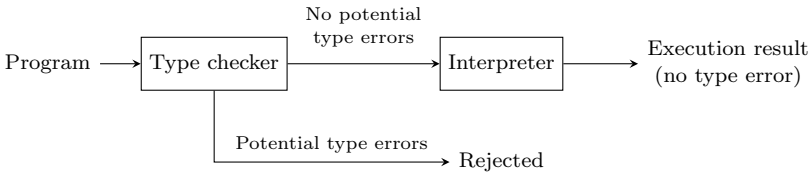


Figure 8.1: A programming language system equipped with a type system

According to the semantic structure, in the conditional expression `if` E_1 `then` E_2 `else` E_3, the value of E_1 must be either *true* or *false*. In this example, since an integer value is being evaluated, the meaning is not defined at runtime, and a type error occurs.

- `(fun x (3 x)) 1`

  The function call expression `(3 x)` treats an integer value as a function and calls it. Again, no applicable rule can be found in the semantic structure, and a type error occurs at runtime.

In real-world programming languages, type errors can occur in complex and varied situations. It is difficult to write a program while anticipating all such errors. A static type system makes it easy to develop safe programs by automatically detecting all type errors at the compilation stage.

Figure 8.1 illustrates the process by which a program is executed in a programming language equipped with a static type system. First, the type checker analyzes the input program to determine whether there is a possibility of a type error occurring during program execution. If a potential type error is detected, the type checker rejects the program, and the input program is not executed. Therefore, programs that pass the type checker and are executed by the interpreter are those with no potential for type errors. It is guaranteed that programs that pass the type checker will never encounter a type error during execution.

8.1 Target Language

Let us consider the language shown in Figure 8.2 as the target for designing a type system. This language possesses only the key features of a functional language. Its semantic space is as follows.

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{RecProcedure} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \\ \mathit{RecProcedure} &= \mathit{Var} \times \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \end{aligned}$$

The execution rules are as shown in Figure 8.3.

$$
\begin{array}{rcl}
E & \to & n \\
& | & x \\
& | & E_1 \texttt{ + } E_2 \\
& | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{iszero } E \\
& | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\
& | & \texttt{fun } x\ E \\
& | & \texttt{letrec } f(x) = E \texttt{ in } E \\
& | & E_1\ E_2
\end{array}
$$

Figure 8.2: Grammatical structure of the target language

8.2 Type

First, let's define what a "type" is. In a programming language, a type refers to a set of specific values. For example, the integer type (int) refers to the entire set of integers ($\mathbb{Z}$), and the Boolean (bool) type refers to the set $\{\mathit{true}, \mathit{false}\}$. Thus, a type can be described as an abstraction of the values used in a program, grouped by their kind. It is similar to summarizing integer values such as $1, 2, 3, \ldots$—rather than distinguishing them individually—and referring to them collectively as int.

Generally, a type is an abstraction of values, and types can be defined at various levels of abstraction;[1)] In most programming languages, types are typically defined at the following lev-

1) For example, instead of simply referring to all integers as int, the type of natural numbers is $\text{int}_{\geq 0}$, and the type of negative integers is $\text{int}_{<0}$, allowing for a more granular distinction.

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 \texttt{ + } E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{true}} \qquad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{iszero } E \Rightarrow \mathit{false}}\ n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{true} \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathit{false} \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \{x \mapsto v_1\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

$$\frac{}{\rho \vdash \texttt{fun } x\ E \Rightarrow (x, E, \rho)}$$

$$\frac{\{f \mapsto (f, x, E_1, \rho)\}\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad \{x \mapsto v\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

$$\frac{\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \qquad \{x \mapsto v, f \mapsto (f, x, E, \rho')\}\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

Figure 8.3: Semantic structure of the target language

els.[2)]

$$\frac{}{\text{int}} \qquad \frac{}{\text{bool}} \qquad \frac{T_1 \quad T_2}{T_1 \to T_2}$$

2) If you're curious about a programming language that uses types in a very general way, take a look at Coq (https://coq.inria.fr). Coq is a functional language similar to OCaml, but it supports a very rich set

The set of all types was defined inductively. int and bool denote the integer and Boolean types, respectively. Functions were defined by the types of their arguments and return values. When T_1 and T_2 are arbitrary types, $T_1 \to T_2$ denotes the function type from T_1 to T_2. Just as all integers can be summarized as 'int', all functions could also be summarized as 'function', but doing so would severely limit the expressiveness of the type system, ultimately resulting in a type system that is not useful.

Let's look at a few examples of function types.

- int → int: This denotes the set of functions that take an integer as input and return an integer. For example,

  `fun n (n+1)`, `fun n (let x = 1 in x + n)`

  are functions that belong to this type.

- bool → int: Refers to functions that take a true/false value as input and return an integer. For example, a function such as `fun b (if b then 0 else 1)` belongs to this type.

- int → (int → bool): Refers to functions that take an integer as input and return a function of type int → bool. For

of types, allowing any property of a program to be expressed as a type. Coq's type checker automatically verifies whether a program possesses these properties; from this perspective, Coq is referred to as a proof assistant.

example,

```
fun n (fun m (iszero (m + n)))
```

Functions such as this belong to this type.

- (int → int) → (bool → bool): Refers to functions that take a function of type int → int as input and return a function of type bool → bool. For example, the following function belongs to this type.

  ```
  fun f (fun b (if b then b else (iszero (f 1))))
  ```

  As a simpler example, `fun f (fun b (b))` also belongs to this type.

- (int → int) → (bool → (bool → int)): This is a function type that takes a function of type int → int as an argument and returns a function of type (bool → (bool → int)). For example,

  ```
  fun f (fun b1 (fun b2 (f 1)))
  ```

  belongs to this type.

8.3 Type Environment

Just as with the definition of a semantic structure, the type of the expression E can only be determined once the types of the free variables belonging to E are known. For example, the type of the expression `x+1` is int if the type of the variable `x` is int, but it is undefined in other cases. The data structure that represents the types of currently declared variables is called the type environment. Similar to the execution environment, the type environment can be defined as a function that maps variables to types.

$$\Gamma \in \mathit{TyEnv} = \mathit{Var} \rightarrow \mathit{Type}$$

Here, *TyEnv* denotes the universal set of type environments, and Γ represents a single type environment. Similar to the notation for execution environments, let's define the notation for type environments as follows.

- $\emptyset$: Denotes the empty type environment.
- $\{x \mapsto t\}\Gamma$: Denotes a new type environment that extends the type environment Γ so that the variable x has type t.
- $\Gamma(x)$: Refers to the type that the variable x has in the type environment Γ.

Note that just as a type denotes a set of values, a type environment denotes a set of execution environments. For example, the type environment

$$\{x \mapsto \text{int}, y \mapsto \text{bool}\}$$

is a finite abstraction of an infinite number of execution environments, as shown below.

$$\begin{gathered}
\vdots \\
\{x \mapsto 0, y \mapsto \mathit{true}\}, \\
\{x \mapsto 1, y \mapsto \mathit{true}\}, \\
\{x \mapsto 2, y \mapsto \mathit{true}\}, \\
\vdots \\
\{x \mapsto 0, y \mapsto \mathit{false}\}, \\
\{x \mapsto 1, y \mapsto \mathit{false}\}, \\
\{x \mapsto 2, y \mapsto \mathit{false}\}, \\
\vdots
\end{gathered}$$

From this perspective, a static type system can be intuitively understood as an abstract interpreter that executes a program using a summarized value called a type. For example, the actual execution of the expression $1 + 2$ computes 3, but the abstract execution summarizes the integer 1 and 2 into the type int, and

then performs addition in the world of types to compute int as the result. Although it is impossible (undecidable) to precisely compute the actual meaning of a program's execution before it runs, it may be possible (decidable) to compute it after summarizing it into types.

8.4 Type Inference Rules

Let us express "In type environment Γ, expression E has type t" as follows.

$$\Gamma \vdash E : t$$

Let's devise rules for inferring these facts. We can define an inference rule inductively, one for each case of E.

- In the case of $E = n$: The type of the integer n is always int, regardless of the type environment. This can be expressed as an inference rule as follows.

  $$\frac{}{\Gamma \vdash n : \text{int}}$$

  This means that, given a type environment Γ, the type of any integer n can always be concluded to be int.

- In the case of x: The type of the variable x is determined by the given type environment. Expressed as an inference

rule, it is as follows.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

Type inference is possible only if $\Gamma(x)$ is defined—that is, only if the type environment Γ contains information about the variable x.

- In the case of E_1 `+` E_2: The result of the addition expression E_1 `+` E_2 is of type int. However, it does not always have a type; it has a type only when both expressions E_1 and E_2 are of type int. This can be expressed by the following inference rule.

$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 \texttt{ + } E_2 : \text{int}}$$

- In the case of `iszero` E: Since the result of evaluating `iszero` E is always a true or false value, it has the type bool. However, the only case where it has a type is when E computes an integer value. This can be expressed as the following inference rule.

$$\frac{\Gamma \vdash E : \text{int}}{\Gamma \vdash \texttt{iszero } E : \text{bool}}$$

- In the case of `if` E_1 `then` E_2 `else` E_3: First, let's assume that the type of E_1 is bool. Furthermore, assuming that the types of E_2 and E_3 are identical, let's call that type t. In that case, the type of the entire conditional expression can be denoted as t. Expressed as an inference rule, this is as follows.

$$\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : t \quad \Gamma \vdash E_3 : t}{\Gamma \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 : t}$$

  For this rule, we considered only cases where the types of E_2 and E_3 are the same. Therefore, the type cannot be inferred for a program that generates values of different types in each case of the conditional expression, as shown below.

```
if true then 1 else false
```

  OCaml's type system also has the same constraint, because including cases where the types of expressions E_2 and E_3 differ would make the cost of static type checking too high.

- In the case of `let` x `=` E_1 `in` E_2: Let's assume that in the type environment Γ, the type of E_1 can be inferred as t_1. Furthermore, suppose that in the type environment $\{x \mapsto t_1\}\Gamma$, the type of E_2 can be inferred as t_2. In that case, we can conclude that the type of the entire expres-

sion `let` x `=` E_1 `in` E_2 is t_2. This process can be expressed as an inference rule as follows:

$$\frac{\Gamma \vdash E_1 : t_1 \quad \{x \mapsto t_1\}\Gamma \vdash E_2 : t_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2 : t_2}$$

- In the case of $E_1\ E_2$: For this to be meaningful, the expression E_1 must first be an expression that produces a function value. Let the type of that function value be $t_1 \to t_2$. And let's assume that the type of E_2 is identical to the argument type t_1. In that case, the function call expression $E_1\ E_2$ can be executed without any issues, and the type of its result will be t_2, which is the function's return type. This can be expressed using the following inference rule.

$$\frac{\Gamma \vdash E_1 : t_1 \to t_2 \quad \Gamma \vdash E_2 : t_1}{\Gamma \vdash E_1\ E_2 : t_2}$$

- In the case of `fun` x E: Suppose we assume that the type of the argument x is t_1, and that we can then infer the type of E to be t_2. In that case, we can conclude that the type of the given function expression is $t_1 \to t_2$. Expressed as an inference rule, it is as follows.

$$\frac{\{x \mapsto t_1\}\Gamma \vdash E : t_2}{\Gamma \vdash \texttt{fun}\ x\ E : t_1 \to t_2}$$

When devising inference rules, there is no need to worry about how to find the type t_1 of x. The inference rule above merely illustrates what properties can be logically inferred under the assumption that x has type t_1.

- In the case of `letrec` $f(x) = E_1$ `in` E_2: Let the argument and return types of the recursive function be t_2 and t_1, respectively and the type of the argument x is assumed to be t_2, let's assume that the type of the function body E_1 can be inferred as t_1. Furthermore, for such t_1 and t_2 (after assuming that f is of type $t_2 \rightarrow t_1$), if we can infer the type of E_2 to be t, then we can conclude that the type of the entire expression is t. Expressed as an inference rule, it is as follows.

$$\frac{\{x \mapsto t_2, f \mapsto (t_2 \rightarrow t_1)\}\Gamma \vdash E_1 : t_1 \qquad \{f \mapsto (t_2 \rightarrow t_1)\}\Gamma \vdash E_2 : t}{\Gamma \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 : t}$$

  In this rule as well, the specific nature of t_1, t_2 is irrelevant. Assuming the type of x is t_2, and the type of f as $t_1 \rightarrow t_1$, this rule expresses the logical inference relationship regarding what the type of the entire expression should be.

A summary of the type inference rules defined so far is shown

in Figure 8.4.

The type checker shown in Figure 8.1 analyzes the given program E to determine what type t it can have based on the inference rules. It checks whether the inference rules can prove the following facts for a given type t and the empty type environment ($\emptyset$).

$$\emptyset \vdash E : t$$

If it succeeds in inferring this fact, the type checker concludes that the type of the program E is t and, therefore, there is no type error. If it fails to infer this fact, the type checker concludes that E cannot have a type and, therefore, may contain a type error, and it will refuse to execute the program. From this perspective, the type checker can be regarded as an automatic theorem prover that, when given the program E,

$$\frac{\vdots}{\emptyset \vdash E : t}$$

.

Let's look at a few examples. The expression `iszero (1 + 2)` produces a value of type bool and does not result in a type error during execution, because $\emptyset \vdash$ `iszero (1 + 2)` : bool can be

$$\frac{}{\Gamma \vdash n : \text{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 \texttt{ + } E_2 : \text{int}}$$

$$\frac{\Gamma \vdash E : \text{int}}{\Gamma \vdash \texttt{iszero } E : \text{bool}}$$

$$\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : t \quad \Gamma \vdash E_3 : t}{\Gamma \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 : t}$$

$$\frac{\Gamma \vdash E_1 : t_1 \quad \{x \mapsto t_1\}\Gamma \vdash E_2 : t_2}{\Gamma \vdash \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 : t_2}$$

$$\frac{\Gamma \vdash E_1 : t_1 \to t_2 \quad \Gamma \vdash E_2 : t_1}{\Gamma \vdash E_1 \ E_2 : t_2}$$

$$\frac{\{x \mapsto t_1\}\Gamma \vdash E : t_2}{\Gamma \vdash \texttt{fun } x \ E : t_1 \to t_2}$$

$$\frac{\begin{array}{c}\{x \mapsto t_2, f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_1 : t_1 \\ \{f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_2 : t\end{array}}{\Gamma \vdash \texttt{letrec } f(x) = E_1 \texttt{ in } E_2 : t}$$

Figure 8.4: Type Inference Rules

proven using the type inference rules as follows.

$$\frac{\dfrac{\emptyset \vdash \texttt{1} : \text{int} \quad \emptyset \vdash \texttt{2} : \text{int}}{\emptyset \vdash \texttt{1 + 2} : \text{int}}}{\emptyset \vdash \texttt{iszero (1 + 2)} : \text{bool}}$$

The type of the expression `(fun x (x)) 1` can also be in-

$$\dfrac{\dfrac{\dfrac{\{\texttt{x} \mapsto \text{int}, \texttt{y} \mapsto \text{bool}\} \vdash \texttt{y} : \text{bool} \quad \{\texttt{x} \mapsto \text{int}, \texttt{y} \mapsto \text{bool}\} \vdash \texttt{x} : \text{int} \quad \{\texttt{x} \mapsto \text{int}, \texttt{y} \mapsto \text{bool}\} \vdash \texttt{1} : \text{int}}{\{\texttt{x} \mapsto \text{int}, \texttt{y} \mapsto \text{bool}\} \vdash \texttt{if y then x else 1} : \text{int}}}{\{\texttt{x} \mapsto \text{int}\} \vdash \texttt{fun y (if y then x else 1)} : \text{bool} \to \text{int}}}{\emptyset \vdash \texttt{fun x (fun y (if y then x else 1))} : \text{int} \to (\text{bool} \to \text{int})}$$

Figure 8.5: An example of the type inference process

ferred as follows.

$$\dfrac{\dfrac{\{\texttt{x} \mapsto \text{int}\} \vdash \texttt{x} : \text{int}}{\emptyset \vdash \texttt{fun x (x)} : \text{int} \to \text{int}} \quad \emptyset \vdash \texttt{1} : \text{int}}{\emptyset \vdash \texttt{(fun x (x)) 1} : \text{int}}$$

The type checker automatically generates such proof trees and verifies that there are no type errors.

The expression `fun x (fun y (if y then x else 1))` is also a well-typed program, and as shown in Figure 8.5, its type can be proven to be int → (bool → int).

On the other hand, a program with type errors does not have a type according to the inference rules above. For example, the type of the expression `(fun x (3 x)) 1` cannot be inferred using the rules above.

Let's examine the features of the type inference rules we have designed so far.

Type Inference Is Not Unique First, given the program E, the type determined by the above rules may not be unique. For example,

- `fun x (x)` can be assigned an infinite number of types. For any type t, any type of the form $t \to t$ can be considered the type of the function `fun x x`. Here are a few examples:

$$\frac{\{\texttt{x} \mapsto \text{int}\} \vdash \texttt{x} : \text{int}}{\emptyset \vdash \texttt{fun x x} : \text{int} \to \text{int}}$$

$$\frac{\{\texttt{x} \mapsto \text{bool}\} \vdash \texttt{x} : \text{bool}}{\emptyset \vdash \texttt{fun x x} : \text{bool} \to \text{bool}}$$

$$\frac{\{\texttt{x} \mapsto (\text{int} \to \text{int})\} \vdash \texttt{x} : \text{int} \to \text{int}}{\emptyset \vdash \texttt{fun x x} : (\text{int} \to \text{int}) \to (\text{int} \to \text{int})}$$

$$\vdots$$

- `fun f (f 3)`: For any type t, it has type $(\text{int} \to t) \to t$. If $t = \text{bool}$, the type is inferred as follows.

$$\frac{\dfrac{\{\texttt{f} \mapsto \text{int} \to \text{bool}\} \vdash \texttt{f} : \text{int} \to \text{bool} \quad \{\texttt{f} \mapsto \text{int} \to \text{bool}\} \vdash \texttt{3} : \text{int}}{\{\texttt{f} \mapsto \text{int} \to \text{bool}\} \vdash \texttt{f 3} : \text{bool}}}{\emptyset \vdash \texttt{fun f (f 3)} : (\text{int} \to \text{bool}) \to \text{bool}}$$

- `fun f (fun x (f (f x)))`: For any type t, it has type $((t \to$

$t) \rightarrow (t \rightarrow t))$.

Type inference is sound The type inference rules we have designed are sound. A program with a type error will never have its type successfully inferred. In other words, if type inference succeeds according to the inference rules, no type errors will occur during program execution. If E—an expression with no free variables—and an empty type environment can be used to infer $\emptyset \vdash E : t$ using the above inference rules, then E will not cause a type error during execution. Furthermore, if the evaluation of the expression E is completed and the value v is computed, it is guaranteed that the type of that value is t. For example,

- `(fun (x) x) 1` can have its type (int) inferred using the above rule, so it executes correctly without a type error, and the result is an integer.
- `(fun (x) (x 3)) 4` is a program that causes a type error. Programs with type errors like this cannot have their types inferred using the rules above.

The soundness of this type system can be rigorously proven. The soundness of the type system is a property that holds between the execution rules of the programming language (Figure 8.3) and the type inference rules (Figure 8.4). Since both sets of

rules are mathematically defined, the relationship and properties between them can also be rigorously defined and proven.[3)]

Type Inference Is Incomplete The type system we designed is safe but incomplete. This means that while a program may run correctly without type errors, it may still fail to have its type inferred. For example, the two programs below execute without any problems when run according to their semantic structure, but according to our type system, they have no type.

- `if iszero 1 then 2 else (iszero 3)`
- `(fun f (f f)) (fun x x)`

It is impossible to design a safe and complete static type system for a general-purpose programming language that is Turing-complete. Therefore, a static type system must always sacrifice either safety or completeness (or both); in the case of our type system, we chose to sacrifice completeness while preserving safety.

3) In fact, the style of execution rules we have defined is not well-suited for proving type systems. If you are curious about how to prove the safety of a type system, refer to the following book: Benjamin C. Pierce, *Types and Programming Languages*, MIT Press.

8.5 Type Checker Implementation Approach

The type checker is an algorithm that takes the program E as input and verifies whether it is free of type errors. It works by attempting to infer the type of E in an empty type environment using type inference rules; if the inference is successful, it determines that there are no type errors, and if the type cannot be inferred, it determines that there is a type error and refuses to execute the program. As explained earlier, the type checker is an automated prover that uses type inference rules to find a proof for $\emptyset \vdash E : t$ given a type t. If it succeeds in finding a type t for which a proof tree exists, it accepts the input program; otherwise, it rejects it.

- The type checker accepts the program E only if it has successfully proven $\emptyset \vdash E : t$ for some t.
- If the proof is impossible, E is not accepted.

How can we implement such a type checker? Just as we did when implementing the executor, let's try implementing the type inference rules as recursive functions. We can implement the type inference rules using a function like the one shown in Figure 8.6.

While it is possible to implement inference rules using recursive functions for most cases, there is one problem. It is impos-

```
let rec typecheck Γ E =
  match E with
  | n → int
  | x → Γ(x)
  | E1 + E2 →
    let t1 = typecheck Γ E1
    let t2 = typecheck Γ E2
      if t1 = int and t2 = int then int
      else raise TypeError
  | ···
```

Figure 8.6: Implementation of a type checker using recursive functions

sible to implement the following rule as a recursive function.

$$\frac{\{x \mapsto t_1\}\Gamma \vdash E : t_2}{\Gamma \vdash \texttt{fun}\ x\ E : t_1 \to t_2}$$

This rule can find the appropriate type t_1 for the function's argument x, and and if the type of E is t_2, it can conclude that the function's type is $t_1 \to t_2$. Although this is a logically sound inference rule, implementing it requires being able to find the appropriate type t_1. However, when implementing this rule as a recursive function, there is no straightforward way to determine what the type t_1 should be. This is because t_1 is not obtained by recursively inferring the type of a sub-expression of the function expression `fun` x E, but rather is a type that must be guessed in advance when applying the inference rule above.

Since it is difficult to predict the type t_1 of the argument x in advance, the inference rule above cannot be directly converted into an algorithm in the form of a recursive function. Similarly, in the inference rule below, it is difficult to predict the types of the arguments and the function (t_1, t_2) in advance.

$$\frac{\{x \mapsto t_2, f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_1 : t_1 \quad \{f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_2 : t}{\Gamma \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 : t}$$

Depending on the approach taken to solve this problem, the type systems implemented in programming languages can be classified into two categories.

- The approach that leaves type inference to the programmer: This is the approach used in languages such as C, C++, and Java. In these languages, for example, it is impossible to omit the function type; it must always be specified.

- Automatic type inference via algorithms: This is the approach used in languages such as OCaml and Haskell. In these languages, even if function types are omitted, the type system automatically infers the appropriate types.

The approach that leaves type inference to the programmer is straightforward. First, the language's syntax is modified as

follows to allow types to be written in the program.

$$
\begin{array}{rcl}
E & \rightarrow & \vdots \\
 & | & \texttt{fun}\ (x : t)\ E \\
 & | & \texttt{letrec}\ t_1\ f(x : t_2) = E\ \texttt{in}\ E
\end{array}
$$

All other parts remain the same; only the syntax for defining functions has been changed. When defining a regular function, the argument type t must be specified, and for recursive functions, both the argument type t_2 and the return type t_1 must be specified. For example, when defining a function as follows, specifying the types is now mandatory rather than optional.

```
fun (x: int) (x+1)
```

```
letrec int double (x: int) =
  if iszero x then 0 else (double (x-1)) + 2
 in double 2
```

```
fun (f: (bool -> int))
  fun (n : int)
    (f (iszero n))
```

The following two cases represent changes to the type infer-

```
let rec typecheck Γ E =
  match E with
  | fun (x : t1) E1 →
    let t2 = typecheck ({x ↦ t1}Γ) E1
    in t1 → t2
  | letrec t1 f(x : t2) = E1 in E2
    let t' = typecheck ({x ↦ t2, f ↦ (t2 → t1)}Γ) E1
    in if t' = t1 then typecheck ({f ↦ (t2 → t1)}Γ) E2
       else raise TypeError
  | ···
```

Figure 8.7: A type checker that leaves type inference to the programmer and implements it as a recursive function

ence rules.

$$\frac{\{x \mapsto t_1\}\Gamma \vdash E : t_2}{\Gamma \vdash \texttt{fun}\ (x : t_1)\ E : t_1 \to t_2}$$

$$\frac{\{x \mapsto t_2, f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_1 : t_1 \quad \{f \mapsto (t_2 \to t_1)\}\Gamma \vdash E_2 : t}{\Gamma \vdash \texttt{letrec}\ t_1\ f(x : t_2) = E_1\ \texttt{in}\ E_2 : t}$$

Since the types of function arguments are specified in the program, the type inference rules now use them directly. When defining recursive functions, the specified argument and return types are also used as-is.

Type inference rules can now be easily implemented as recursive functions. If we consider only the two cases mentioned above, the result is shown in Figure 8.7.

8.6 Automatic Type Inference Algorithm

Let's design a type-checker algorithm that automatically infers types without human intervention. This algorithm, used in programming languages such as OCaml, analyzes a given program to infer the types of function arguments and results. It operates in two steps: generating type equations from the given program and solving them.

8.6.1 Generating Type Equations

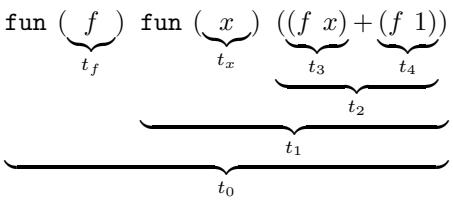
In the first step, type variables are introduced for each sub-expression and variable in the program, and the conditions they must satisfy are expressed as equations.

Example 1 For example, consider the following program.[4)]

$$\texttt{fun}\ (f)\ \texttt{fun}\ (x)\ ((f\ x) + (f\ 1))$$

4) Examples 1–4 are adapted from the examples in Chapter 7 of the following book: Daniel P. Friedman and Mitchell Wand, *Essentials of Programming Languages*, MIT Press.

Let us introduce type variables for each sub-expression and variable in the program as follows.



For example, t_f is a type variable denoting the type of the variable f, t_0 is a type variable denoting the type of the entire program, t_3 is a type variable denoting the type of the sub-expression $(f\ x)$. After introducing type variables for each sub-expression and variable in this manner, we determine the conditions they must satisfy. The conditions that type variables must satisfy are derived from the type inference rules. For example, since the entire program is a function definition with argument type t_f and return type t_1, the following relationship must hold between the type variables t_0, t_1, t_f.

$$t_0 = t_f \to t_1$$

This means that the type denoted by t_0 must be equal to the function type composed of t_f and t_1. Similarly, the following re-

lationship must hold between t_1, t_x, t_2 and

$$t_1 = t_x \to t_2$$

Furthermore, for the expression $((f\ x) + (f\ 1))$ to have a type, that type must be int, and the sub-expressions $(f\ 3)$ and $(f\ x)$ must both be of type int. This can be expressed as an equation as follows:

$$t_2 = \text{int}, \quad t_3 = \text{int}, \quad t_4 = \text{int}$$

Furthermore, looking at the function call expression $(f\ 1)$, we see that f must have the function type, its argument type must be int, and the return type must be t_4.

$$t_f = \text{int} \to t_4$$

Finally, according to the function call expression $(f\ x)$, the following relationship must hold between t_f, t_x, t_3.

$$t_f = t_x \to t_3$$

Combining all the constraints generated so far yields a system of equations as shown on the left side of Figure 8.8. Since the number of unknowns (type variables) equals the number of equations, if a solution exists, it is a case where the values of all

Type Equation			Solution to the Type Equation		
t_0	=	$t_f \to t_1$	t_0	=	$(\text{int} \to \text{int}) \to (\text{int} \to \text{int})$
t_1	=	$t_x \to t_2$	t_1	=	$\text{int} \to \text{int}$
t_2	=	int	t_2	=	int
t_3	=	int	t_3	=	int
t_4	=	int	t_4	=	int
t_f	=	$\text{int} \to t_4$	t_f	=	$\text{int} \to \text{int}$
t_f	=	$t_x \to t_3$	t_x	=	int

Figure 8.8: Type Equations and Solutions

unknowns can be determined. The solution to the equations is shown on the right side of Figure 8.8. If we substitute the solution into the unknowns of the above equations, the left-hand and right-hand sides of all equations will match.

Example 2 For the program below,

$$\underbrace{\texttt{fun}\ (\underbrace{f}_{t_f})\ \underbrace{(f\ 0)}_{t_1}}_{t_0}$$

the type equations can be set up as follows.

$$\begin{aligned} t_0 &= t_f \to t_1 \\ t_f &= \text{int} \to t_1 \end{aligned}$$

The solutions are as follows.

$$\begin{aligned} t_0 &= (\text{int} \to t_1) \to t_1 \\ t_f &= \text{int} \to t_1 \end{aligned}$$

In this case, since there are more unknowns than equations, we cannot determine the solution for all unknowns.

Example 3 Consider the following program.

$$\underbrace{\texttt{if } \underbrace{x}_{t_x} \texttt{ then } \underbrace{(x+1)}_{t_1} \texttt{ else } 0}_{t_0}$$

According to the type inference rules, in the conditional expression `if` E_1 `then` E_2 `else` E_3, E_1 must be of type bool, and the types of E_2 and E_3 must be the same, and that type becomes the type of the entire conditional expression. Expressed as an equation, it is as follows.

$$\begin{aligned} t_x &= \text{bool} \\ t_1 &= t_0 \\ \text{int} &= t_0 \\ t_x &= \text{int} \\ t_1 &= \text{int} \end{aligned}$$

The last two equations were generated from $x + 1$.

The above equations have no solution. This is because the first and fourth equations contain mutually contradictory conditions.

Example 4 For the program below,

$$\underbrace{\texttt{fun}\ (\underbrace{f}_{t_f})\ \underbrace{(\texttt{iszero}\ \underbrace{(f\ f)}_{t_2})}_{t_1}}_{t_0}$$

The equations can be set up as follows.

$$\begin{aligned} t_0 &= t_f \to t_1 \\ t_1 &= \text{bool} \\ t_2 &= \text{int} \\ t_f &= t_f \to t_2 \end{aligned}$$

The last equation was generated from the sub-expression $(f\ f)$. Assuming the type of f is t_f, this is because f is used as an argument for f, and its result is used as an argument for `iszero`.

The equation above also has no solution. Looking at the last equation, the same type variable t_f is used in the same form as $t_f = t_f \to \ldots$. No type can exist that satisfies these constraints.

Algorithm for Generating Type Equations

Let's describe the process of generating type equations precisely. Since type variables are used in the equations, let's add type variables to the types defined earlier.

$$\begin{array}{rcl} T & \rightarrow & \text{int} \\ & | & \text{bool} \\ & | & T \rightarrow T \\ & | & \alpha \ (\in \mathit{TyVar}) \end{array}$$

Here, α denotes a type variable, and TyVar denotes the set of type variables.

Type equations can be defined inductively as follows.

$$\begin{array}{rcl} \mathit{TyEqn} & \rightarrow & \emptyset \\ & | & T_1 \doteq T_2 \ \wedge \ \mathit{TyEqn} \end{array}$$

Here, $T_1 \doteq T_2$ is an equation meaning that the two types T_1 and T_2 (The symbol $\doteq$ is used to denote the equality of these two types to avoid confusion with the general $=$. When there is no risk of confusion, $=$ may be used instead of $\doteq$). A type equation is a collection of these individual equations, linked together by $\wedge$ (and).

The type of the algorithm that derives equations from a pro-

gram is as follows.

$$\mathcal{V} : (\mathit{Var} \to T) \times E \times T \to \mathit{TyEqn}$$

The algorithm $\mathcal{V}$ takes three arguments. The first is the type environment $\Gamma \in \mathit{Var} \to T$, the second is the program from which the equation is to be derived $e \in E$, and the last is the type $t \in T$ that the given program must have. $\mathcal{V}(\Gamma, e, t)$ generates the conditions that must hold for the expression e to have type t in environment Γ. For example,

- $\mathcal{V}(\{x \mapsto \text{int}\}, \texttt{x+1}, \alpha)$ generates the equation $\alpha \doteq \text{int}$. This is because, in the environment $\{x \mapsto \text{int}\}$, for the program `x+1` to have type α, condition $\alpha \doteq \text{int}$ must hold.

- $\mathcal{V}(\emptyset, \texttt{fun}\ (x)\ (\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ 2), \alpha \to \beta)$ generates the equation

  $$\alpha \doteq \text{bool} \ \wedge \ \beta \doteq \text{int}$$

  . This is because, for the type of a given program to be denoted as $\alpha \to \beta$, both $\alpha \doteq \text{bool}$ and $\beta \doteq \text{int}$ must be satisfied.

If we recursively define algorithm $\mathcal{V}$ to generate equations like this, the result is shown in Figure 8.9.

$$
\begin{aligned}
\mathcal{V}(\Gamma, n, t) &= t \doteq \text{int} \\
\mathcal{V}(\Gamma, x, t) &= t \doteq \Gamma(x) \\
\mathcal{V}(\Gamma, e_1 + e_2, t) &= t \doteq \text{int} \ \wedge \ \mathcal{V}(\Gamma, e_1, \text{int}) \ \wedge \ \mathcal{V}(\Gamma, e_2, \text{int}) \\
\mathcal{V}(\Gamma, \texttt{iszero}\ e, t) &= t \doteq \text{bool} \ \wedge \ \mathcal{V}(\Gamma, e, \text{int}) \\
\mathcal{V}(\Gamma, \texttt{if}\ e_1\ e_2\ e_3, t) &= \mathcal{V}(\Gamma, e_1, \text{bool}) \ \wedge \ \mathcal{V}(\Gamma, e_2, t) \ \wedge \ \mathcal{V}(\Gamma, e_3, t) \\
\mathcal{V}(\Gamma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2, t) &= \mathcal{V}(\Gamma, e_1, \alpha) \ \wedge \ \mathcal{V}([x \mapsto \alpha]\Gamma, e_2, t) \ (\text{new } \alpha) \\
\mathcal{V}(\Gamma, \texttt{fun}\ (x)\ e, t) &= t \doteq \alpha_1 \to \alpha_2 \ \wedge \ \mathcal{V}([x \mapsto \alpha_1]\Gamma, e, \alpha_2) \\
&\quad (\text{new } \alpha_1, \alpha_2) \\
\mathcal{V}(\Gamma, e_1\ e_2, t) &= \mathcal{V}(\Gamma, e_1, \alpha \to t) \ \wedge \ \mathcal{V}(\Gamma, e_2, \alpha) \ (\text{new } \alpha)
\end{aligned}
$$

Figure 8.9: Algorithm for Generating Type Equations

- $\mathcal{V}(\Gamma, n, t)$: For an integer n in the type environment Γ to have type t, t must be int ($t \doteq \text{int}$).

- $\mathcal{V}(\Gamma, x, t)$: In the type environment Γ, for the variable x to have type t, t must have the type environment Γ must be the same as the type $\Gamma(x)$ that x has ($t \doteq \Gamma(x)$).

- $\mathcal{V}(\Gamma, e_1 + e_2, t)$: In the type environment Γ, if the expression $e_1 + e_2$ has type t, t must be int ($t \doteq \text{int}$). Next, type equations are generated recursively for the expressions e_1 and e_2 ($\mathcal{V}(\Gamma, e_1, \text{int})$, $\mathcal{V}(\Gamma, e_2, \text{int})$). Since both e_1 and e_2 must be integer types, int was provided as the last argument to $\mathcal{V}$.

- $\mathcal{V}(\Gamma, \texttt{iszero}\ e, t)$: In the type environment Γ, for the ex-

pression `iszero` e to have type t, t must be bool. Then, generate the equation recursively for e ($\mathcal{V}(\Gamma, e, \text{int})$).

- $\mathcal{V}(\Gamma, \texttt{if}\ e_1\ e_2\ e_3, t)$: For conditional expressions, the type of e_1 must be bool, and the types of e_2 and e_3 must be the same. Type equations are generated recursively to incorporate these constraints.

- $\mathcal{V}(\Gamma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2, t)$: First, the conditions that must hold for the expression e_1 are generated recursively as ($\mathcal{V}(\Gamma, e_1, \alpha)$). Since there are no specific conditions that expression e_1 must satisfy, a new type variable α is generated and passed as the last argument. The recursive call $\mathcal{V}(\Gamma, e_1, \alpha)$ will parse the expression e_1 to generate the conditions that α must satisfy. Next, it extended the current type environment, assuming that the variable x is of type α, and then generated an equation for the expression e_2. For the expression e_2, since its type must be the same as that of the entire `let` expression, t was provided as the third argument.

- $\mathcal{V}(\Gamma, \texttt{fun}\ (x)\ e, t)$: First, the function expression `fun` (x) e must have a function type. Since there are no conditions yet that the function's argument and return types must satisfy, a new type variable α_1, α_2 was created and named $t \doteq \alpha_1 \to \alpha_2$. Next, assuming the type of variable x in

the current environment to be the argument type α_1, an equation is generated recursively for the function body e_2. Since the type of the body expression must match the function's return type, α_2 was provided as the third argument.

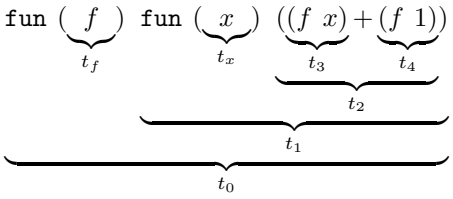
- $\mathcal{V}(\Gamma, e_1\ e_2, t)$: Since the expression e_1 must be of function type, we introduce the type variable α and set the type of e_1 to $\alpha \to t$ and generate the equation recursively. Next, we recursively generate an equation for the expression e_2 provided as an argument; its type must be identical to the function's argument type α.

When calling $\mathcal{V}$ for a program with no free variables, simply pass an empty environment and a new type variable as the first and third arguments. For example, the process of generating a type equation for (`fun` (x) (x)) 1 is as follows (α is a type variable representing the type of the entire expression).

$$
\begin{aligned}
&\mathcal{V}(\emptyset, (\texttt{fun}\ (x)\ (x))\ 1, \alpha) \\
&= \mathcal{V}(\emptyset, \texttt{fun}\ (x)\ (x), \alpha_1 \to \alpha) \wedge \mathcal{V}(\emptyset, 1, \alpha_1) && \text{(new } \alpha_1\text{)} \\
&= \alpha_1 \to \alpha \doteq \alpha_2 \to \alpha_3 \wedge \mathcal{V}([x \mapsto \alpha_2], x, \alpha_3) \wedge \alpha_1 \doteq \text{int} && \text{(new } \alpha_2, \alpha_3\text{)} \\
&= \alpha_1 \to \alpha \doteq \alpha_2 \to \alpha_3 \wedge \alpha_2 \doteq \alpha_3 \wedge \alpha_1 \doteq \text{int}
\end{aligned}
$$

8.6.2 Solving Type Equations

Now let's examine the process of finding the solution to the type equation established earlier. We'll explain this using the program below as an example.



The type equation and solution for the program above are shown in Figure 8.8.

The solution to a type equation is also called a substitution because it can be viewed as a function that replaces the type variable on the left-hand side of the solution with the type on the right-hand side. Applying such a function to each type variable in the type equation makes the left-hand and right-hand sides of each equation identical. For example, in the first equation $t_0 = t_f \to t_1$ in Figure 8.8, substituting the solution of the equation for each type variable yields the following.

$$(\text{int} \to \text{int}) \to (\text{int} \to \text{int}) = (\text{int} \to \text{int}) \to (\text{int} \to \text{int})$$

The solution to a type equation is found using a unification

algorithm. The unification algorithm operates by starting with the empty substitution ($\emptyset$) and sequentially transforming each equation into a substitution. For example, after transforming the first equation $t_0 = t_f \to t_1$ into an empty substitution, the state of the equation and the substitution is as follows.

Equation			Substitution		
t_1	$=$	$t_x \to t_2$	t_0	$=$	$t_f \to t_1$
t_2	$=$	int			
t_3	$=$	int			
t_4	$=$	int			
t_f	$=$	int $\to t_4$			
t_f	$=$	$t_x \to t_3$			

Next, move equation $t_1 = t_x \to t_2$ to the substitution section. If a variable corresponding to the left-hand side of the equation being moved (t_1) exists in the substitution section, replace those variables with the right-hand side of the equation ($t_x \to t_2$). Therefore, when $t_1 = t_x \to t_2$ is moved to the substitution, the

state becomes as follows.

Equation			Substitution		
t_2	$=$	int	t_0	$=$	$t_f \to (t_x \to t_2)$
t_3	$=$	int	t_1	$=$	$t_x \to t_2$
t_4	$=$	int			
t_f	$=$	$\text{int} \to t_4$			
t_f	$=$	$t_x \to t_3$			

In the same way, all remaining equations are moved to the substitutions. However, if a solution for a type variable contained in the equation currently being moved already exists in the substitutions, the substitution is applied to the equation to replace that variable with the solution before moving it. For example, consider the following state.

Equation			Substitution		
t_f	$=$	$\text{int} \to t_4$	t_0	$=$	$t_f \to (t_x \to \text{int})$
t_f	$=$	$t_x \to t_3$	t_1	$=$	$t_x \to \text{int}$
			t_2	$=$	int
			t_3	$=$	int
			t_4	$=$	int

Before moving equation $t_f = \text{int} \to t_4$ to the substitution, we replace variable t_4 with int. This is because a solution for the

type variable t_4 currently exists in the substitution. After moving the equation into the substitution, the state becomes as follows.

Equation	Substitution
$t_f = t_x \to t_3$	$t_0 = (\text{int} \to \text{int}) \to (t_x \to \text{int})$
	$t_1 = t_x \to \text{int}$
	$t_2 = \text{int}$
	$t_3 = \text{int}$
	$t_4 = \text{int}$
	$t_f = \text{int} \to \text{int}$

Next, to process equation $t_f = t_x \to t_3$, we first apply the substitution, which changes the equation as follows.

$$\text{int} \to \text{int} = t_x \to \text{int}$$

In cases like this, where neither the left nor the right side of the equation contains variables, the equation is split. If both sides are function types, for them to be equal, the types of the two arguments and the result must each be the same. The equation

can be split as follows.

Equation			Substitution		
int	=	t_x	t_0	=	$(\text{int} \to \text{int}) \to (t_x \to \text{int})$
int	=	int	t_1	=	$t_x \to \text{int}$
			t_2	=	int
			t_3	=	int
			t_4	=	int
			t_f	=	$\text{int} \to \text{int}$

For equations like $\text{int} = t_x$, where the left-hand side does not contain variables, we swap the sides and then proceed to substitution. The final equation, $\text{int} = \text{int}$, is a case that always holds true, so it is ignored. Ultimately, we obtain the solution corresponding to the right-hand side of Figure 8.8. You can see that the solution includes not only the type of the entire program (t_0) but also the types of all sub-expressions and variables.

There are cases where the unification algorithm cannot find a solution. Consider the following program:

$$\underbrace{\texttt{if } \underbrace{x}_{t_x} \texttt{ then } \underbrace{(x+1)}_{t_1} \texttt{ else } 0}_{t_0}$$

solving the equations leads to a contradiction, as shown below.

Equation	Substitution
bool = int	t_x = bool
t_1 = int	t_1 = int
	t_0 = int

Since no solution can satisfy the equation bool = int, the unification algorithm fails and does not proceed further. This means that type inference cannot be performed for the given program. This is, in fact, a case of a type error.

Let's consider a case where no type error occurs during execution, but type inference fails. For the program below,

$$\underbrace{\texttt{fun}\ (\underbrace{f}_{t_f})\ \underbrace{(\texttt{iszero}\ \underbrace{(f\ f)}_{t_2})}_{t_1}}_{t_0}$$

when we attempt to find a solution, we encounter the following situation.

Equation	Substitution
$t_f = t_f \to \text{int}$	$t_0 = t_f \to \text{bool}$
	t_1 = bool
	t_2 = int

The types we are currently considering are defined using the following syntax:

$$T \to \text{int} \mid \text{bool} \mid T \to T \mid \alpha$$

No type belonging to this set can satisfy the equation $t_f = t_f \to$ int. Therefore, when encountering an equation of this form, the unification algorithm fails to infer the type and terminates. However, in this case, if the function f is a function that takes another function as an argument and returns an integer, the program above will execute without errors. For example, if f is `fun (g) 1`, it will execute without any problems. This is a case where, despite the program containing no type errors, it cannot be accepted due to the limitations of the type system we designed.

Summarizing the examples above, the process of finding a solution to a type equation is as follows. For each individual equation $T_1 = T_2$ in the type equation, repeat the following steps to transform it into a substitution.

- First, apply the current substitution to the equation to update the values of the type variables that are already known.
- If, as a result, the current equation takes a form that always holds true, such as int = int, it is ignored, and the process moves on to the next equation.

- If, on the other hand, a contradiction arises as shown in bool = int, this indicates a failure in type inference, so the algorithm terminates.

- If both sides of the current equation are not variables, as shown in int $\to t_1 = t_2 \to$ bool, simplify the equation until one side becomes a variable.

- If the left-hand side is not a variable, swap the sides of the equation.

- If a variable from the left-hand side appears on the right-hand side, as in $t = \ldots t \ldots$, type inference fails and the algorithm terminates.

- Otherwise, the current equation is moved to the substitution section, and the variable corresponding to the left-hand side of the equation in the substitution is replaced with the variable on the right-hand side.

Algorithm for Solving Type Equations

Let's describe the above process precisely as an algorithm. A substitution can be defined as a function that maps a type variable to a type, as follows.

$$S \in Subst = TyVar \to T$$

The process of applying the substitution S to a type is defined inductively as follows. It is an operation that replaces all type variables present in the substitution with their corresponding types.

$$\begin{aligned} S(\text{int}) &= \text{int} \\ S(\text{bool}) &= \text{bool} \\ S(\alpha) &= \begin{cases} t & \text{if } \alpha \mapsto t \in S \\ \alpha & \text{otherwise} \end{cases} \\ S(T_1 \to T_2) &= S(T_1) \to S(T_2) \end{aligned}$$

For example, the process of applying the following substitution

$$S = \{t_1 \mapsto \text{int}, t_2 \mapsto \text{int} \to \text{int}\}$$

to type $(t_1 \to t_2) \to (t_3 \to \text{int})$ is as follows.

$$\begin{aligned} & S((t_1 \to t_2) \to (t_3 \to \text{int})) \\ &= S(t_1 \to t_2) \to S(t_3 \to \text{int}) \\ &= (S(t_1) \to S(t_2)) \to (S(t_3) \to S(\text{int})) \\ &= (\text{int} \to (\text{int} \to \text{int})) \to (t_3 \to \text{int}) \end{aligned}$$

The process of transforming equation $t_1 \doteq t_2$ into the current substitution can be defined by the following function unify.

$$\mathsf{unify} : T \times T \times \mathit{Subst} \to \mathit{Subst}$$

$\mathsf{unify}(T_1, T_2, S)$ processes the equation $T_1 \doteq T_2$ and transforms it into the substitution S. unify is defined in Figure 8.10. Although not explicitly stated, let us assume that in the case of $\mathsf{unify}(\alpha, t, S)$, extending S involves updating the existing substitution with information from the current equation. Here are a few examples:

- $\mathsf{unify}(\alpha, \text{int} \to \text{int}, \emptyset) = \{\alpha \mapsto \text{int} \to \text{int}\}$
- $\mathsf{unify}(\alpha, \text{int} \to \alpha, \emptyset) = \text{fail}$
- $\mathsf{unify}(\alpha \to \beta, \text{int} \to \text{int}, \emptyset) = \{\alpha \mapsto \text{int}, \beta \mapsto \text{int}\}$
- $\mathsf{unify}(\alpha \to \beta, \text{int} \to \alpha, \emptyset) = \{\alpha \mapsto \text{int}, \beta \mapsto \text{int}\}$.

It is important to understand how the last example works. This corresponds to the case of $\mathsf{unify}(t_1 \to t_2, t_1' \to t_2', S)$ in the algorithm shown in Figure 8.10, it illustrates why, when computing S'', S' must be applied first to t_2 and t_2'.

Let unifyall be the algorithm that takes a complete type equation as input and computes its solution. It is defined as follows.

$$\mathsf{unifyall} : \mathit{TyEqn} \to \mathit{Subst} \to \mathit{Subst}$$

$$\begin{array}{rcl} \mathsf{unifyall}(\emptyset, S) & = & S \\ \mathsf{unifyall}((t_1 \doteq t_2) \wedge u, S) & = & \text{let } S' = \mathsf{unify}(S(t_1), S(t_2), S) \\ & & \text{in } \mathsf{unifyall}(u, S') \end{array}$$

$$
\begin{aligned}
\mathsf{unify}(\mathrm{int}, \mathrm{int}, S) &= S \\
\mathsf{unify}(\mathrm{bool}, \mathrm{bool}, S) &= S \\
\mathsf{unify}(\alpha, \alpha, S) &= S \\
\mathsf{unify}(\alpha, t, S) &= \begin{cases} \text{fail} & \alpha \text{ occurs in } t \\ \text{extend } S \text{ with } \alpha \doteq t & \text{otherwise} \end{cases} \\
\mathsf{unify}(t, \alpha, S) &= \mathsf{unify}(\alpha, t, S) \\
\mathsf{unify}(t_1 \to t_2, t_1' \to t_2', S) &= \text{let } S' = \mathsf{unify}(t_1, t_1', S) \text{ in} \\
&\quad \text{let } S'' = \mathsf{unify}(S'(t_2), S'(t_2'), S') \text{ in} \\
&\quad\quad S'' \\
\mathsf{unify}(\_, \_, \_) &= \text{fail}
\end{aligned}
$$

Figure 8.10: Identification Algorithm

When the type equation u is given, the second argument of unifyall should always be the empty substitution. Let us define the algorithm $\mathcal{U}$, which takes the equation u as input and computes its solution, as follows.

$$\mathcal{U}(u) = \mathsf{unifyall}(u, \emptyset)$$

Final Type Inference Algorithm

Using the algorithms $\mathcal{V}$ and $\mathcal{U}$ defined so far, we can define the type-checking algorithm $\mathsf{typecheck}$ as follows.

$$\begin{aligned}&\mathsf{typecheck}(E) = \\ &\quad \text{let } S = \mathcal{U}(\mathcal{V}(\emptyset, E, \alpha)) \quad (\text{new } \alpha) \\ &\quad \text{in } S(\alpha)\end{aligned}$$

First, a new type variable (α) is created and assumed to be the type of the given program E; then, an equation is generated ($\mathcal{V}(\emptyset, E, \alpha)$) and solved ($\mathcal{U}$). If the equation is solvable, a substitution (S) is derived. From the substitution S, it finds and returns the type ($S(\alpha)$) corresponding to α.

Safety and Completeness of Type Inference Algorithms

The type inference algorithm described above is both safe and complete with respect to the type inference rules. The role of the inference algorithm is to determine whether $\emptyset \vdash E : t$ is provable; this means that for every provable $\emptyset \vdash E : t$, the algorithm can produce a proof (completeness), and if the algorithm succeeds in the inference, the statement is necessarily provable according to the inference rules (safety). This means that the algorithm faithfully implements the type inference rules.

The safety and completeness of this type inference algorithm

are distinct concepts from the safety and completeness of the inference rules explained earlier. While the safety and completeness of the type inference rules are defined with respect to the execution rules, in the case of the algorithm, they are defined with respect to the type inference rules.

Execution rules $\xleftarrow{\text{Safe/Incomplete}}$ Type inference rules $\xleftarrow{\text{Safe/Complete}}$ Type inference

8.7 Polymorphic Type Systems

The type systems discussed so far are called simple type systems. Simple type systems are less sophisticated than the type systems actually used in languages such as OCaml; the main difference is that they do not support polymorphic types. For example, the program below contains no type errors, but it is a classic example of a program that our type system cannot accept, whereas the OCaml type system does.

```
let f = fun (x) x in
  if (f (iszero 0)) then (f 1) else (f 2)
```

Let's analyze why the simple type system cannot perform type checking on the program above.

- When generating a type equation for `let f = fun (x) x`, we introduce the type variables t_x and t_f to generate the

equation $t_f = t_x \rightarrow t_x$. Looking at this equation alone, there is only one constraint that the type of the function `f` must satisfy. Given that the type of the variable `x` is t_x, the type of `f` must be of the form $t_x \rightarrow t_x$. In other words, there are no specific constraints that the argument type t_x must satisfy; the only constraint introduced is that the function's argument type and return type must be the same.

- Intuitively, under these constraints, the program above should pass type checking. This is because, whenever `f` is used, the entire program can be made to have a type while adhering to the above constraints. For the first function call expression `(f (iszero 0))`, let's say it is $t_f = \text{bool} \rightarrow \text{bool}$, and and if we denote the second and third call expressions as $t_f = \text{int} \rightarrow \text{int}$, then the type inference rules for the conditional expressions are satisfied while always meeting the above constraint on `f`. Therefore, there is no logical issue with concluding that the type of the entire program is int.

- Nevertheless, the simple type system fails at type inference because it uses the same type variable t_x to generate equations at different points where `f` is used. At the first function call site, a Boolean-type value is passed as an argu-

ment to `f`, so the equation $t_x = \text{bool}$ is generated, and at the second and third call sites, since integer-type values are passed as arguments to `f`, the equation $t_x = \text{int}$ is generated. Combining these two equations results in a contradiction ($\text{bool} = \text{int}$), causing type inference to fail.

A natural way to solve this problem is to introduce a new type variable at each different point where `f` is used. If we denoted the first call site as $t_f = t_x \to t_x$, then at the second call site, we introduce a new type variable t'_x and set it to $t_f = t'_x \to t'_x$ to generate the equation. As a result, equations such as $t_x = \text{bool}$ and $t'_x = \text{int}$ are generated, thereby avoiding the problem where two logically unrelated equations become entangled and produce a contradiction. In static analysis, this method of distinguishing between different function call sites is generally referred to as context-sensitive analysis, and in the context of type systems, it is called a polymorphic type system. In languages equipped with such a type system, it becomes possible to define and use functions (polymorphic functions) that operate on arbitrary types.

While the basic idea behind extending a simple type system to a polymorphic one is straightforward, the degree of sophistication allows for the design of various type systems. The type system used in OCaml is called the let-polymorphic type sys-

tem, which generalizes only those functions defined by 'let' expressions as polymorphic functions.

8.8 Implementation

Let's implement the type system designed in this chapter. Consider the following language, defined in OCaml.

```
type exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string
```

Types, type environments, and substitution can be implemented as follows.

```
type typ =
  TyInt
  | TyBool
  | TyFun of typ * typ
  | TyVar of tyvar
and tyvar = string

let tyvar_num = ref 0
let fresh_tyvar () =
```

```
  (tyvar_num := !tyvar_num + 1;
   (TyVar ("t" ^ string_of_int !tyvar_num)))

(* type environment : var -> type *)
module TEnv = struct
  type t = var -> typ
  let empty =
    fun _ -> raise (Failure "Type Env is empty")
  let extend (x,t) tenv =
    fun y -> if x = y then t else (tenv y)
  let find tenv x = tenv x
end

(* substitution *)
module Subst = struct
  type t = (tyvar * typ) list
  let empty = []
  let find x subst = List.assoc x subst

  let rec apply : typ -> t -> typ
  =fun typ subst ->
    match typ with
    | TyInt -> TyInt
    | TyBool -> TyBool
    | TyFun (t1,t2) ->
      TyFun (apply t1 subst, apply t2 subst)
    | TyVar x ->
      try find x subst
      with _ -> typ

  let extend tv ty subst =
    (tv,ty) ::
    (List.map (fun (x,t) ->
      (x, apply t [(tv,ty)])) subst)
```

```
end
```

Let's define a type equation as follows.

```
type typ_eqn = (typ * typ) list
```

Let's implement the process of generating type equations and finding their solutions using the functions below.

```
let rec gen_equations : TEnv.t -> exp -> typ -> typ_eqn
=fun tenv e ty -> (* TODO *)

let solve : typ_eqn -> Subst.t
=fun eqn -> (* TODO *)
```

A final type checker `typecheck` can be easily implemented using these functions.

```
let typecheck : exp -> typ
=fun exp ->
  let new_tv = fresh_tyvar () in
  let eqns = gen_equations TEnv.empty exp new_tv in
  let subst = solve eqns in
  let ty = Subst.apply new_tv subst in
    ty
```

Extending the Type System

In this chapter, we designed and implemented a type system for a simple language. Let's extend this to design type inference rules and implement a type checker for the language Fundefined in Chapter 5.

9

Lambda Calculus

The origin of programming languages lies in the lambda calculus, which was introduced by Alonzo Church in 1936. The lambda calculus can be considered the smallest and simplest form of a programming language, containing only the essential features of a programming language. We conclude this book by introducing lambda calculus.

9.1 Lambda Calculus

In 4, we stated that the `let` expression is syntactic sugar that can be rewritten as a function call as follows.

$$\texttt{let}\ x = E_1\ \texttt{in}\ E_2 \overset{desugar}{\Longrightarrow} (\texttt{fun}\ x\ E_2)\ E_1$$

Syntactic sugar is not essential in a programming language. Even if a programming language does not support `let`, the number of programs that can be written is not reduced.

One interesting question arises. What would a programming language look like if all syntactic sugar were removed? For example, consider the language shown in Figure 9.1, as defined in

Chapter 4. Which parts of this language's syntax are syntactic sugar? What would a programming language look like if all syntactic sugar were removed, leaving only the essential elements?

Surprisingly, most of what constitutes a programming language is syntactic sugar. In the language shown in Figure 9.1, integers (n) are syntactic sugar, and all arithmetic operations are syntactic sugar. Conditional expressions and recursive functions are also syntactic sugar, so arbitrary conditions and loops can be expressed without them. The only grammatical structures that are not "sugar" are variables (x), function definitions (`fun` x E), and function calls (E_1 E_2). After removing all "sugar" structures, the programming language looks like this.

$$\begin{array}{rcl} E & \rightarrow & x \\ & | & \texttt{fun}\ x\ E \\ & | & E_1\ E_2 \end{array}$$

This language is a lambda calculus. Since any program that can be written in a general-purpose programming language can also be written in lambda calculus, lambda calculus can be considered the smallest programming language that is Turing-complete.

$$
\begin{array}{rcl}
E & \rightarrow & n \\
& | & x \\
& | & E_1 \texttt{ + } E_2 \\
& | & E_1 \texttt{ - } E_2 \\
& | & E_1 \texttt{ * } E_2 \\
& | & \texttt{iszero } E \\
& | & \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\
& | & \texttt{let } x \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{letrec } f(x) \texttt{ = } E_1 \texttt{ in } E_2 \\
& | & \texttt{fun } x \ E \\
& | & E_1 \ E_2
\end{array}
$$

Figure 9.1: 4The programming language defined in Chapter

Syntax

Lambda calculus has a very simple syntactic and semantic structure. Traditional lambda calculus has the following syntactic structure.

$$
\begin{array}{rcll}
E & \rightarrow & x & \text{variable} \\
& | & \lambda x.E & \text{function definition} \\
& | & E_1 \ E_2 & \text{function call}
\end{array}
$$

The only difference is that the function definition `fun` x E is expressed as $\lambda x.E$. For example, the expressions below are all lambda expressions that can be created using the syntax described

above.

$$x \qquad y \qquad z$$

$$\lambda x.x \qquad \lambda x.y \qquad \lambda x.\lambda y.x$$

$$x\ y \qquad (\lambda x.x)\ z \qquad x\ \lambda y.z \qquad ((\lambda x.x)\ \lambda x.x)$$

If parentheses are not used sufficiently, the scope of the function body in a lambda expression may be unclear. For example, it is ambiguous whether the lambda expression $\lambda x.\lambda y.x\ y$ refers to $\lambda x.\lambda y.(x\ y)$ or $(\lambda x.\lambda y.x)\ y$. In such cases, it is customary to interpret it as a function body whenever possible. This means interpreting the lambda expression $\lambda x.\lambda y.x\ y$ as $\lambda x.\lambda y.(x\ y)$.

Free variables and bound variables can also be defined for lambda expressions. If, for a given variable x, there exists a λx in which that variable is introduced, it is called a bound variable. All other variables are free variables. For example,

- In the lambda expression $\lambda y.(x\ y)$, x is a free variable and y is a bound variable.
- In the lambda expression $\lambda z.\lambda x.\lambda y.(y\ z)$, the variables y and z are both bound variables.
- In the lambda expression $(\lambda x.x)\ x$, the first x is a bound variable, and the second x is a free variable.

Evaluation of Lambda Expressions

To evaluate the lambda expression E, repeat the following two steps.

1. In the given lambda expression E, find sub-expressions of the following form.

$$(\lambda x.E_1)\ E_2$$

   This takes the form of calling the function $\lambda x.E_1$ with the argument E_2; such a sub-expression is called a redex (re-dex[1)]).

   For example, in the lambda expression $(\lambda x.x)\ y$, the entire expression is the re-dex, whereas in the lambda expression $\lambda x.((\lambda y.y)\ z)$, the sub-expression $(\lambda y.y)\ z$ is the re-dex.

2. Redex is rewritten as follows. This process is called β-computation (β-reduction).

$$(\lambda x.E_1)\ E_2 \to [x \mapsto E_2]E_1$$

   Here, $[x \mapsto E_2]E_1$ refers to the expression obtained by substituting the variable x, which appears in expression E_1,

1) reducible expression

with expression E_2. For example, the two lambda expressions above are evaluated as follows.

$$(\lambda x.x)\ y \rightarrow [x \mapsto y]x = y$$
$$\lambda x.((\lambda y.y)\ z) \rightarrow \lambda x.([y \mapsto z]y) = \lambda x.z$$

Repeat this process until there are no more Redexes. A lambda expression for which no further computation is possible because there are no more Redexes is called a normal term.

Let's define the substitution $[x \mapsto E_2]E_1$ precisely. It is defined recursively as follows, based on the structure of E_1.

$$\begin{aligned}
[x \mapsto E_2]x &= E_2 \\
[x \mapsto E_2]y &= y \quad (x \neq y) \\
[x \mapsto E_2](\lambda y.E_1') &= \lambda z.[x \mapsto E_2]([y \mapsto z]E_1') \quad (\text{new } z) \\
[x \mapsto E_2](E_1'\ E_1'') &= ([x \mapsto E_2]E_1'\ [x \mapsto E_2]E_1'')
\end{aligned}$$

It is important to consider the case where E_1 is a function definition $(\lambda y.E_1')$. When substituting the variable x with E_2 in E_1', the bound variable must not be substituted, nor should a new variable be unintentionally bound. To achieve this, the function argument name y is changed to a new name z that does not appear anywhere in the entire lambda expression, and then the substitution $[x \mapsto E_2]$ is applied.

For example, the lambda expression $(\lambda x.(\lambda x.x))\ y$ is evalu-

ated as follows.

$$\begin{aligned}(\lambda x.(\lambda x.x))\ y &\rightarrow [x \mapsto y](\lambda x.x) \\ &= \lambda z.[x \mapsto y]([x \mapsto z]x) \quad (\text{new } z) \\ &= \lambda z.[x \mapsto y](z) \\ &= \lambda z.z\end{aligned}$$

When evaluating $[x \mapsto y](\lambda x.x)$, x within $\lambda x.x$ is not a free variable; therefore, its name should not be changed by the substitution $[x \mapsto y]$. To prevent this, the name of the bound variable was changed to a new name before applying the substitution to the body of the function. Since the names of arguments in a function do not affect its meaning, the final result is semantically equivalent whether it is $\lambda z.z$ or $\lambda x.x$. If the new name z had not been introduced during the substitution process above, the calculation would have proceeded as follows:

$$\begin{aligned}(\lambda x.(\lambda x.x))\ y &\rightarrow [x \mapsto y](\lambda x.x) \\ &= \lambda x.[x \mapsto y](x) \\ &= \lambda x.y\end{aligned}$$

The function $\lambda x.x$, which returns the argument as-is, should have been evaluated, but it was incorrectly evaluated as the function $\lambda x.y$, which has a different meaning.

Renaming arguments in this way can also prevent unintended

variable binding. For example, when evaluating the lambda expression $(\lambda x.(\lambda y.(x\ y)))\ y$, if the name of the bound variable y is not renamed, the result will be $\lambda y.(y\ y)$. However, y inside $\lambda y.(x\ y)$ and y, which is passed as an argument, are unrelated variables. Since unrelated variables have become related due to substitution, this is an incorrect result. In $\lambda y.(x\ y)$, if we rename y to z and then evaluate the expression, we obtain $\lambda z.(y\ z)$, thereby preventing free variables from being unintentionally bound.

A lambda expression E may contain multiple redexes. For example, the lambda expression below contains three redexes.

$$\underbrace{\lambda x.x\ \underbrace{(\lambda x.x\ \underbrace{(\lambda z.\underbrace{(\lambda x.x)}_{redex3}\ z)}_{redex2})}_{redex1}}$$

Generally, depending on which of several re-dexes is chosen to be evaluated first, the evaluation of a lambda expression may or may not reach a normal form and terminate. For a lambda expression that has a normal form, evaluating it in normal order ensures that it always reaches that normal form. Normal order is the method of evaluating the outermost, leftmost redex first. For example, the process of evaluating the lambda expression above in normal order is as follows. The redex evaluated at each step

is marked with an underscore.

$$
\begin{aligned}
&\underline{\lambda x.x\ (\lambda x.x\ (\lambda z.(\lambda x.x)\ z))} \\
&\to (\lambda x.x\ \underline{(\lambda z.(\lambda x.x)\ z)}) \\
&\to (\lambda z.\underline{(\lambda x.x)\ z}) \\
&\to \lambda z.z
\end{aligned}
$$

Even if the order of evaluation changes, as long as the evaluation of the lambda expression is complete, the result is always the same.[2)] For example, even if the lambda expression above is evaluated starting from the innermost redex, the result is $\lambda z.z$.

$$
\begin{aligned}
&\lambda x.x\ (\lambda x.x\ (\lambda z.\underline{(\lambda x.x)\ z})) \\
&\to \lambda x.x\ \underline{(\lambda x.x\ (\lambda z.z))} \\
&\to \underline{\lambda x.x\ (\lambda z.z)} \\
&\to \lambda z.z
\end{aligned}
$$

9.2 Converting to a Lambda Expression

We stated that if we remove all sugar structures from the language in Figure 9.1, it becomes a lambda expression. Given a program E written in the language of Figure 9.1, let's define the process of converting it into a lambda expression with the same

2) This is known as the Church–Rosser Theorem or the Confluence Theorem.

$$
\begin{array}{rcl}
\underline{\mathit{true}} & = & \lambda t.\lambda f.t \\
\underline{\mathit{false}} & = & \lambda t.\lambda f.f \\
\underline{0} & = & \lambda s.\lambda z.z \\
\underline{1} & = & \lambda s.\lambda z.(s\ z) \\
\underline{n} & = & \lambda s.\lambda z.(s^n\ z) \\
\underline{x} & = & x \\
\underline{E_1 + E_2} & = & (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ \underline{E_1}\ \underline{E_2} \\
\underline{\texttt{iszero}\ E} & = & (\lambda m.m\ (\lambda x.\underline{\mathit{false}})\ \underline{\mathit{true}})\ \underline{E} \\
\underline{\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3} & = & \underline{E_1}\ \underline{E_2}\ \underline{E_3} \\
\underline{\texttt{let}\ x = E_1\ \texttt{in}\ E_2} & = & (\lambda x.\underline{E_2})\ \underline{E_1} \\
\underline{\texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2} & = & \underline{\texttt{let}\ f = Y\ (\lambda f.\lambda x.E_1)\ \texttt{in}\ E_2} \\
\underline{\texttt{fun}\ x\ E} & = & \lambda x.\underline{E} \\
\underline{E_1\ E_2} & = & \underline{E_1}\ \underline{E_2}
\end{array}
$$

Figure 9.2: Rules for converting to lambda calculus

meaning. Let's denote the result of converting the expression E into a lambda expression as $\underline{E}$. The conversion rules can be defined recursively as shown in Figure 9.2.

First, the Boolean values *true* and *false* were defined as the functions $\lambda t.\lambda f.t$ and $\lambda t.\lambda f.f$, respectively. *true* is a function that takes two arguments, t and f, and returns the first argument, t; and we have agreed to represent *false* as a function that returns the second argument, f. The conditional expression `if` E_1 `then` E_2 `else` E_3 was transformed as shown in $\underline{E_1}\ \underline{E_2}\ \underline{E_3}$. If we evaluate E_1, we will ultimately obtain a function $\lambda t.\lambda f.t$ or

$\lambda t.\lambda f.f$, which represents a Boolean value. Therefore, if the original meaning of E_1 was *true*, then $\underline{E_2}$ would be computed, and if the meaning of E_1 is *false*, then $\underline{E_3}$ would be computed. For example, when the expression `if` *true* `then` 0 `else` 1 is transformed and evaluated, the lambda expression 0—which means 0—is evaluated as follows.

$$\begin{aligned}
\underline{\texttt{if } \textit{true} \texttt{ then 0 else 1}} &= \underline{\textit{true}}\ \underline{0}\ \underline{1} \\
&= (\lambda t.\lambda f.t)\ \underline{0}\ \underline{1} \\
&\rightarrow (\lambda f.\underline{0})\ \underline{1} \\
&\rightarrow \underline{0}
\end{aligned}$$

Let's consider the case where n is a natural number.[3)] In Figure 9.2, the natural number n is represented by the lambda expression $\lambda s.\lambda z.(s^n\ z)$. Here, $s^n\ z$ is a lambda expression defined as follows.

$$\begin{aligned}
s^0\ z &= z \\
s^n\ z &= s\ (s^{n-1}\ z)
\end{aligned}$$

For example, the natural number 0 is encoded as a function that takes two arguments, s and z, and returns z. The natural number 1 is encoded as a function that takes the argument s ap-

3) Integers can be encoded using natural numbers, but this is omitted here.

$$
\begin{array}{lcl}
\underline{1+2} & = & (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ \underline{1}\ \underline{2} \\
& \rightarrow & (\lambda m.\lambda s.\lambda z.m\ s\ (\underline{1}\ s\ z))\ \underline{2} \\
& \rightarrow & \lambda s.\lambda z.\underline{2}\ s\ (\underline{1}\ s\ z) \\
& = & \lambda s.\lambda z.\underline{2}\ s\ (\lambda s.\lambda z.(s\ z)\ s\ z) \\
& \rightarrow & \lambda s.\lambda z.\underline{2}\ s\ (\lambda z.(s\ z)\ z) \\
& \rightarrow & \lambda s.\lambda z.\underline{2}\ s\ (s\ z) \\
& = & \lambda s.\lambda z.(\lambda s.\lambda z.(s\ (s\ z)))\ s\ (s\ z) \\
& \rightarrow & \lambda s.\lambda z.(\lambda z.(s\ (s\ z)))\ (s\ z) \\
& \rightarrow & \lambda s.\lambda z.s\ (s\ (s\ z)) \\
& = & \underline{3}
\end{array}
$$

Figure 9.3: The process of converting $1+2$ to a lambda expression and evaluating it

plied once to z, and 2 is encoded as a function that applies s to z twice. In general, n is a function that applies s to z n times. If we agree to represent natural numbers in this way, the addition operation $E_1 + E_2$ can be expressed as follows.

$$\underline{E_1 + E_2} = (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ \underline{E_1}\ \underline{E_2}$$

For example, if we convert the expression $1 + 2$ into a lambda expression and evaluate it, we obtain a lambda expression representing 3, as shown in Figure 9.3.

The expression `iszero` E is converted to $(\lambda m.m\ (\lambda x.\underline{\mathit{false}})\ \underline{\mathit{true}})\ \underline{E}$. Since the type of E is an integer, if we convert E to a lambda expression ($\underline{E}$) and evaluate it, we will ultimately obtain a function of the form $\lambda s.\lambda z.(s^k\ z)$. Let's call this m and evaluate the

function call m $(\lambda x.\underline{\mathit{false}})$ $\underline{\mathit{true}}$. If k is 0, then $\underline{\mathit{true}}$ is returned; if k is not 0, we obtain the result of applying $\underline{\mathit{true}}$ to $\lambda x.\underline{\mathit{false}}$ k times. For any k, the result is $\underline{\mathit{false}}$.

Any recursive function `letrec` $f(x) = E_1$ `in` E_2 can be converted into the following non-recursive function.

$$\texttt{let}\ f = Y\ (\lambda f.\lambda x.E_1)\ \texttt{in}\ E_2$$

Here, Y represents a lambda expression known as the Y combinator, which looks like this:

$$Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

For example, the recursive function

$$\texttt{letrec}\ f(n) = \text{if}\ n = 0\ \text{then}\ 1\ \text{else}\ n * f(n-1)$$

can be converted into the following non-recursive function.

$$\texttt{let}\ \text{fact} = Y(\lambda f.\lambda n.\text{if}\ n = 0\ \text{then}\ 1\ \text{else}\ n * f(n-1))$$

After defining it this way, evaluating fact n yields $n!$.

For example, Figure 9.4 shows the process of evaluating fact 1 in normal order. In this evaluation, F and G refer to the follow-

$$
\begin{aligned}
&\text{fact } 1\\
&= (Y\ F)\ 1\\
&= \underline{(\lambda f.((\lambda x.f(x\ x))(\lambda x.f(x\ x)))\ F)}\ 1\\
&= ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ 1\\
&= \underline{(G\ G)}\ 1\\
&= (F\ (G\ G))\ 1\\
&= \underline{(\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (G\ G)(n-1))\ 1}\\
&= \text{if } \underline{1 = 0} \text{ then } 1 \text{ else } 1 * (G\ G)(1-1))\\
&= \underline{\text{if false then } 1 \text{ else } 1 * (G\ G)(1-1))}\\
&= 1 * (\underline{G\ G})(1-1)\\
&= 1 * (F\ (G\ G))(1-1)\\
&= 1 * \underline{(\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (G\ G)(n-1))(1-1)}\\
&= 1 * \text{if } \underline{(1-1)} = 0 \text{ then } 1 \text{ else } (1-1) * (G\ G)((1-1)-1)\\
&= 1 * \text{if } \underline{0 = 0} \text{ then } 1 \text{ else } (1-1) * (G\ G)((1-1)-1)\\
&= 1 * \underline{\text{if } \mathit{true} \text{ then } 1 \text{ else } (1-1) * (G\ G)((1-1)-1)}\\
&= 1 * 1
\end{aligned}
$$

Figure 9.4: The process of evaluating fact 1 in normal order. Where the step does not involve rewriting using the definition of Y, F, G, the rexes to which the β evaluation applies are underlined.

ing expressions.

$$
\begin{aligned}
F &= \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)\\
G &= \lambda x.F(x\ x)
\end{aligned}
$$

The key step is the process of evaluating the expression $G\ G$ (β-reduction) to produce $F\ (G\ G)$, an expression that contains

itself.

$$\begin{aligned} G\ G &= (\lambda x.F(x\ x))\ (\lambda x.F(x\ x)) \\ &\rightarrow F((\lambda x.F(x\ x))\ (\lambda x.F(x\ x))) = F(G\ G) \end{aligned}$$

Since the Y combinator already possesses this recursive property, we can express iteration without using recursive functions.

9.3 Implementation

Let's define the language covered in this chapter (Figure 9.1) and lambda expressions as OCaml data types as follows.

```
  type exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string

type lambda =
  | LVAR of var
  | LPROC of var * lambda
  | LCALL of lambda * lambda
```

1. Let's write a function `reduce` that evaluates lambda expressions in normal order.

```
reduce : lambda -> lambda
```

2. Let's write a function `translate` that translates the target language into a lambda expression.

```
translate : exp -> lambda
```

For example,

```
reduce(translate (ADD(CONST 1, CONST 2)))
```

should result in the lambda expression

```
LPROC ("s",
 LPROC ("z",
  LCALL (LVAR "s",
    LCALL (LVAR "s",
      LCALL (LVAR "s", LVAR "z")))))
```

, which represents 3 (the variable names may differ).