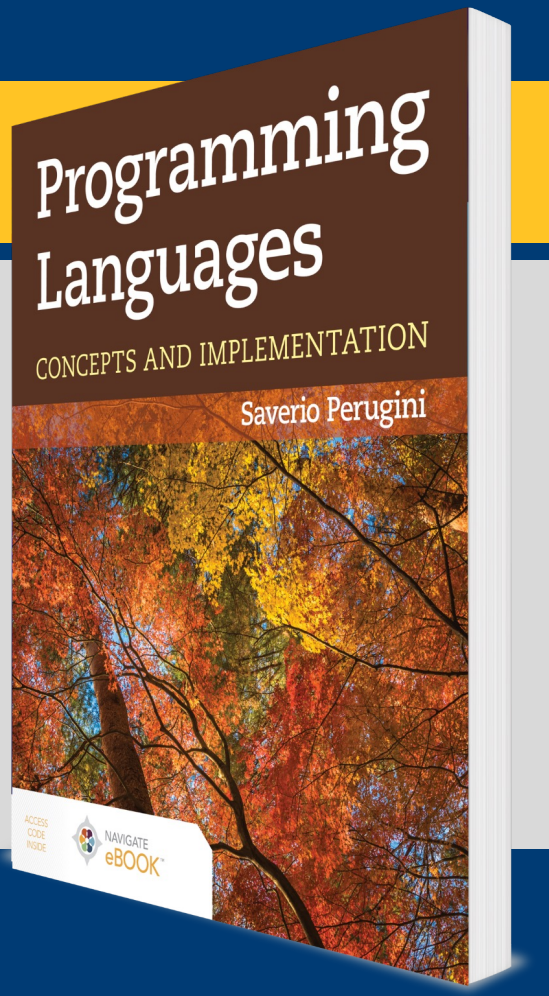
CHAPTER 9

Data Abstraction



Chapter 9: Type Systems and Data Abstraction

Reimplementing [familiar] algorithms and data structures in a significantly different language often is an aid to understanding of basic data structure and algorithm concepts.

—Jeffrey D. Ullman, Elements of ML Programming (1997)

Type systems support data abstraction and, in particular, the definition of user-defined data types that have the properties and behavior of primitive types.

Outline

- **9.1 Chapter Objectives**
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.1 Chapter Objectives

- Introduce *aggregate data types* (e.g., arrays, records, unions) and type systems supporting their construction in a variety of programming languages.
- Introduce *inductive data types*—an aggregate data type that refers to itself—and *variant records*—a data type useful as a node in a tree representing a computer program.
- Introduce *abstract syntax* and its role in a representing a computer program.
- Describe the design, implementation, and manipulation of efficacious and efficient data structures representing computer programs.
- Explore the conception and use of a data structure as an *interface*, *implementation*, and *application*, which render it an *abstract data type*.
- Recognize and use a *closure representation* of a data structure.
- Describe the design and implementation of data structures for language environments using a variety of representations.

Outline

- 9.1 Chapter Objectives
- **9.2 Aggregate Data Types**
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.2 Aggregate Data Types

- 9.2.1 Arrays
- 9.2.2 Records
- 9.2.3 Undiscriminated Unions
- 9.2.4 Discriminated Unions

9.2 Aggregate Data Types: Arrays and Records

- An *array* is an aggregate data type indexed by integers:

```
/* declaration of integer array scores */
int scores[10];
/* use of integer array scores */
scores[0] = 97;
scores[1] = 98;
```

- A *record* (also referred to as a `struct`) is an aggregate data type indexed by strings called field names:

```
/* declaration of struct employee */
struct {
   int id;
   double rate;
} employee;
/* use of struct employee */
employee.id = 555;
employee.rate = 7.25;
```

9.2.3 Undiscriminated Unions

An *undiscriminated* union is an aggregate data type that can only store a value of one of multiple types (i.e., a union of multiple types):

```
/* declaration of an undiscriminated union int_or_double */
union {
  /* C compiler only allocates memory for the largest */
  int id;
  double rate;
} int_or_double;
int main() {
   /* use of union int_or_double */
   int_or_double.id = 555;
   int_or_double.rate = 7.25;
}
```

9.2.4 Discriminated Unions

- A *discriminated* union is a record containing a `union` as one field and a flag as the other field.
- The flag indicates the type of the value currently stored in the union.

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- **9.3 Inductive Data Types**
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.3 Inductive Data Types

- An *inductive data type* is an aggregate data type that refers to itself.
- The type being defined is one of the constituent types of the type being defined.
- A node in a singly linked list is a classical example of an inductive data type.
- The node contains some value and a pointer to the next node, which is also of the same node type:

```
struct node_tag {
   int id;
   struct node_tag* next;
};
struct node_tag head;
```

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- **9.4 Variant Records**
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.4 Variant Records

- A *variant record* is an aggregate data type that is a union of records (i.e., a `union` of `struct`s).
- It can hold any one of a variety of records.
- Each constituent record type is called a *variant* of the union type.
- Variant records can also be inductive.

Variant Record Example

Consider the following EBNF definition of a list:

<L> ::= <A>
<L> ::= <A><L>
<A> ::= $-2^{31} \mid \ldots \mid 2^{31} - 1$ (i.e., ints)

```
1  /* a variant record: a union of structs */
2  #include<stdio.h>
3
4  int main() {
5
6     typedef union llist_tag {
7
8        /* <L> ::= <A> variant */
9        struct aatom_tag {
10          int number;
11       } aatom;
12
```

9.4.1 Variant Records in Haskell

- *Type systems* support *data abstraction* and, in particular, the definition of user-defined data types that have the properties and behavior of primitive types.
- A *type system* of a programming language includes the mechanism for creating new data types from existing types.
- ML and Haskell each have a powerful type system.

Table 9.1 Support for C/C++ Style structs and unions in ML, Haskell, Python, and Java

Data Type	C/C++	ML	Haskell	Python	Java
records	`struct`	`type`	`type`	`class`	`class`
unions/variant records	`union`	`datatype`	`data`	`class`	`class`

9.4.2 Variant Records in Scheme: `(define-datatype ...)` and `(cases ...)`

- Unlike ML and Haskell, Scheme does not have built-in support for defining and manipulating variant records, so we need a tool for these tasks in Scheme.
- The `(define-datatype ...)` and `(cases ...)` extensions to Racket Scheme created by Friedman, Wand, and Haynes (2001) provide support for constructing and deconstructing respectively, variant records in Scheme.

(define-datatype ...)

- The `(define-datatype ...)` form defines variant records.

```
#lang eopl
(define-datatype llist llist?
   (aatom
      (aatom_tag number?))
   (aatom_llist
      (aatom_tag number?)
      (next llist?)))
```

Syntax:

(define-datatype <*type-name*> <*type-predicate-name*>
{(<*variant-name*> {(<*fieldname*> <*predicate*>)}*)}+)

- A new function called a *constructor* is automatically created for each variant to construct data values belonging to that variant.
- Data types can be mutually-recursive (e.g., recall grammar for S-expressions)

A List of Integers (1 of 2)

This definition automatically creates a linked list variant record and an *implementation* of the following *interface*:

- A unary function `aatom`, which creates an atom node
- A binary function `aatom_llist`, which creates an atom list node
- A binary predicate `llist?`

(cases ...)

- The `(cases ...)` form, in the EOPL extension to Racket Scheme, provides support for decomposing and manipulating the constituent parts of a variant record created with the constructors automatically generated with the `(define-datatype ...)` form.

Syntax:

```
(cases <type-name> <expression>
  {(<variant-name> ({<field-name>}*) <consequent>)}*
  (else <default>))
```

- `(cases ...)` provides a convenient way to manipulate data types created with `(define-datatype ...)`
- Can think of `(cases ...)` as pattern matching (values bound to symbols)

A List of Integers (2 of 2)

The following function accepts a value of type `llist` as an argument and manipulates its fields with the `cases` form to sum its nodes:

```
(define llist_sum
   (lambda (ll)
      (cases llist ll
         (aatom (aatom_tag) aatom_tag)
         (aatom_llist (aatom_tag next)
            (+ aatom_tag (llist_sum next))))))
> (llist_sum ouraatom)

3
> (llist_sum ouraatom_llist)
5
> (llist_sum ourllist)
6
```

Table 9.2 Support for Composition (Definition) and Decomposition (Manipulation) of Variant Records in a Variety of Programming Languages

Language	Composition	Decomposition
C	`union` of `structs` with flag	`switch (...) { case ...}`
C++	`union` of `structs` with flag	`switch (...) { case ...}`
Java	`class` with flag	`switch (...) { case ...}`
Python	`class` with flag	`if/elif/else`
ML	`datatype`	pattern-directed invocation
Haskell	`data`	pattern-directed invocation
Racket Scheme	`define-datatype` form	`cases` form

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- **9.5 Abstract Syntax**
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.5 Abstract Syntax (1 of 5)

- Consider the string `((lambda (x) (f x)) (g y))` representing an expression in λ-calculus.
- This program string is an external representation (i.e., it is external to the system processing it) and uses *concrete syntax*.
- Programs in concrete syntax are not readily processable.

9.5 Abstract Syntax (2 of 5)

Notably, the preceding program is more manipulable and, thus, processable when represented using the following definition of an `expression` data type:

```
(define-datatype expression expression?
   (variable-expression
     (identifier symbol?))
   (lambda-expression
      (identifier symbol?)
      (body expression?))
   (application-expression
      (operator expression?)
      (operand expression?)))
```

9.5 Abstract Syntax (3 of 5)

- An *abstract-syntax tree* (AST) is similar to a parse tree, except that it uses abstract syntax or an *internal representation* (i.e., it is internal to the system processing it) rather than *concrete syntax*.
- While the structure of a parse tree depicts how a sentence (in concrete syntax) conforms to a grammar:
  - the structure of an abstract-syntax tree illustrates how the sentence is represented internally,
  - typically with an inductive, variant record data type.
- Figure 9.1 illustrates an AST for the λ-calculus expression.

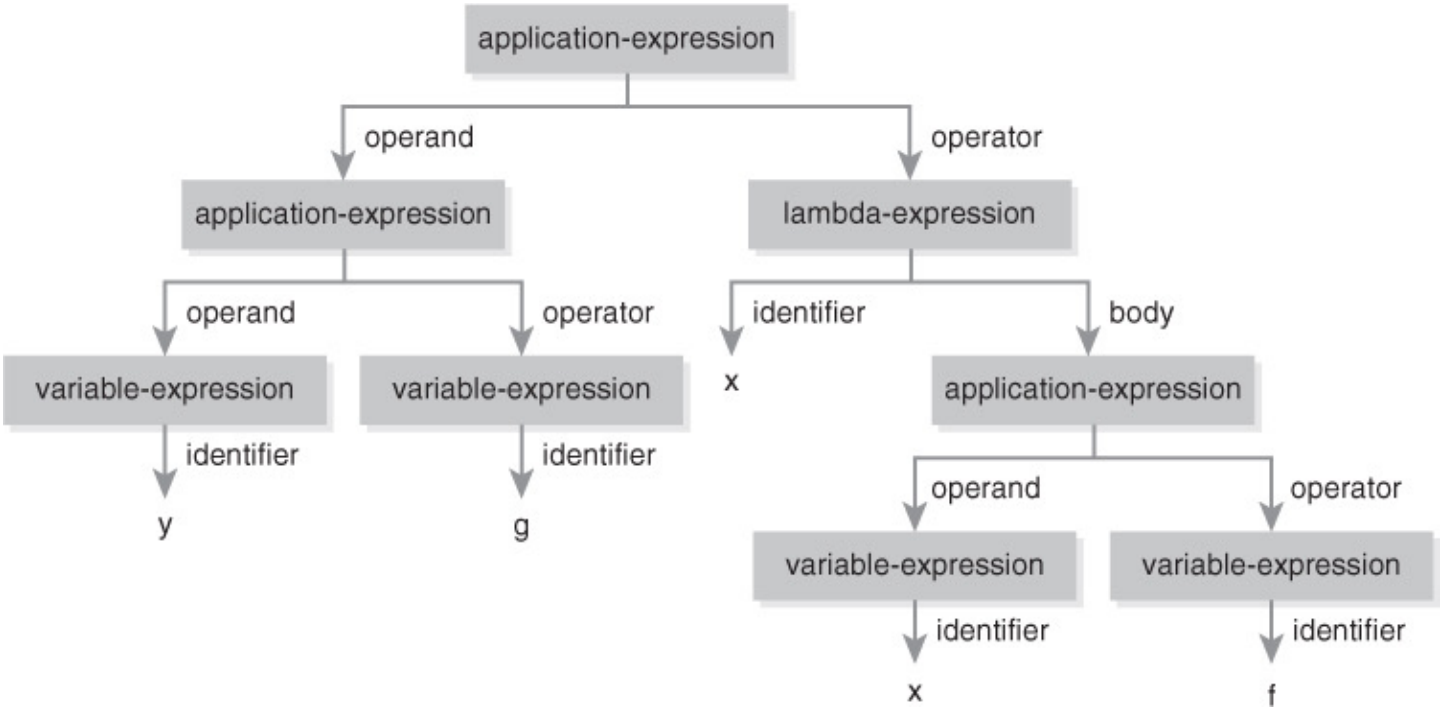
```
((lambda (x) (f x)) (g y))
```

9.5 Abstract Syntax (4 of 5)

- Abstract syntax is a representation of a program as a data structure—in this case, an inductive variant record.
- Consider the following grammar for λ-calculus, which is annotated with variants of this expression inductive variant record data type above the right-hand side of each production rule:

```
                                  variable-expression (identifier)
<expression>   ::=   <identifier>
                                  lambda-expression (identifier body)
<expression>   ::=   (lambda (<identifier>) <expression>)
                                  application-expression (operator operand)
<expression>   ::=   (<expression><expression>)
```

Figure 9.1 Abstract-Syntax Tree for `((lambda (x) (f x)) (g y))`



9.5 Abstract Syntax (5 of 5)

- Simple grammar for λ-calculus expressions revisited
- *Concrete* vs. *abstract* syntax (external vs. internal representation)
- `expression` datatype
- One-to-one mapping between production rules and constructors

Inductive Data Types and Abstract Syntax Summary

- Discriminated unions
- *Inductive data types* (e.g., *variant record*: a `union` of `struct`s)
- `define-datatype` constructs inductive data types (specifically, *variant records*)
- `cases` decomposes inductive data types
- *Concrete syntax* vis-à-vis *abstract syntax*

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- **9.6 Abstract-Syntax Tree for Camille**
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.6 Abstract-Syntax Tree for Camille

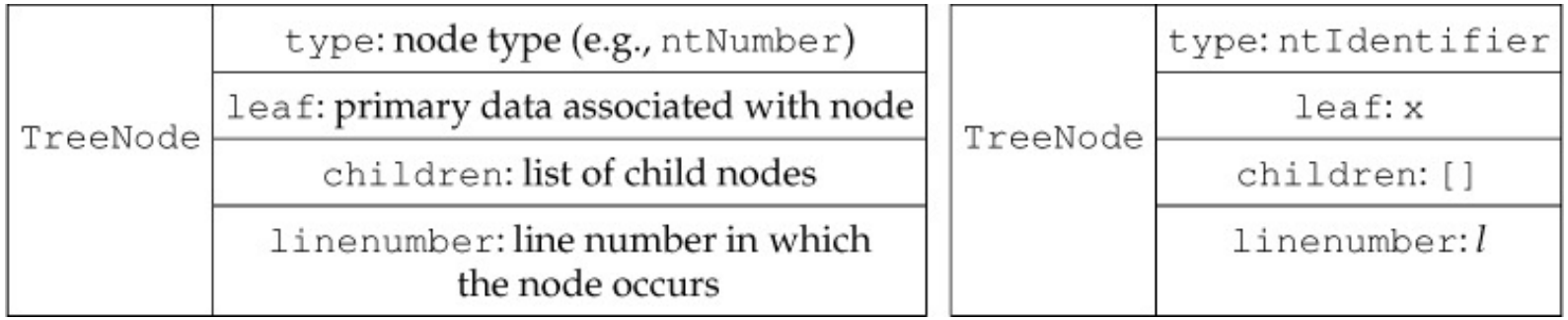
A goal of Part II of this text is to establish an understanding of data abstraction techniques so we can harness them in our construction of environment-passing interpreters, for purposes of simplicity and efficiency, in Part III.

9.6.1 Camille Abstract-Syntax Tree Data Type: `TreeNode`

The following abstract-syntax tree data type `TreeNode` is used in the abstract-syntax trees of Camille programs for our Camille interpreters developed in Part III:

```
41  class Tree_Node:
42     def __init__(self,type,children, leaf, linenumber):
43        self.type = type
44        # save the line number of the node so run-time
45        # errors can be indicated
46        self.linenumber = linenumber
47        if children:
48           self.children = children
49        else:
50           self.children = [ ]
51        self.leaf = leaf
52  # end expression data type #
```

Figure 9.2 (left) Visual Representation of `TreeNode` Python `class`. (right) A Value of Type `TreeNode` for an Identifier



Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- **9.7 Data Abstraction**
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.7 Data Abstraction

Data abstraction involves the conception and use of a data structure as:

- an *interface*, which is implementation-neutral and contains function declarations;
- an *implementation*, which contains function definitions; and
- an *application*, which is also *implementation-neutral* and contains invocations to functions in the implementation; the application is sometimes called the *main program* or *client code*.

- The underlying implementation can change without disrupting the client code as long as the contractual signature of each function declaration in the interface remains unchanged.
- In this way, the implementation is *hidden* from the application.
- A data type developed this way is called an *abstract data type* (ADT).

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- **9.8 Case Study: Environments**
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- 9.10 Thematic Takeaways

9.8 Case Study: Environments (1 of 4)

- 9.8.1 Choices of Representation
- 9.8.2 Closure Representation in Scheme
- 9.8.3 Closure Representation in Python
- 9.8.4 Abstract-Syntax Representation in Python

9.8 Case Study: Environments (2 of 4)

- A *referencing environment* is a mapping that associates variable names (or symbols) with their current bindings at any point in a program in an implementation of a programming language
- A *symbol* table is an example of an environment.
- A symbol table is used in a compiler to associate variable names with lexical address information.
- An environment is a *mapping* (a set of pairs)
  - *domain*: the finite set of Scheme symbols
  - *range*: the set of all Scheme values

(e.g., {(a, 4), (b, 2), (c, 3), (x, 5)}).

9.8 Case Study: Environments (3 of 4)

Consider an interface specification of an environment, where formally an environment expressed in the mathematical form

$$\{(s_1, v_1), (s_2, v_2), \ldots, (s_n, v_n)\}$$

is a mapping (or a set of pairs) from the *domain*—the finite set of Scheme symbols—to the *range*—the set of all Scheme values:

$$\begin{aligned} \texttt{(empty-environment)} &= \lceil \varnothing \rceil \\ \texttt{(apply-environment } \lceil f \rceil s\texttt{)} &= f(s) \\ \texttt{(extend-environment '(}s_1, s_2, \ldots s_n\texttt{) '(}v_1, v_2, \ldots v_n\texttt{)} \lceil f \rceil\texttt{)} &= \lceil g \rceil, \end{aligned}$$

where $g(s') = v_i$ if $s' = s_i$ for some i, $1 \leqslant i \leqslant n$, and $f(s')$ otherwise;

$\lceil v \rceil$ means "the representation of data v."

9.8 Case Study: Environments (4 of 4)

- The environment {(a, 4), (b, 2), (c, 3), (x, 5)} may be constructed and accessed with the following client code:

```
> (define simple-environment
     (extend-environment '(a b) '(1 2)
        (extend-environment '(c d e) '(3 5 5)
           (empty-environment))))
> (apply-environment simple-environment 'e)
5
```

- *Constructors* create: `empty-environment` and `extend-environment`
- *Observers* extract: `apply-environment`

9.8.1 Choices of Representation

- We consider the following representations for an environment:
  - Data structure representation (e.g., lists)
  - Abstract-syntax representation (ASR)
  - Closure representation (CLS)

9.8.2 Closure Representation in Scheme (1 of 2)

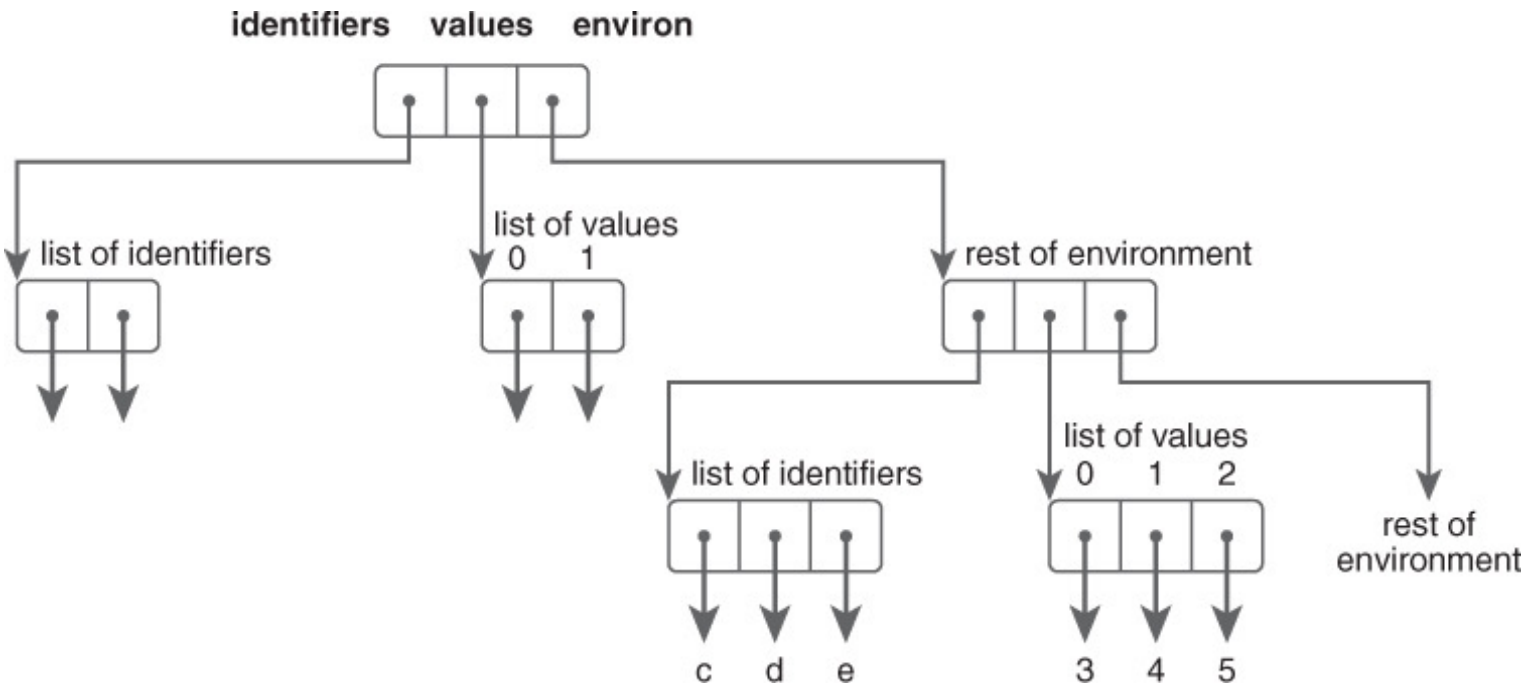
- Often the set of *values* of a data type can be advantageously *represented* as a set of *functions*, particularly when the abstract data type has multiple constructors but only a single observer.
- Moreover, languages with first-class functions facilitate use of a closure representation.
- Representing a data structure as a function—here, a closure—is a non-intuitive use of functions, because we do not typically think of data as code.
- How can we represent an environment (which we think of as a data structure) as a function?
- The most natural closure representation for the environment is a Scheme closure that accepts a symbol and returns its associated value.

9.8.2 Closure Representation in Scheme (2 of 2)

- Getting acclimated to the reality that the data structure is a function can be a cognitive challenge.
- Step through the evaluation of the following application code:

```
1  > (define simple-environment
2       (extend-environment '(a b) '(1 2)
3          (extend-environment '(c d e) '(3 4 5)
4             (empty-environment))))
5
6  > (apply-environment simple-environment 'e)
7  5
```

Figure 9.3 An Abstract-Syntax Representation of a Named Environment in Python



Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- **9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis**
- 9.10 Thematic Takeaways

9.9.3 Comparison of ML and Haskell

Haskell	= ML	+ Lazy Evaluation	- Side Effects
ML	= Lisp	- Homoiconicity	+ Safe Type System
Haskell	= Lisp	- Homoiconicity	+ Safe Type System
		- Side Effects	+ Lazy Evaluation

Table 9.7 Comparison of the Main Concepts and Features of ML and Haskell

Concept	ML	Haskell
lists	homogeneous	homogeneous
cons	`::`	`:`
append	`@`	`++`
integer equality	`=`	`==`
integer inequality	`<>`	`/=`
strings	not a list of characters use `explode`	a list of Characters
renaming parameters	*lst* `as` `(x::xs)`	*lst*`@(x:xs)`
functional redefinition	permitted	not permitted
pattern-directed invocation	yes, with `|`	yes
parameter passing	call-by-value, strict, applicative-order evaluation	call-by-need, non-strict, normal-order evaluation
functional composition	`o`	`.`
infix to prefix	`(op` *operator*`)`	`(`*operator*`)`
sections	not supported	supported, use `(`*operator*`)`
prefix to infix		`` `operator` ``
user-defined functions	introduced with `fun` can be defined at the prompt or in a script	must be defined in a script
anonymous functions	`(fn` *tuple* `=>` *body*`)`	`(\` *tuple* `->` *body*`)`
curried form	omit parentheses, commas	omit parentheses, commas
curried	partially	fully
type declaration	`:`	`::`
type definition	`type`	`type`
data type definition	`datatype`	`data`
type variables	prefaced with `'` written before data type name	not prefaced with `'` written after data type name
function type	optional, but if used, embedded within function definition	optional, but if used, precedes function definition
type inference/checking	Hindley-Milner	Hindley-Milner
function overloading	not supported	supported through qualified types and type classes
ADTs	module system (`structures`, `signatures`, and `functors`)	class system

Outline

- 9.1 Chapter Objectives
- 9.2 Aggregate Data Types
- 9.3 Inductive Data Types
- 9.4 Variant Records
- 9.5 Abstract Syntax
- 9.6 Abstract-Syntax Tree for Camille
- 9.7 Data Abstraction
- 9.8 Case Study: Environments
- 9.9 ML and Haskell: Summaries, Comparison, Applications, and Analysis
- **9.10 Thematic Takeaways**

9.10 Thematic Takeaways (1 of 2)

- A goal of a *type system* is to support *data abstraction* and, in particular, the definition of *abstract data types* that have the properties and behavior of primitive types.
- An inductive *variant record* data type—a union of records—is particular useful for representing an *abstract-syntax tree* of a computer program.
- Data types and the functions that manipulate them are natural reflections of each other.
- The conception and use of an *abstract data type* data structure are distributed among an implementation-neutral *interface*, an *implementation* containing function *definitions*, and an *application* containing invocations to functions in the implementation.

9.10 Thematic Takeaways (2 of 2)

- The underlying representation/implementation of an abstract data type can change without breaking the application code as long as the contractual signature of each function declaration in the interface remains unchanged.
- In this way, the implementation *hidden* from the application.
- A variety of representation strategies for data structures are possible, including list, abstract syntax, and closure representations.
- Well-defined data structures as abstract data types are an essential ingredient in the implementation of a programming language (e.g., interpreters and compilers).
- A programming language with an expressive type system is indispensable for the construction of efficacious and efficient data structures.