1989

HOPS: A Hierarchical/object-oriented programming environment system

Ryuichiro Kodama

Rochester Institute of Technology
School of Computer Science

HOPS:
A Hierarchical/Object-Oriented Programming Environment System

by
Ryuichiro Kodama

A thesis, submitted to
The Faculty of
the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

11/21/89
Professor James E. Heliotis Date

21 Nov 89
Professor Peter G. Anderson Date

11/21/89
Professor John A. Biles Date

Acknowledgements

I would like to thank Hitachi, Ltd. for its scholarship during the study in the MS program at Rochester Institute of Technology. Being a foreign student, I greatly appreciate Janene Oettel's help in pointing out the grammatical mistakes in English. Finally, I would like to thank my wife, Kazuko, for her support. This paper could not have been completed without her cheerful encouragement.

About the Reproduction of This Thesis

Title of Thesis:

HOPS: A Hierarchical / Object-Oriented Programming Environment System

I, Ryuichiro Kodama, hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole. Any reproduction will not be for commercial use of profit.

Signature:_____ Date: 11/21/89

Abstract

The concept of object-oriented programming is becoming an important paradigm in which programmers represent and solve problems. HOPS (Hierarchical / Object-oriented Programming Environment System), developed in this thesis project, employs this concept to represent and manipulate the resources of a programming environment. All the resource objects (such as files, directories, simple integer data and so on) are hierarchically arranged for easy access and management, as the user moves through a tree structure just as in the UNIX file system. HOPL (Hierarchical / Object-oriented Programming Language), whose grammar is similar to C language, has been designed so that the expression of resource objects can be directly embedded in a statement, which is translated into intermediate codes and then is executed. The system allows the user to store the intermediate codes as methods and to inherit methods from objects which already contain methods. HOPS, functioning now under the UNIX environment, can share the UNIX directories and files as its resources, so that it can be an object-oriented front-end processor for UNIX shell commands. An experimental screen editor, HOPE (Hierarchical / Object-oriented Programming Editor), has also been developed. It is written partly in HOPL to facilitate customization of some editing commands.

Computing Review Subject Codes

The following are the primary and secondary classification codes for this thesis, according to the ACM's *Computing Reviews*.

The primary classification code:

D 3.3 Language Constructs

The secondary classification codes:

F 3.3 Studies of Program Constructs

E 2 DATA STORAGE REPRESENTATIONS

Table of Contents

1. Introduction

One of the primary goals of an interactive programming environment is to enhance the productivity of software programs. Easy manipulation of system resources (such as files, disks, and other input/output devices) is essential for productivity. The UNIX† operating system [7], based on a hierarchical file system, is a successful interactive programming environment which makes it easy and understandable to use system resources by representing them as files. That is, UNIX enables us to abstract resources such as I/O devices by read/write operations. This project develops an interactive programming environment which introduces object-oriented methods into the representation of resources, and arranges every object hierarchically. For instance, even a directory, which itself contains objects, is regarded as an object that can accept messages to manipulate its embedded objects. Because object-oriented methods provide more precise abstraction of resources than just read/write abstraction, the resources, acquiring the ability to standardize their interfaces, gain greater connectivity. Most importantly, the hierarchical arrangement of objects makes it easy to access and manage objects.

The environment system developed in this project is called HOPS (Hierarchical/Object-oriented Programming Environment System). HOPS is a single-user, single-task system. The system includes two components: (1) an interpreter of the newly-designed object-oriented language, HOPL (Hierarchical/Object-oriented Programming Language), which handles objects by internally interpreting messages sent to them, and (2) a screen editor, HOPE (Hierarchical/Object-oriented Programming Editor), whose instructions are written in HOPL so that the user can customize some of its functions. The editor is also an example of an application program written in HOPL; therefore, an understanding of the editor will help the reader understand HOPL language.

---

† Unix is a trademark of AT&T Bell Laboratories.

1.1. More Precise Abstraction through HOPS

In the UNIX operating system, all peripheral devices are represented as files in the hierarchical file system [7]. Most of the device-dependent issues are concealed inside the device drivers, so that application programs can communicate with those devices through basically only read and write operations. Even the communications between processes are done through a pseudo file called a 'pipe' or through standard input and output. While this standardization of resource connections by a file provides great flexibility, there are many resources that cannot be abstracted by only read and write operations. A directory, implemented as a file in UNIX, holds some files or some other directories. The directory obviously needs some operations other than read and write, such as a lookup operation, which could test if a file specified as an argument exists in the directory or not.

As an alternative to file abstraction, this paper advocates hierarchically-arranged objects which represent all the resources in an interactive programming environment. By regarding the resources as objects which can accept messages, it is possible to represent many different types of resources as objects. A directory object can accept the lookup operation as a message. A directory object can accept a message X, then execute it by sending a message such as "evaluate" to the method object X that the directory associates with X. Thus, customized commands can reside on a directory as objects. An interpreter object can accept a message like "change the working directory." And an editor object can even accept all the messages to edit a text, so that the users can create customized editor commands using those messages.

This system is based on object-oriented methods [3], which standardize all executions as messages sent to objects, so that we can take advantage of dynamic high connectivity between objects. This feature is also referred to as an advantage of a typeless language. Suppose that an object is attached as an argument to the message EAT, and receives only the message GRILL in the process of executing the EAT method

(implementation program of EAT). In this case, you can invoke EAT with any argument which can receive GRILL; the argument might be an object PORK, CHICKEN, or BEEF. One day you create an object LAMB and want to invoke EAT with LAMB. At that time, you don't need to create a specialized version of the EAT message, such as EAT-LAMB. All you need to do is just add a method for the GRILL message to the object LAMB, then you can send a message, EAT LAMB. In this way, the connectivity of EAT is enhanced.

In HOPS, every named object is located in at least one directory and all directories are arranged hierarchically, so that the objects related to a project can be collected in one directory for administrative convenience and they can be searched in a hierarchical way. Basically, there are no globally-named objects other than a root directory and a current working directory. The root directory represents the HOPL interpreter; it holds the variable objects used in the interpreter, such as a previous input statement. The current working directory is the object to which most typed-in messages are sent. Every command in HOPS starts with a message symbol which may be followed by some arguments, if any. You can change that directory by sending the interpreter a message to change the current working directory.

1.2. Previous Work

The Smalltalk-80† programming environment [3] is a pioneer system which not only provides but is also described in an object-oriented language. This system has been developed through several versions: Smalltalk-72, Smalltalk-76, and finally Smalltalk-80 in 1980. In the Smalltalk-80 system, a window called a browser enables the users to find a specific class and message [4]. Both classes and messages are categorized, so that the users can find the code of a specific method by selecting (1) a class category, (2) a class in that categories, (3) a message category in that class, and, finally, (4) a message in that category.

---

† Smalltalk-80 is a trademark of ParcPlace Systems.

One difference between the Smalltalk-80 system and HOPS is the display device; HOPS doesn't rely on anything but a TTY-type device, while the Smalltalk-80 system is dependent on a bitmapped display. In a sense, HOPS provides more primitive functions which would be basic operations for even using the bitmapped display. The Little Smalltalk system [2] largely supports the specification of the Smalltalk-80 under UNIX. The system, written in C language for portability, shares one of its purposes with HOPS in the sense that it pursues a TTY-version of a object-oriented language. But when you look at the scoping rule of object variables in those two systems, Little Smalltalk appears not to have such a structural scope as the hierarchical arrangement of variables that HOPS provides.

Directories play a significant role in hierarchically arranging the objects in HOPS. Objects are connected to a dictionary in which instance variables and user-defined methods are looked up. There is no direct way to access instance variables; a method bound to the symbol '.' (period), receiving an instance variable name as an argument, returns the value of an instance variable. Among currently investigated object-oriented languages, *Self* [17] employs a similar structure for an object's dictionary, whose slots contain both variables and procedures. There is another concept that Self and HOPS share: elimination of a class object. While *Self* gets rid of class objects by exploiting prototype-based objects (substituting instantiation by cloning or copying an object), a method dictionary works as if it were a class object in HOPS. For instance, if a message 'new' is sent to a method dictionary, the dictionary object creates an instance that has operations in the method dictionary. With either method, this elimination reduces complexities (such as those in metaclass connections) of the object-oriented system.

For a compiled and strongly typed object-oriented language, there is C++ [15] developed at AT&T. The objects in that language are scoped based on the scoping rule for C language variables [6]; a variable is only considered to be only either global among some

functions, local to a function, or local to an object. The HOPL is interactively interpreted as well as translatable into intermediate codes.

UNIX provides a lot of useful built-in commands that can manipulate files or their contents. A new command can be built by combining those built-in commands in what is called a shell script. On the other hand, there are a lot of library functions which directly access the structure of a file or allow the quick manipulation of system resources. Those functions help build up an application program written in C language. Thus, UNIX has two basic interfaces: shell script commands and C library functions. For example, there is both a command called 'rm' and a C language library function called 'unlink' which are used to delete a file. In HOPL those two interfaces are standardized in one format; HOPL just sends a message to a object. The deletion of a file is done by sending a message like "delete the file xyz" to the directory containing xyz.

In addition to the HOPL interpreter, another system component in HOPS is the screen editor, HOPE, which can be customized by modifying HOPE itself, because it is written in HOPL. Emacs [13] has a similar feature in that it is written in its own language called Mlisp (Mock lisp). The users can create a customized editing environment by changing some functions in Emacs. For example, it can simulate an environment similar to the UNIX editor called *'vi'*. The users can also add a new function to the Emacs and save it as a library function. While Emacs takes advantage of a lisp-like programming style, such as the dynamic scope of variables, HOPE is built up based on object-oriented methods; it consists of several chief objects and executes edit commands by sending messages to those objects.

1.3. Theoretical and Conceptual Development

1.3.1. Mechanism of HOPS Inheritance

In the Smalltalk-80 system, when the users want to run a simple demo program, they are required first to create a class, then to instantiate that class and send a message to it.

While this indirect procedure preserves programs as assets of the system, it implies that there may be a more direct, quicker procedure to run a prototype program. In HOPS, the class is replaced by a simple dictionary. The object in HOPS is categorized into two types: a *simple object* and a *compound object.* A simple object has only one dictionary, so that it is used for representing a primitive object which does not inherit any properties from other objects, except from a public dictionary. The public dictionary includes methods (such as a logical-or operator "||" and a logical-and operator "&&") which are commonly used by any object. "Compound object" is another name for "directory object", which is connected to a directory-method dictionary, a user-customized dictionary, and zero or more inherited dictionaries. When the users see and modify the content of the current directory, the user-customized dictionary is used to look up an instance variable. Therefore, when the users want to run a demo program, all they need to do is just create a new method in a directory and type in that method name. If a message is sent to a directory, the system looks it up first, (1) in the directory-method dictionary, (2) in the user-customized dictionary, (3) in any inherited dictionaries, (4) in the dictionaries for the HOPL interpreter, then finally (5) in the public dictionary described above. There is a method in the directory-method dictionary which allows inheriting of methods from other directories.

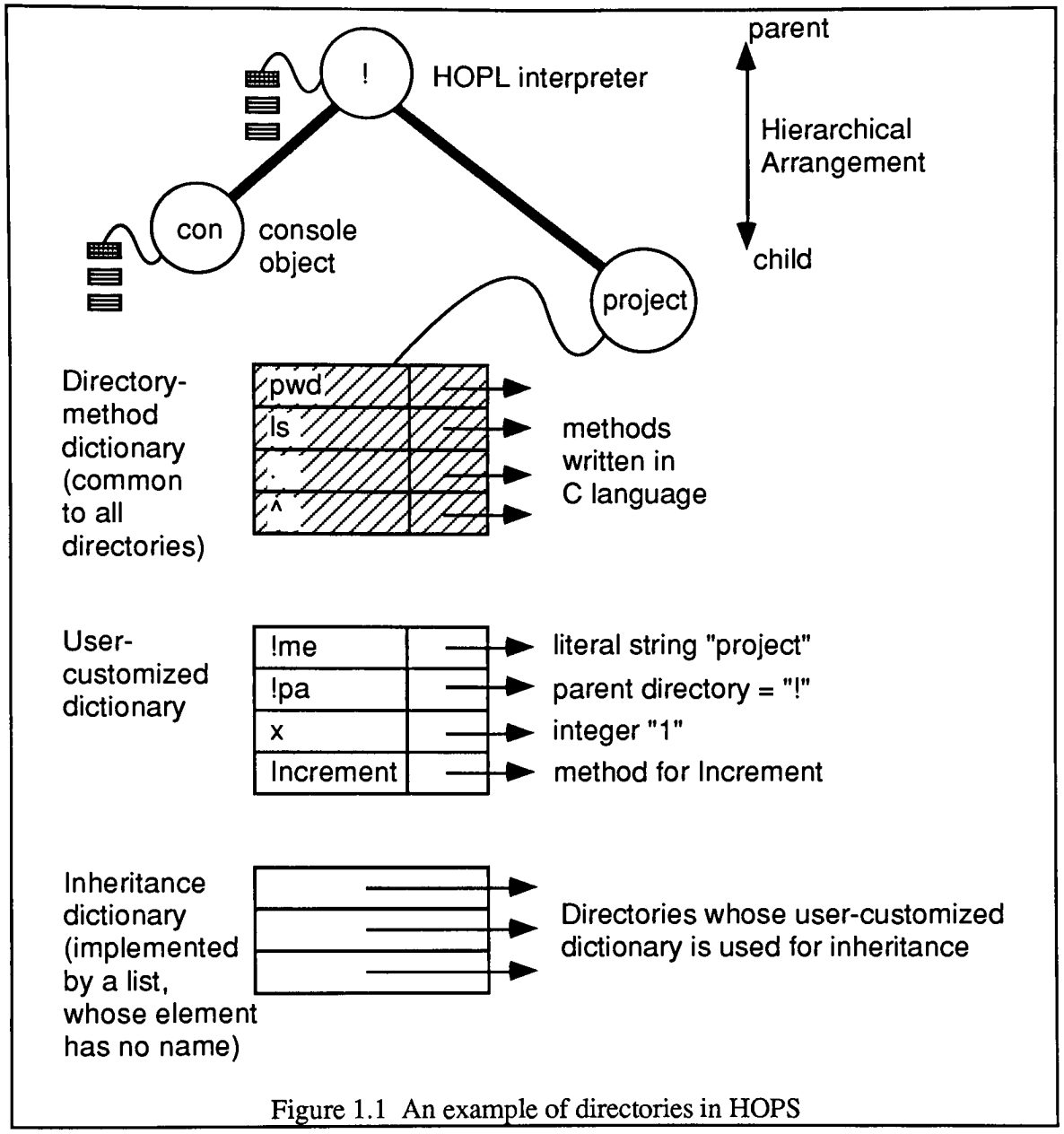


Figure 1.1 An example of directories in HOPS

Figure 1.1 shows an example of directories in HOPS. Every directory shares one directory-method dictionary which consists of tables between a message symbol and a method that is typically written in C language. Each directory has its own user-customized dictionary and inheritance dictionary. The user-customized dictionary contains two important pointers to objects: a pointer to the parent object of the directory and a pointer to the string name of the directory. These two pointers maintain the hierarchical structure in HOPS. The inheritance dictionary is implemented by using a list, which is an array of

inherited directories. Only the user-customized dictionary connected to those directories is used for inheriting methods.

One of the important characteristics in HOPS directories is that there is no difference between an instance variable's location and a method's location. The reference fields in the dictionary point to both instance variables and methods. For example, an instance variable *x* (which holds an integer 1) and a method *Increment* reside on a user-customized dictionary in Figure 1.1. This structure makes it easy to manage bindings for instance variables, because an instance variable can be dynamically obtained by carrying out the same procedure as a method is looked up.

1.3.2. Evaluation Process in the HOPL Interpreter

Most versions of the Smalltalk system interpret an intermediate code (byte code) by incorporating stack operations [3]. This was demonstrated by a simple Smalltalk interpreter written in Smalltalk-76 itself [5]. HOPS applies lisp-like evaluation to the process of interpretation instead of using byte code. Figure 2.1 shows that process, in which the message-sending action is interpreted as an object (method object), then the system sends a special message 'eval' to the method object with no arguments. This design allows us to construct a simple interpreter, thereby eliminating the tedious work necessary to design a byte code.

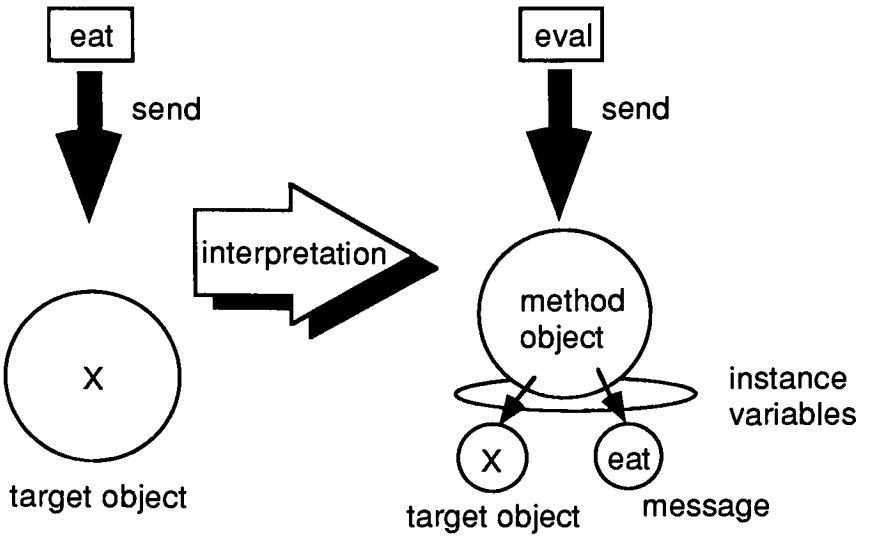


Figure 1.2 Interpretation of sending a message in HOPS

1.3.3. Hierarchical Arrangement of Objects

The hierarchical arrangement of objects plays a significant role in HOPS. In a hierarchical structure, objects can be located in a well-organized way so that more detailed components (objects) can be accessed as the leaf of the tree structure is searched. This structure is also suitable for project management because separated subprojects can be developed in directories independently.

1.3.4. Representation of Objects

In HOPS, the manner of specifying a hierarchical object retains the consistency of object-oriented methods: the execution of a program by sending a message. HOPS represents everything in the system as an object. There are three ways to specify an object: (1) by sending a message '.' with an argument, object name, to the current directory (to get an object embedded in that directory), (2) by sending a message '^' to the current directory (to get the object holding that directory), or (3) by sending a message '.' with an argument, object name, to the root directory (to get an object embedded in that directory). Because only the current directory and the root directory can be globally accessed, the users have to start by sending a message to these two directories. For convenience, the current directory

name does not have to be typed in when a message is sent to it; every statement in HOPL starts with a message symbol which is sent to the current directory (if the directory doesn't have such a message, then it will be sent to the root directory, HOPL interpreter), unless the root directory is specified first in the statement. Instead of typing the statement interactively, if it is written in a method (a program body invoked by a message) and the root directory is not specified first in the statement, then the first identifier appearing in the statement means a message identifier sent to 'self' in the sense of Smalltalk-80 terminology. If such a message symbol cannot be found in the message dictionary of 'self', a special object called ILLEGAL is returned after evaluation process. ILLEGAL is used in case an error condition is satisfied: for example, the case in which a specified message is not found. ILLEGAL responds ILLEGAL to any kind of message except the 'print' message which will print out the name "ILLEGAL" to standard output. ILLEGAL must not be bound to any variables; if one tries to assign it to an instance variable, the assigned value is automatically changed to a system constant object, NIL, which means false in conditional expressions.

1.3.5. Continuity of Conventional Systems and HOPS

The syntactic sugar described above gives the users a sense of continuity between a non-object-oriented, conventional system and HOPS. A conventional system allows for a set of globally-accessible commands, which consist of first, a command name, and then some arguments, if any. When a file is deleted, the command 'rm xyz' is typed in, where 'rm' is a command name and xyz is the argument that indicates the file to be deleted. Behind this system is the idea that the system works as a "servant" accepting commands. HOPS does not change this idea except that it employs as many different "servants" (which can understand different sets of commands) as possible. The "servant" in HOPS is a directory object, in which (in addition to the standard set of commands for the directory manipulation) the users can register new commands or inherit some commands from the

other directories. In this way, as the users move through the hierarchy, HOPS provides a customized command set that is connected to the current working directory.

1.3.6. TTY-based System

Generally, an advanced programming environment requires users to spend a lot of time at the keyboard typing in text and debugging. As long as the chief task in programming is to assign a symbol to an idea, the keyboard will not be eliminated. Even when a system employs menus and icons a great deal, programmers still have to type in a newly-created name related to the concept that they are trying to model. Use of a pointing device, or mouse, which is inevitable with a bitmapped display, interrupts that typing work; the programmers have to keep switching their hands between keyboards and mice. So for this reason, and to make the system as portable as possible, it was desirable to use a TTY rather than bitmapped display for the basic components of the system. However, if one wanted to make the user interface friendly to novice programmers, it is still possible to construct a bitmap-based system on HOPS, because of its primitive functionality.

1.4. HOPS Glossary

The following list shows HOPS terminology in this paper, most of which is from Goldberg [3]. The purpose of this section is to clarify the difference between Smalltalk-80 terms and HOPS terms. Most importantly, HOPS implements a "class" in a different way.

object — A component of the HOPS system represented by some private memory and a set of operations.

message — A request for an object to carry out one of its operations.

class — An object that describes the implementation of a set of similar objects. This is represented by a simple dictionary in HOPS.

instance — One of the objects described by a class.

instance variable	A variable available to a single object for the entire lifetime of the object; instance variables can be named.
method	A description of how to perform one of object's operations. While a message is an external representation of the request to an object, a method describes an internal specification about how to carry out that request.
receiver	The object to which a message is sent.
message symbol	The name of the type of operation a message requests of its receiver.
message argument	An object that specifies additional information for an operation.
block	A description of a deferred sequence of actions.
block argument	A parameter that must be supplied when certain blocks are evaluated.
simple object	An object that does not inherit any properties from other objects and has only one method-dictionary. This is original terminology in HOPS.
compound object	Another name of a directory in HOPS. In contrast to a simple object, this object can inherits properties from other objects; therefore it can hold several method-dictionaries in it. This is also original terminology in HOPS.

2. Functional Specification

2.1. Object Management

Because everything in HOPS is represented as an object, the object structure plays a significant role in determining the system's performance and basic properties, such as primitive methods. The basic idea of an object is that it is a container of pointers to objects. In HOPS, the requirements for an object is that the number of pointers must be expandable (as far as the system's resources allow) and reducible at any time, and that the object be a sequence of those pointers or simple integer values. A reference counter method is employed to recover free space.

2.2. HOPL (Hierarchical / Object-Oriented Programming Language)

Below is the grammar of HOPL. Here non-terminal symbols are written in un-bold fonts, and terminal symbols are in bold face. Alternative categories are listed on separate lines; in a few cases, a long set of narrow alternatives is presented on one line, marked by the phrase "one of". An optional terminal or nonterminal symbol carries the subscript "*opt*". This syntax notation comes from Kernighan [6].

```
compilation-unit:
        method-definition-unit
        interactive-unit

method-definition-unit:
        method-structure method-definition-unit
        method-structure

method-definition:
        instance-variable := method-structure

instance-variable:
        . identifier
        . character-string-object

method-structure:
        local-declaration-list opt { local-declaration-list opt statement-list opt }

interactive-unit:
        local-declaration-list opt statement-list

local-declaration-list:
        local-declaration local-declaration-list
        local-declaration

local-declaration:
        var identifier-list ;

identifier-list:
        identifier , identifier-list
        identifier
```

There are two compilation units: a method definition unit and a interactive unit. The method definition unit contains a list of method definitions, which define how to respond to a message. This unit is normally stored in a permanent file, and is compiled as needed. The current working directory gets to accommodate methods with names after the compilation. Because in HOPS, instance variables and methods share their location (in a directory), the method-definition has the form of an instance variable assignment.

The interactive unit is a collection of HOPL statements. This unit specifies the format of commands which are typed in from a TTY. This unit can be saved in a permanent file, which is compiled and executed just like a *command batch file*. The production rule says local variables can be declared for those commands.

```
statement:
        expression-statement
        if-statement
        iteration-statement
        jump-statement
        compound-statement

compound-statement:
        { statement-list opt }

statement-list:
        statement statement-list
        statement

expression-statement:
        expression opt ;

if-statement:
        if ( expression ) statement
        if ( expression ) statement else statement

iteration-statement:
        while ( expression ) statement
        for ( expression opt ; expression opt ; expression opt ) statement

jump-statement:
        return term-expression opt ;
```

A statement is similar to a C language statement except that a switch-statement is not included in the HOPL statement and that a compound-statement in HOPL does not contain a local variable declaration. If an expression is not NIL (constant-object; described later), then it is regarded as TRUE in a conditional expression of if-statement and iteration-statement. The solution of the "dangling-else" problem is the same as C language's: associating the *else* with the closest previous *else*-less *if*.

```
expression:
        instance-variable = expression
        local-variable = expression
        instance-variable := expression
        local-variable := expression
        message-expression

local-variable:
        $ identifier
        $ positive-integer-object
        identifier
```

Assignment statements change the value of instance variables or local variables. There are two assignment operators:"=" called simply an assignment operator and ":=" called a quoted assignment operator. If the quoted assignment operator is used, the right-hand-side expression, remaining not evaluated, is assigned to the variable which is specified by the left-hand-side expression.

To create an instance variable, these assignment operators are used normally. Even if the specified instance variable does not exist, the assignment operators create that variable.

There are three ways to reference a local variable. In the most formal way, the dollar sign '$' is prefixed before the variable name identifier. This gives the safest grammar. Another way is the dollar sign plus positive number, which indexes the local variable. The easiest way to represent a local variable is to use just the variable name identifier without the dollar sign. The HOPL compiler tries to match the identifier to some local variable. If it fails to do so, an error message will be printed.

```
message-expression:
        term-expression  cascaded-message opt
        cascaded-message

cascaded-message:
        special-message : opt  cascaded-message
        keyword-message : cascaded-message
        special-message
        keyword-message

keyword-message:
        identifier  multiple-arguments opt

special-message:
        special-identifier  object-expression opt
        special-identifier  identifier opt
```

A special identifier is a sequence of the following characters:

% & * + – / @ ~ | < > ? \ ` =

but the identifier consisting of only the equal sign, '=', is not a special identifier (the equal sign is used for an assignment statement).

Using cascaded messages, several messages can be typed in one statement. For example,

1+2 factorial: /(2+1 xor 3)

is transformed into the following parsing tree (Figure 2.1).

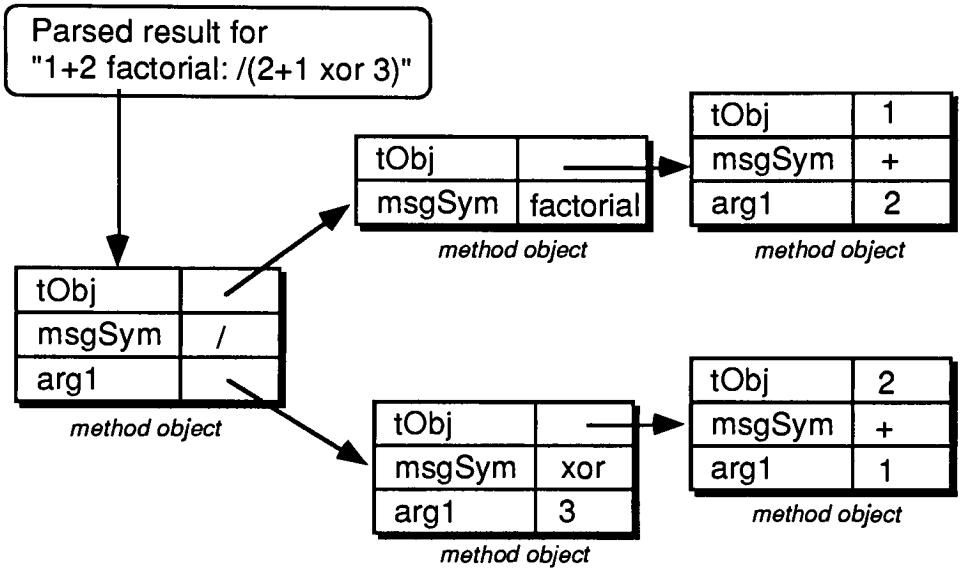


Figure 2.1 Parsed result of 1+2 factorial: /(2+1 xor 3)

A parsing tree, consisting of the method objects, receives the "eval" message from the HOPL interpreter, so that the message propagates through the leaves in the tree and finally returns the result to the caller.

```
multiple-arguments:
        single-argument , multiple-arguments
        single-argument

single-argument:
        term-expression

term-expression:
        object-expression cascaded-special-message opt
        cascaded-special-message

cascaded-special-message:
        special-message : opt  cascaded-special-message
        special-message
```

When a keyword message has several arguments, some special messages can be included in each argument. For example, the keyword message,

```
1 send 1+2+3,1–3
```

has two arguments (1+2)+3 and 1–3.

```
object-expression:
        constant-object
        ( expression )
        block-expression
        local-variable
        hierarchical-object
        identifier

hierarchical-object:
        instance-variable hierarchical-object
        ^ hierarchical-object
        instance-variable
        ^

block-expression:
        [ local-declaration-list opt  statement-list opt ]
```

The hierarchical-object enables us to reference any instance variable by moving through the hierarchical structure. The following expression specifies an instance variable ins2 of an object bound to an instance variable ins1 of a parent object of the parent object of the current working directory.

```
^^.ins1.ins2
```

The block-expression is a block in Smalltalk-80: a description of a deferred sequence of actions. It will accept a message "value" with arguments, if any. Those arguments should be declared inside [ ].

```
constant-object:
        number-object
        character-string-object
        character-object
        ! hierarchical-object opt
        system-constant-object

number-object:
        - number-object
        positive-integer-object
        positive-float-object

system-constant-object: one of
        TRUE  FALSE  NIL  SELF  SUPER
```

The character-string-object and character-object are represented in the same way as in C language; the character-string-object is enclosed by double quotes and the character-object by single quotes. The exclamation mark "!" stands for the root directory, or HOPL interpreter. It is possible to specify a directory object from the root directory. The system-constant-object FALSE is another name for NIL; those internal representations are the same. SELF and SUPER are pseudo variables in the sense of Smalltalk-80 terminology.

Miscellaneous Lexical Conventions

- A comment begins with /* and terminates with */, or begins with // and terminates with new-line character or end of file.

- An identifier is a sequence of letters and digits. The first character must be a letter; the underscore counts as a letter.

- A semicolon ';' is automatically added to the end of a statement which is interactively typed in. That is, the user does not have to terminate an input statement with a semicolon.

The HOPL interpreter interactively accepts only expression-statement. A statement executed interactively ends with a new-line character which can be an alternative to the character ';'. All the interpreter does is read an object from a TTY, evaluate it, and write the result of the evaluation to a TTY. The following are examples of constant-object, where the sentence following "//" is a comment.

```
152                          // integer-object
-234.34                      // float-object with an unary operator "-"
"hello world\n"              //character-string-object
                             //(same as the one in C language [6])
'z'                          //character-object
                             //(same as the one in C language [6])
```

The following example shows how to work on a customized object (directory), where "HOSP> " is a system prompt message.

```
HOPS> l
project_A  project_B
                        //The 'l' message will be sent to the current directory.
                        //There are two objects (both are directories
                        //in this case).

HOPS> cd .project_A
                    //Cd will be interpreted by the root directory, because
                    //the current directory does not accept that message.
                    //The dot '.' is sent to the current directory with
                    //the argument, project_A, and then the directory object
                    //bound to the name 'project_A' will be returned.
                    //That directory is passed as an argument for the message
                    //'cd'. Finally, the interpreter (root directory) changes
                    //the current directory to the project_A directory.
HOPS> l
x     Increment     //The 'l' message will be sent to the current
                    //directory "project_A".
                    //Suppose that 'x' is an integer and 'Increment' is bound
                    //to a method.
HOPS> .x = 1+2+3
6                   //X is bound to 6. The plus '+' is sent to 1 and
                    //the result of 1+2.
HOPS> .Increment type
                        //The method Increment receives the message
                        //"type", which displays the method content.
var i;
{
      .x = .x + $i;
}
HOPS> Increment 3
9                   //The message Increment is sent to the current directory
```

```
                    //with the argument 3. Finally, X is bound to 9.
HOPS> .Increment vi
                    //The message vi (the UNIX screen editor) is sent to
                    //'.Increment'. The method for "vi" invokes the UNIX vi
                    //if there is a documentation file attached to the '.Increment'.
                    //You can edit a method in this way.
```

2.3. HOPE (Hierarchical / Object-Oriented Programming Editor)

There are two main objects that support the screen editor HOPE: a line display object called "curses" and a scratch file object called "ed". Curses organizes text lines on a TTY display and manages a current cursor position in which input characters are inserted. In order to enhance its portability among many kinds of TTY displays, curses employs UNIX library curses as the name literally describes. It includes some information about how many lines are displayed and where each line starts or ends in a screen, so that it can prevent the cursor from moving around the screen area in which no lines are displayed.

Ed is a simple line editor which can be used without the line display object. It can retrieve one line specified by a line number from a text file and edit the line. It can also move, copy, and delete some lines specified by two line numbers. The scratch file is a temporary file which is used for keeping up all the edited lines of a text file according to Kernighan [8].

HOPE works by sending messages to these two objects, curses and ed. The user can edit a text file by invoking two kinds of commands: keyboard-mapped commands and named commands. When you open HOPE, every key typed in first is a command as in UNIX *vi* editor. For example, if you type in the character 'i', then you can insert characters into the location pointed to by the cursor. By typing in the named command following ':', you can send messages to HOPE. The customization of the named commands is done by adding methods onto the directory for HOPE or by inheriting methods into the HOPE directory.

Figure 2.2 shows the structure of HOPE as described above. The file with which HOPE deals is also an object representing a UNIX ASCII file. The user can handle a UNIX ASCII file as an object that can accept messages such as concatenate, append a string, show its contents, and so on.

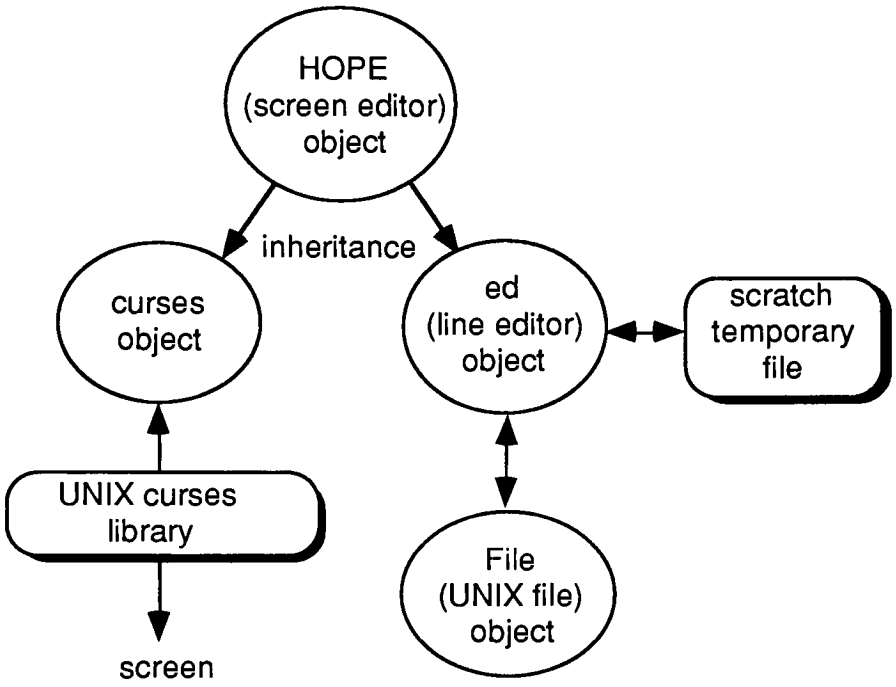


Figure 2.2 Structure of HOPE
(Hierarchical/Object-oriented Programming Editor)

3. Program Description

3.1. Basic Structure of an Object

Because every resource in HOPS is an object, it is necessary to identify an object somehow. The identification number for an object consists of 16 bits, whose positive representation (15 bits with the most significant bit being zero) indicates the index of an array of basic object blocks. The length of the object block is 16 bytes and it contains some information about an object, as described later. The negative representation is used for the identification of special objects, which do not need to be allocated space when they are created; they include system constant objects such as NIL and TRUE.

The structure of an object can be explained following three steps, each of which has a considerably independent function.

In order to conveniently fulfill the need to allocate new space in the process of object creation, all the positive identification numbers are initially queued in a free block stack. When new space is needed for object creation, a number popped from this stack is used, and when destruction of objects is needed, the identification numbers of destroyed objects are pushed to this stack.

At that step, every block is considered to be transparent. That is, there is no difference in information among the blocks dequeued or enqueued. In the next step, an object array whose volume can be changed as needed is implemented using these transparent blocks. Here the blocks are categorized into two types: *header blocks* and *node blocks*. Figure 3.1 shows how the volume-changeable array is constructed. One half of a *header block* (8 bytes) is dedicated to pointing to up to four objects, since each object is represented as 16 bits (2 bytes). If the volume of an array becomes greater than four object elements, *node blocks* are added under the *header block*. The *node block* points to up to eight object elements, which means that one *header block* and four *node blocks* can hold a

maximum of 32 (4 x 8) elements. If the number of elements is greater than 32, then the addition of *node blocks* to the tree-like structure can be continued in a similar manner until the free block stack is exhausted. This is how the practically infinite length of an instance object is attained. This gives HOPS another advantage: the average time to access each element is proportional to the logarithm of the element number (based on 2). This access speed is relatively fast, compared with the implementation using sequential lists, whose element access costs the order of the element number.

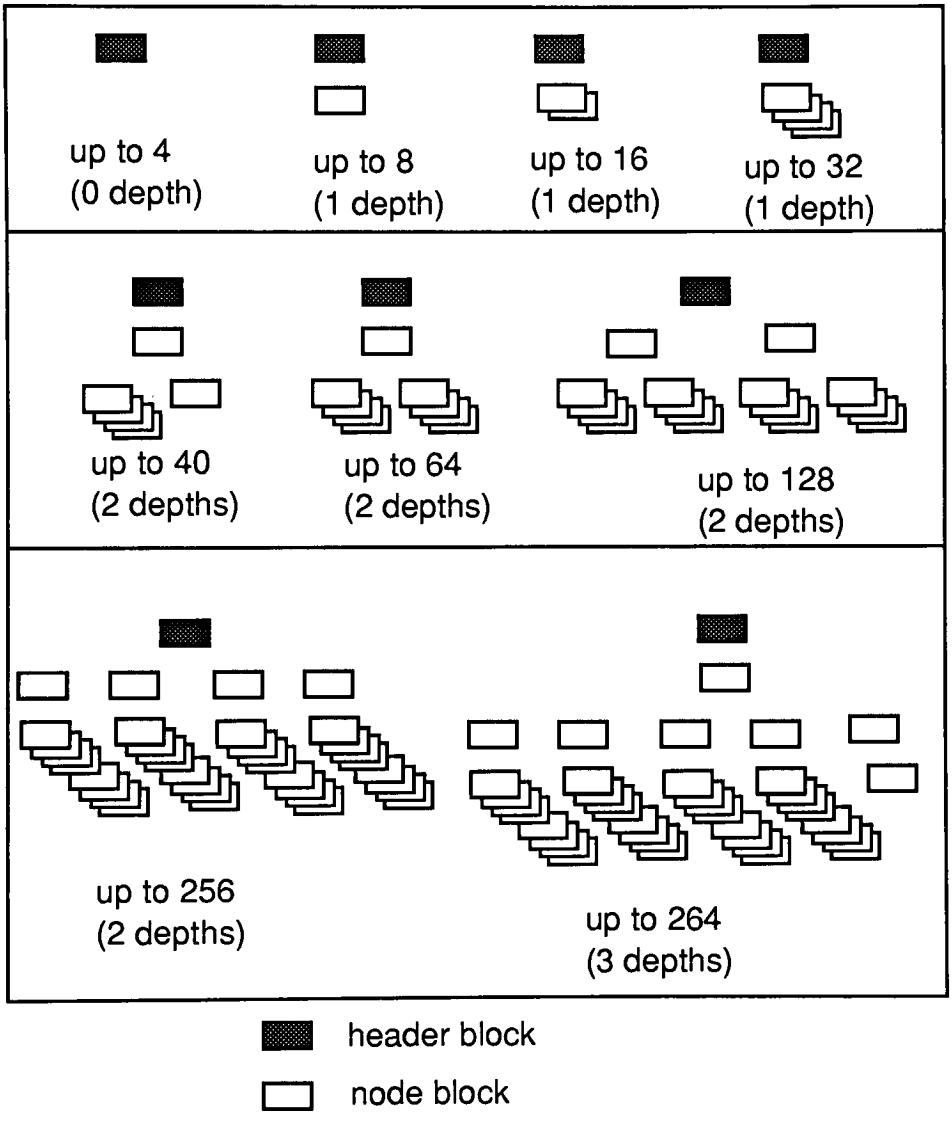


Figure 3.1 The implementation of infinite-length object arrays

By taking advantage of the unbounded size of the tree structure mentioned above, it is possible to represent an object regardless of its length. On the other hand, there is more information to be added in this structure. The upper half of a 16-byte *header block* is used to store the information which the system needs to handle an object, such as the flag which indicates whether the contents of an object are pointers or not.

Figure 3.2 illustrates the structure of a *header block*. The first slot (2-byte field) in this block is called *magic*, which is actually the index of a method dictionary object. If the message is sent to this object, that dictionary object is used to look up the message symbol. The second specifies how many words (16 bits) an object has as its slots. Note that this number uniquely decides the tree structure. For example, if the length is 15, then the tree consists of one *header block* and two *node blocks*. The third is a reference counter,which is described in a later section for reference counts. The fourth is a set of bit flags, four of which are explained here.

(1) GC flag: This flag indicates whether the elements held by a header block are object pointers or not. It is used for reference counts when an object is discarded.

(2) KILL flag: When the object space is collected as garbage, the system checks for this flag to decide whether or not a special message is to be sent to this object before returning it back to the free block stack. This is used, for example, to delete a UNIX file connected to a File object (an object behaving like an ASCII file) which is being discarded.

(3) EVAL flag: The HOPL interpreter evaluates every object by sending a special message "eval". This flag eliminates the frequent search for the "eval" method; if the flag is off, then the object is regarded as a constant throughout the evaluation process, and if the flag is on, the system starts to look for the "eval" method. Most of the objects do not have eval method in the dictionary; their EVAL flags are all off because they are constant throughout the evaluation.

(4) MOUNT flag: If a directory object is marked with this flag on when created, then the file objects (objects behaving like a file containing ASCII characters) created under that directory are automatically linked to the UNIX files. When a directory object is created under that directory, the same named directory is created under the UNIX file system. This is how the project develops programs written in HOPL; a File object containing HOPL source codes is automatically saved in the UNIX file. This mechanism will be discussed later in detail.

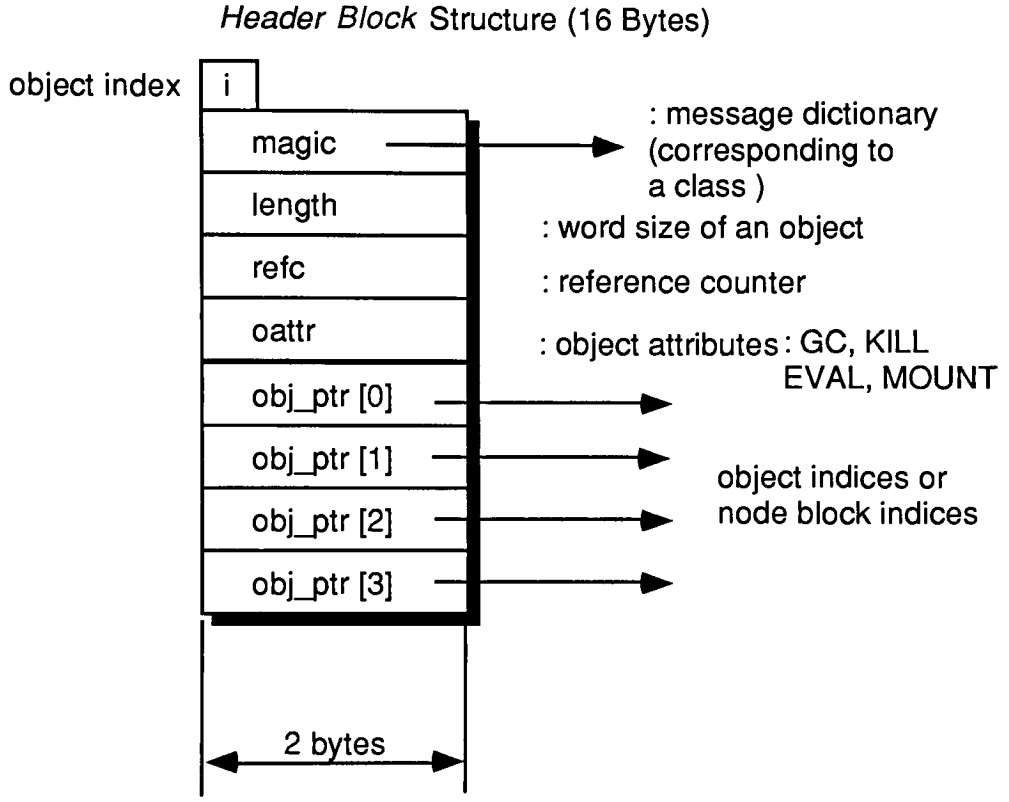
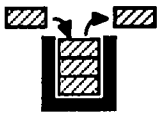


Figure 3.2 A *header block* structure

Table 3.1 summarizes the process of constructing an object. It is interesting to note that the process can be compared to the reference model [16]. Each layer's function described in Table 3.1 is implemented by using the lower layer's *services*. Objects communicate to each other during their lifetime, driving the services provided by the uppermost layer. Protocols between objects could be naturally introduced in this model.

Table 3.1 Reference model in implementing an object

layer #	Function	Implementation	Illustration
0	To supply transparent, regular-size blocks as needed	A stack containing free blocks	
1	To implement a variable length objects	A tree structure consisting of a header block and some node blocks	
2	To represent any kinds of objects, considering the characteristics of each	(positive) block indices for ordinary objects and negative numbers for system reserved objects	**An object**

3.1.1. Basic Operations on an Object

An object in HOPS is a tree structure consisting of blocks as described above. There are several basic operations on this tree structure. The tree structure is invisible; all that is observed from outside is a value's sequence, whose length is infinitely enlarged or reduced. Therefore, two basic operations, insertion and deletion of elements, are provided. It is possible to place an element anywhere in the sequence. Another important operation is the transformation of data between a tree structure and a flat, contiguous memory. Most primitive objects are implemented by C language, in which data is represented in flat memory. In a sense, that transformation is a basic interface between HOPL and C language.

3.1.2. Physical Structure of Some Primitive Objects

All the objects in HOPS are implemented using the structure mentioned above. In this section, let us take a look at the physical structures of some primitive objects. The category name (class name) of the primitive object shown below starts with a capital letter. The user of HOPS is encouraged to make it capital for clarification if he or she adds a new

class at all, although the system does not require it. This convention comes from Smalltalk-80's notation [3].

(1) Integer: It is represented physically in four bytes, which is called 'long integer' in C language terminology. Because object pointer slots of a header block are eight bytes, enough to contain those four bytes, an integer object occupies 16 bytes for just one header block.

(2) Float: It occupies 8 bytes, whose format is the same as 'double' in UNIX C language.

(3) String and Character: It is a sequence of byte characters; if the length is odd, then it is adjusted to even byte length by terminating with the null character. The object length kept in a header block still indicates word size, not character size.

A Character object is represented in negative 16-bit numbers. That is, when it is created, the system does not have to allocate space to hold the character value. The negative index number itself tells us what it represents. The range from -100 to -355 is reserved for a Character object, so that those objects can be converted to ASCII characters by adding 100 to the index number in C language.

(4) C function method (Cmethod): The content of this object consists of a C language function address and a token ID (2-byte integer). The token ID is supposed to identify each message symbol in a simple object. If this object receives a message symbol "eval", then the C function pointed to by the function address ('fnc') would be as follows:

```
(*fnc)(obj, token, env)
OBJ obj, env; short token;
```

Typically, a switch statement inside the function pointed to by 'fnc' switches C language-written methods by using the token ID 'token'. 'Obj' is a target object in the caller's environment (the current target object is a Cmethod object), and 'env' is the caller's

activation record object. This type of object is added to a dictionary in the initialization phase.

(5) Method: This includes three object pointers: a target object, a message symbol, and an argument list. If this method receives a message symbol "eval", first, the target object and each argument are evaluated, then a new activation record is created, and then the message symbol included in the method object is sent to the evaluated target object with the new activation record holding, if any, evaluated arguments. The algorithm will be described in detail later.

(6) Method list: This consists of a list of method objects and a File object (an ASCII file linked to a UNIX file, explained more in the section "Coordination with UNIX"). This is the place where translated codes for statements separated with a semicolon are stored. The File object is used to refer to the documentation of the translated codes. It is attached to a Method list when translation starts, right after the File object receives the message "trans". The user can still see the original source by sending 'type' to the Method list.

(7) Dictionary: This is an even-length list in which even-numbered (2n) slots hold key objects and the corresponding odd-numbered (2n+1) slots hold value objects. When the Dictionary object receives a message "lookup" with an argument KEY, the list is scanned only for even-numbered slots to look up KEY and the corresponding odd-numbered slot's value is returned if it is found. Otherwise, the Dictionary object returns the special object ILLEGAL.

(8) Directory: It consists of a list of key objects and a list of value objects. Both lists have the same size, because each key object corresponds to each value object. Notice that a Directory object is another name for a *compound object*, which is the only instance object that can inherit methods from other objects. In that sense, the key object and the value object are supposed to be the instance variable name (using a String object) and the instance variable value, respectively.

In HOPS, Directory objects are hierarchically arranged. To implement this, special instance variables called "!pa" and "!me" are employed. The variable "!pa" holds a pointer to a parent Directory object, and the variable "!me" holds the name of a self Directory object rather than the self Directory object itself, in order to avoid the cyclical structure. Figure 3.3 illustrates an example of these instance variable connections.

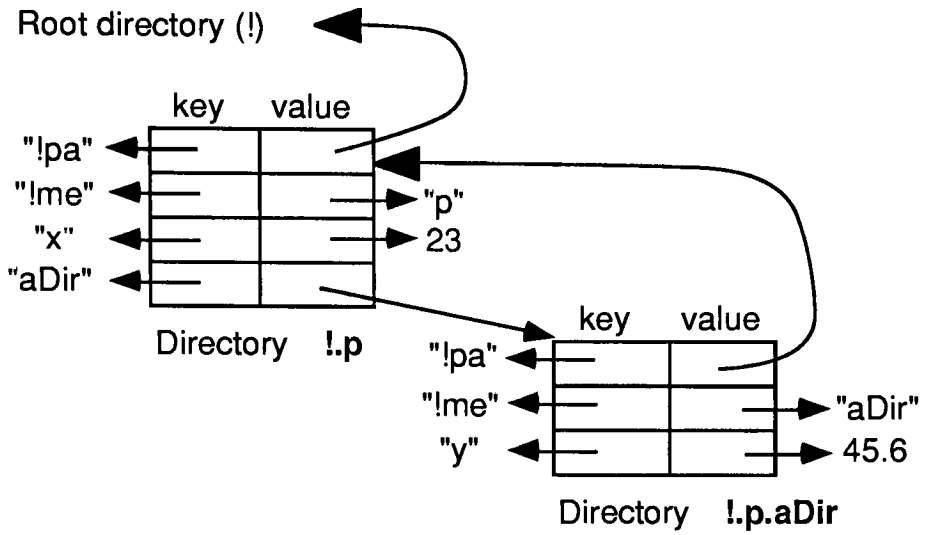


Figure 3.3 An example of directory connections: a directory !.p has an integer instance variable "x" and a directory instance variable "aDir", while a directory !.p.aDir has an float instance variable "y".

A Directory object, namely a compound object, is not always connected to the parent directory; it could be assigned to a local variable in a statement without having a parent directory. The general rule of "!pa" and "!me" connections is that the connections are preserved unless a directory is disconnected from its original parent directory. If a directory is assigned to a local variable right after it is created, then even the variables "!pa" and "!me" will not exist in that directory. There is also a rule about disconnection: if a directory is disconnected from an instance variable and that variable belongs to a directory pointed to by a "!pa" variable of the disconnected directory, then the "!pa" variable of the disconnected directory is erased to NIL. All these rules tell us that there could be an

'orphan' directory whose "!pa" connection is not completed and that the consistency of the hierarchical structure is preserved.

3.1.3. Creation of Objects

Constant objects (such as integer, float, or string) are created by a parser. There are two ways to create other types of objects. One is to send a message "new" to a method dictionary (called *magic*) which is attached to the first slot of a header block in an object being created. HOPS initially creates a special directory "!.dict", under which all the magics for primitive objects are bound to the corresponding class name. For example, the instance variable ".Directory" in "!.dict" holds a method dictionary for a Directory object. So the following statement creates a new directory named "aDir" under the current working directory.

```
.aDir = !.dict.Directory new;
```

The other way to create an object is to send a message "new" to the same-class instance object. The following statement expresses the same idea as that above.

```
.aDir = new;
```

The message "new" is received by the current working directory which is a Directory instance object. This way gives the most concise expression of creating a directory.

Although these two are basic means to create an object, there are other ways to do so, depending on the characteristics of the object. For example, an Interval object is created by sending a message "to" to an Integer object as follows:

```
.a = 1 to 5;     // creates an Interval object which is a range from 1 to 5.
```

3.1.4. Reference Counts

HOPS adopts *reference counts* [11] as memory management algorithms. This method avoids causing the long silence in the middle of execution by another typical

method called *garbage collection.* On the other hand, the disadvantage of this method is that it cannot recover a cyclical structure in which one of the slots in a object, directly or indirectly, points to the self object. Therefore, when designing the basic structure of objects in HOPS, some solutions were considered to avoid a cyclical structure. The "!me" connection of a directory, described in the previous section, is an example of those considerations.

Two basic operations are provided for the reference counts method: IncObj and DecObj. IncObj simply increases by one the 'refc' value located in the header block of an object. DecObj decreases the refc value by one and if the value becomes zero, then the object is regarded as discarded because the value zero means that nothing is connected to this object. Before collecting blocks used by the discarded object as garbage, DecObj also starts to decrease the refc values of objects held in the slots of the discarded object. Thus, all the blocks that the discarded object uses are recursively collected and pushed to the top of the free block stack.

The following three paragraphs show techniques for dealing with the reference counts method.

(1) Reference counts in an assignment statement: Suppose we have the following simple assignment statement.

```
$a = $b;
```

An object is bound to each variable before the assignment. IncObj will operate on the object bound to $b and DecObj will operate on the object bound to $a. Here, IncObj must be called first to allow for the next special statement.

```
$a = $a;
```

If DecObj is first and the object bound to $a is not connected to the others, then it is thrown out before IncObj operates on it.

(2) Objects attained in an activation record: The lifetime of any object is determined by the reference counter, so it should be bound to something in order to have a positive reference count. An activation record, implemented as an Environment object in HOPS, plays the role of holding all the locally-bound objects and any returned objects. If a method returns an object to the caller, the activation record for the method is destroyed and the returned object is bound to the activation record for the caller. In the process of destruction, the objects used only inside the method are discarded because the reference counter goes to zero.

(3) Garbage collection in a parsing phase: Because methods are implemented as objects in HOPS, the parsing program (written in C language) creates many objects which remain unbound to object slots. If there is no error, then those objects are bundled into one method or method list, which is collected as garbage after its execution. But it is difficult to collect those objects as garbage when a syntax error is found. Specifically, HOPS applies a syntax-directed translation scheme [1] to parsing, so each C language function corresponds to a nonterminal in the HOPL syntax. If a control is added to collect garbage, the flow of each function will be so distorted that the system could lose the advantage of the clear flows of a syntax-directed translation scheme.

HOPS solves this problem by collecting in advance all the objects created in a parsing phase. Those objects are bound to a simple List object as its members when they are created. This list object is discarded right before the next parsing phase starts. Thus, all the created objects in a parsing phase can be collected even if an error occurs.

3.1.5. C Language Interface

HOPS enables the system to be expanded by linking a new object written in C language. It is an unfortunate fact that the more purely object-oriented the system is, the more slowly it runs. The reason is that a method dictionary search is fundamentally essential. HOPS provides the programming environment in which everything can be written using HOPS objects. But this coding manner is not always recommended if the message-passing overhead gets intolerable. The user is encouraged to find out the functional border which splits programs in HOPL and programs in C language, considering the speed efficiency.

Appendix A shows an example of an object written in C language for HOPS.

3.2. Lexical Analyzer and Parser

3.2.1. TTY Interface and Command File Interface

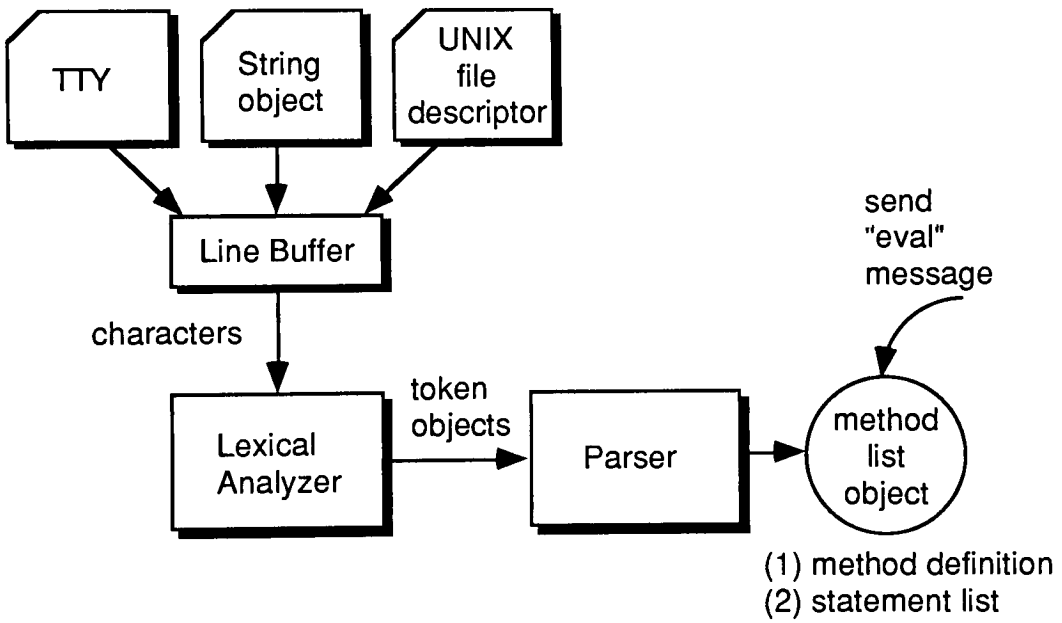


Figure 3.4 A lexical analyzer and parser in HOPS

Figure 3.4 shows a diagram of data flow through the lexical analyzer and parser in HOPS. Input characters are pooled in a line buffer. They are translated into token objects by the lexical analyzer. One of the advantages in using the object-oriented system is that

even tokens are represented as objects. Tokens are usually implemented by two primitives: a token identification number and the related attribute values. For example, the lexical analyzer produces INTEGER (defined as a number somewhere) and its binary value 12, so that using INTEGER, the parser can switch the syntax tree branches correctly and pass the value 12 to the code generation phase. But an integer token in HOPS is simply an Integer object. Its *magic* (see the section 3.1 and Figure 3.2) plays the role of the token identification number, and the value is embedded in it.

HOPS has three input interfaces, as shown in Figure 3.4: TTY, String Object, and UNIX file descriptor. The String object accepts a message "getm" and returns a parsed method object for the character string. Although the TTY interface seems to be implemented by giving a standard input file descriptor to the UNIX file descriptor interface, it is different in several respects from the UNIX file descriptor interface. If a curses mode (in which UNIX curses library is available, and the terminal mode is switched to echo and cbreak) is on, then TTY interface echoes each input character, simulates a backspace, and invokes a command line editor when receiving an escape character. The command line editor enables the user to edit the input line which refers to the history of recently-typed commands, just as in the UNIX *Korn Shell* [9]. Another difference is that TTY interface adds a semicolon at the end of an input line. According to the HOPL syntax, every statement should terminate with a semicolon. The typed-in command has, however, traditionally used a CARRAGE-RETURN or NEW-LINE character as its terminator. The compromise of the two ideas yields the automatic addition of a semicolon.

The output of the parser is always a Method list object or, if only one statement is parsed, a Method object. But there are syntactically two categories to be parsed: a method definition and a statement list. There is ordinarily a File object under a directory, which contains all the definitions of methods in the directory. The parser translates this File object into a method definition. The statement list is typed in from a TTY terminal or is stored in a File object, which is executed as a *batch command file.*

3.2.2 Syntax Analysis

In the initial stage of the syntax design for HOPL, the HOPL grammar was clean in a sense that the parser needed to look ahead at only one token in order to decide the deviation tree. However, in its final form two token look-ahead was used. The structure of the parser program would have been too complicated to allow one token lookahead, due to the problems described below.

(1) Local variable naming problem: Local variables include argument variables and temporarily-defined variables inside a compound statement. They are originally referenced with a dollar sign prefix, '$'. However, after some programs are written in HOPL, adding the '$' sometimes turns out to be cumbersome. For example, consider the following statement in HOPL.

```
for ( $i=0;$i<10;$i=$i+1) $i print;
```

The use of the prefix gives us an easy-to-read format in which the variable 'i' is quickly found to be a local variable. On the other hand, typing the dollar sign on a standard keyboard requires two keys, the '$' key and the shift key. This is not efficient, considering the frequent use of the local variables in a program. Therefore, a statement such as the following is desirable.

```
for (i=0;i<10;i=i+1) i print;
```

Unfortunately there is a problem in this statement. It is against the basic concept in HOPL that every starting identifier is to be regarded as a message symbol sent to the current working directory. Accordingly, the original grammar interprets the "i" as a message symbol to the current working directory in the two expressions, "i+1" and "i print". Finally, the rule added to the grammar is that if the parser meets a special character (such as '+') next to an identifier, then that identifier is interpreted as a local variable. This rule allows the following statement.

```
for (i=0;i<10;i=i+1) $i print;
```

If a keyword message is sent to an object bound to a local variable, the dollar sign must precede the local variable name.

(2) Negative number parsing problem: In the original grammar, the lexical analyzer recognized negative numbers, whose digits must immediately follow a minus sign without any space characters . As a result, the following statements have syntax errors.

```
5     // Sending a message '-' to the current working directory with '5'.
      // (There is a space character between '-' and '5'.)

3-5   // No message symbol between the object '3' and the object '-5'.
3- -5 // '3' receives a message '—' with an argument '5'.
```

The reason for the errors originates in the fact that the grammar adopts binary operators and postfix operators, not prefix operators. What was done to eliminate the above errors was

- to make the minus sign '-' one separated token even though the subsequent character is also a special character,
- to lexically analyze only positive numbers (without '-'),
- to recognize negative numbers if a minus token follows a positive number token in the HOPL's 'expression', and
- to add the following production rules.

```
number-object:
        – number-object
        positive-integer-object
        positive-float-object
```

3.3. Implementation of Control Structure

3.3.1. An Environment Object

An Environment object is an activation record in HOPS. The purpose of this record is principally to give an appropriate scope to the currently running method and to maintain the current working directory. At the same time, its practical role is to point to every locally-used object and to have its reference count be at least one, in order to not permit the reference counts algorithm to collect it as garbage. An Environment object is created just before a message is sent to an object and it is destroyed just after a value is returned to the caller's Environment object. When it is destroyed, the DecObj operator tries to consume all the objects bound to its slots (or instance variables) by decrementing their reference counts. That is how only the objects which the method used locally are released to a free block stack. The following list explains each slot in an Environment object.

(1) Target Object: This slot holds the current working directory or, more exactly, the object which is receiving a message symbol corresponding to the current running method. That is, it is not necessarily a directory and might be a simple object such as an Integer object. If the psuedo variable SELF is evaluated, then this slot is referenced.

(2) Argument List: This slot holds a List object containing the arguments' values for the current running method.

(3) Temporary Variable List: This slot holds a List object containing temporary variables' values for the current running method.

(4) Block Argument List: A Block object is a special method list which can be evaluated only by receiving a message "value". It is constant throughout ordinary evaluations by means of a message "eval". According to its scope rule, a Block object could access the values in the environment where the Block object lexically appears. So, if a Block object is evaluated by accepting a message "value", all the arguments and

temporary variables in that environment must be available together with the arguments for the Block object. This is the reason there is another argument list for a Block object.

(5) Method Holder Object: This slot holds the dictionary or directory in which the current running method was found as opposed to the dictionary directly associated with the object. When a psuedo variable SUPER receives a message, this means it should invoke a method defined in a more 'super' object than the object defining the current running method. If a message is sent to SUPER, an inherited object relative to the current method's dictionary is expected to receive it. The method holder object is used to find that inherited object. We will have more details about this in the section "Inheritance Mechanism".

(6) Caller Env: This points to the caller's environment which is used for returning to the caller.

(7) Callee Env: This points to the callee's environment to keep the reference count of the callee's environment from becoming zero. The top of the callee chain of environments is also pointed to by a C language variable, so that any C function can know the current environment.

3.3.2. The Evaluation Process and Message-Passing

Every statement in HOPS is translated into intermediate codes, such as a Method object or a Method List object and those codes are evaluated. Even the evaluation process can be explained by message-passing: a message "eval" is sent to those translated objects. The method connected to a message "eval" contains codes for how to evaluate the information inside those objects.

Let us investigate the relationship between the evaluation process and message-passing. We need to implement these two somehow in C language. As a useful example to solve this, there are two famous meta-functions adopted in Lisp [10] to evaluate general functions: APPLY and EVAL. The function APPLY plays the role of applying a function

to the function's arguments or, in general, to the lisp environment. Primitive functions such as 'car' and 'cdr' are processed in APPLY, and if the function is a program defined by the user, then APPLY retrieves the code for that function and evaluates it by calling EVAL. EVAL evaluates everything in the system. If the code is a constant number, then EVAL returns it as it is. If the code is a *symbol* (a variable in Lisp which is a special string connected to an associative value), then EVAL returns it as its associative value. If it is a control structure like "if", then EVAL calls a primitive function without evaluating its argument. Finally, if the code is a function, then EVAL recursively calls EVAL for the function's arguments and then calls APPLY with the function *symbol* and those arguments.

Instead of APPLY and EVAL, HOPS employs one meta-function called "SendMsg". Figure 3.5 shows a pseudo program for SendMsg written in C language. SendMsg is involved in the evaluations for local variables, argument variables, SELF, SUPER, and an object receiving a message. Applying a function and Evaluating a function are unified by sending a message "eval" to the corresponding object.

In HOPS, there are five *evaluatable* objects (which means they equip a method corresponding to a message "eval"): a Method object, a Method list object, an Assignment object, a Quote Assignment object, and a Control object. And another special evaluatable object is a Block object, which is constant when receiving an "eval" message. However, it equips a method to evaluate statements inside a Block object. That method is invoked not by "eval", but by "value".

```
extern OBJ interpreter;   /* the root directory*/
extern OBJ evalLit;       /*  String object for "eval*/
"
OBJ SendMsg(obj, msgsym, env)
OBJ obj;                  /* a target object */
OBJ msgsym;               /* a message symbol */
OBJ env;                  /* an Environment object */
{
      OBJ x;
      if ("msgsym indicates variable headers ('$' or '$$'))
            return (get the variable value from "env");
      if ("obj" is a pseudo variable SELF)
            obj = (get the current target object from "env");
      if ("obj" is a pseudo variable SUPER) {
            if ("msgsym" is 'print')
                  print 'SUPER' and return TRUE;
            if ("msgsym" is 'eval')
                  return a token for SUPER;
            // Only directories (compound objects) can have a method which includes
            // 'SUPER'.
            x = (look for a method which is associated with "msgsym"
                  in the inherited directory list of the current target object.
                  If the current running method is included in a member of the inherited
                  directory list, the directory scan starts from the directory
                  next to that including directory );
            if ("x" is the ILLEGAL object, which means the method was not found)
                  x = (look for a method which is associated with "msgsym"
                        in the inherited directory list of the root directory.
                        The way to scan the list is the same as above);
      } else if ("obj" is a directory (a compound object)) {
            x = (look for a method which is associated with "msgsym"
                  in the "obj" directory and its inherited directories);
            if ("x" is the ILLEGAL object, which means the method was not found)
                  x = (look for a method which is associated with "msgsym"
                        in the root directory and its inherited directories);
      } else
            x = (look for a method which is associated with "msgsym"
                  in the magic of the simple object "obj");

      if ("x" is the ILLEGAL object, which means the method was not found)
            x = (look for a method which is associated with "msgsym"
                  in the public method directory);
      if ("x" is evaluatable, which means it holds an "eval" method) {
            if ("x" is a Cmethod object )
                  call the Cmethod evaluation function and return the result;
            else
                  call "SendMsg" with the arguments of "x", the String object
                        for "eval", and "env", and return the result;
      } else return x;
}
```

Figure 3.5 A pseudo program for SendMsg in C language

3.3.3. Instance Variables and Local Variables

The evaluation of instance variables is not explicitly described in SendMsg, as shown in Figure 3.5. This is because a value assigned to an instance variable is retrieved by sending a message '.' to a directory, with its argument being an instance variable name (a String object). So, its evaluation is dealt with through ordinary message-passing.

The HOPL parser translates local variables (including argument variables) into the following.

```
SELF  $positive-offset    // for argument variables
SELF  $negative-offset
                    // for locally-defined variables
SELF  $$positive-offset
                    // for block argument variables
```

These are Method objects in which SELF receives a message "$" or "$$" with an argument. Note that the target object for a variable expression is not checked in SendMsg, i.e., SELF in this translated code has no meaning. The argument must be an Integer object, whose absolute value is interpreted as an offset into a variable list bound in an Environment object.

The HOPL grammar allows the representation of an argument variable in three different ways:

```
$ identifier
$ positive-integer-object
identifier
```

The second rule looks exactly same as the translated argument variable. However, if a local variable turns out to be a block variable according to the scope rule, then it is translated using "$$" as described above.

3.3.4. Method Objects

A Method object represents a basic statement in HOPL; a statement sending a message to an object. Accordingly, it holds a target object, which will receive a message, a

message symbol, and any argument objects. Figure 3.6 shows a pseudo program for the Method object evaluation.

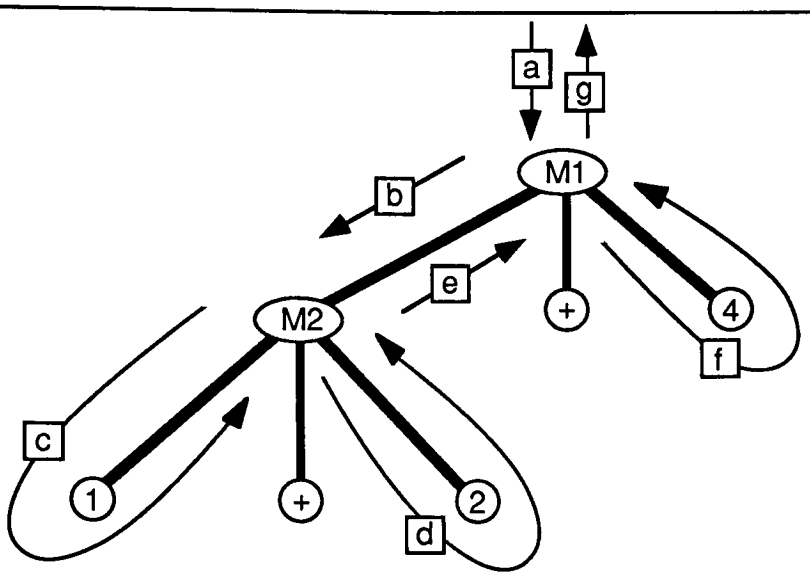
```
MethodEvaluation(mtd, env)
OBJ mtd; // Method object
OBJ env; // the caller's Environment object (activation record)
{
        OBJ tobj, msgsym, newEnv;
        tobj = (get the target object of the Method object "mtd");
        if ("tobj" is the pseudo object 'SELF')
                tobj = (get the current target object from "env");
        if ("tobj" is evaluatable)
                tobj = (the result from sending "eval" to "tobj");
        msgsym = (get the message symbol of the Method object "mtd");

        newEnv = (create a new Environment object);
        for (i=0;i<(the number of arguments accommodated by "mtd");i++) {
                x = (get the "i"th argument from "mtd");
                if ("x" is evaluatable)
                        x = (the result from sending "eval" to "tobj");
                Put "x" into "newEnv" as the "i"th argument;
        }
        Push "newEnv" on the activation record stack;
        ret = (the result from sending "msgsym" to "tobj"
                under the new Environment object "newEnv");
        Pop the top of the activation record stack;
        return ret;
}
```

Figure 3.6 An evaluation of a Method object

The function MethodEvaluation is one of the C method objects which are looked up in the method dictionary for a Method object. It is invoked by sending a message "eval" to a Method object.

Figure 3.7 illustrates the execution of an expression "(1+2)+4". This expression is translated into two Method objects: one for sending "+" with 2 to 1, and another for sending "+" with 4 to an object returned from the execution of "1+2".



a	A message "eval" is sent to the method object M1.
b	M1 sends a message "eval" to its target object (M2).
c	M2 sends a message "eval" to its target object 1. 1 is returned as a result, because it is unchanged by evaluation.
d	M2 sends a message "eval" to its argument object 2. 2 is returned as a result, because it is unchanged by evaluation.
e	The integer object 1 receives "+" with an argument 2. 3 is returned after calculation. 3 becomes a real target object of the method M1.
f	M1 sends a message "eval" to its argument object 4. 4 is returned as a result, because it is unchanged by evaluation.
g	The integer object 3 receives "+" with an argument 4. 7 is returned after calculation.

Figure 3.7 The evaluation process of the expression (1+2)+4

3.3.5. A Method List

A Method list object has a slot containing a list (List object) of objects that is evaluated. It also holds a list of variable identifiers (String objects) and a File object for its

source codes. Figure 3.8 is a C language program for the evaluation of a Method list object. A Method list object is a translation unit for variable declarations plus statement-list.

```
MethodlistEvaluation(mtdlst, env)
OBJ mtdlst;
OBJ env;
{
      int i, l;
      OBJ x;
      if (HasTempVars(mtdlst))
                  /* if mtdlst has temporary variables */
            MakeRoomForTempVars(env,
                  CountTempVarsInMList(mtdlst));
                  /* then make room for them in the environment.*/
      l = CountMethodInMlist(mtdlst);
      for (i=0;i<l;i++) {
            GetMethodFromMlist(mtdlst, i, &x);
            x = SendMsg(x, evalLit, env);
                  /* executes each sequentially */
      }
      return x; /* the last evaluated one is returned */
}
```

Figure 3.8 The evaluation of a Method list object

The program evaluates the elements of its list sequentially. It also makes room for local variables if they are declared.

3.3.6. Assignment Statements

An assignment statement is translated into an Assignment object, which has two slots: 'lval' and 'rval'. The 'lval' slot holds an expression for an instance variable or local variable. It is used as the address in which a value is stored. The 'rval' slot holds a HOPL 'expression', which is stored after its evaluation.

There is another type of assignment called 'quoted assignment'. The quoted assignment uses ':=' instead of '=' in an ordinary assignment statement of HOPL. It differs in that the rval object is assigned to the location designated by the lval object without any evaluation. In the assignment

```
.a  =  1+2;
```

The instance variable 'a' is set to 3, which is the result of the evaluation of "1+2". However, in the quoted assignment

```
.a   :=   1+2;
```

the instance variable 'a' is set to a Method object "1+2" and it is not calculated. If the user types in just "a" under the directory in which the instance variable 'a' is created, then 3 is printed out after the evaluation of "1+2". If s/he types in ".a", then a dump listing for the Method object "1+2" will be seen.

It is not allowed to modify instance variables from outer directories. For example, the following statement is illegal.

```
!.dir.x = 10;          // This was executed under the directory !.
```

In order to change the value of !.dir.x to 10, the commands are as follows:

```
cd !.dir;              // Change the current working directory.
.x = 10;               // assignment
```

However, there is another way to change the value: by preparing a method to change the x's value and sending a message (corresponding to that method) to the directory "!.dir". That is the way HOPS encapsulates instance variables inside a compound object.

3.3.7. Conditional Statements and Iteration Statements

Although there are special constant objects, TRUE and FALSE (or NIL), they do not behave like those in Smalltalk-80; they do not receive messages like ifTrue and ifFalse. Instead, all the objects but FALSE (or NIL) are regarded as true in a conditional expression of HOPL. Only FALSE (or NIL) is regarded as false. That is the reason the public dictionary (which is inherited by every object including a simple object) has several methods for logical operations. "||" is a message symbol for a logical-or operation which returns FALSE only if both its target object and its argument object are FALSE. "&&" is a

message symbol for a logical-and operation which returns TRUE only if both its target object and its argument object are non-FALSE values.

The *if* statement, *while* statement, and *for* statement are translated into an object called a Control object. The first slot in a Control object is for one of four tokens: if-token, while-token, for-token, and return-token. If the token is an if-token, the remaining slots are for a conditional expression (evaluatable object), a then-statement (evaluatable object), and an else-statement (evaluatable object). If the token is a while-token, the remaining slots are for a conditional expression (evaluatable object) and a statement (evaluatable object). If the token is a for-token, the remaining slots are for an initial expression (evaluatable object), a conditional expression (evaluatable object), a repeated expression (evaluatable object), and a statement (evaluatable object).

The evaluation of a Control object is quite simple. For a Control object with its token being an if-token: if the conditional expression returns NIL after receiving a message "eval", then "eval" is sent to the else-statement. Otherwise, "eval" is sent to the then-statement. For a Control object with its token being a while-token: while the conditional expression returns a non-NIL object after receiving a message "eval", the statement continues to receive "eval". Finally, for a Control object with its token being a for-token: first, the initial-expression receives "eval", then while the conditional expression returns a non-NIL object after receiving "eval", the statement and the repeated expression receive "eval".

3.3.8. Jump Statements and Interruption

A *return* statement is the only jump statement provided in HOPS (*break* and *continue* in C language are not supported). It is translated into a Control object with its token being return-token. If it receives "eval", a special global variable in C language (called a return flag) gets set to 1. The return flag is checked every time an object corresponding to a statement is evaluated in a while-statement Control object, a for-

statement Control object, and a Method list object. If that flag is set to 1, then any evaluation terminates immediately, returning the most recently evaluated value. The return flag is reset to zero at the end of the evaluation of a Method object, so that jumps by a return statement affect only the inside of a method. That is how a return statement works. Accordingly, note that a return statement inside a Block object means to return from the context of the Block object, not from the method where the Block object is evaluated, as in Smalltalk-80.

In an interpreter-type system, it is important to provide a way to force an exit from its execution. HOPS takes advantage of a signal called SIGINT, often bound to a control-C key in UNIX. If the user hits a control-C key once, then every evaluation tries by any means to terminate immediately. There is a flag that is checked just like the return flag and is never reset even in the evaluation of a Method object. That flag is reset only at the beginning of every interpretation loop (repetition of the reading line, evaluating it, and printing the evaluated value). This flag gets set to 1 when the system receives a signal SIGINT. If a control-C key is hit seven times while this flag is not reset, then the system terminates after doing "epilogue" processes such as closing edit buffers.

3.3.9. A Block Object

A Block object is an evaluatable object whose evaluation can be deterred, because it is constant against a message "eval", although it is evaluated when receiving a message "value". It forms a "lexical closure" [14], i.e., a list of statements following the rules of lexical scoping. The following example (quoted from Steele [14]) is written in Lisp:

```
(defun adder (x) (function (lambda (y) (+ x y))))
```

The result of the "(adder 3)" in Lisp expression is a function that will add 3 to its argument:

```
(setq add3 (adder 3))
(funcall add3 5) » 8
```

where "»" means an evaluated value appears on the right hand side. This can be written using a HOPL Block object:

```
.adder := var x;{ [var y; $x+$y;]; }
adder 3:value 5  » 8
```

Here is another example (quoted from Steele [14]) rewritten in HOPL:

```
.two_funs := var x;{List:new [$x;], [var y;$x=$y;]}
.funs=two_funs 6
.funs at 0:value » 6
.funs at 1:value 43 » 43
.funs at 0:value » 43
```

The "two_funs" method returns a new list (List object) of two Block objects. This example shows that a Block object preserves the activation record for an environment where the Block object is lexically defined. Let us look at the process to preserve an activation record, using the example shown in Figure 3.9.

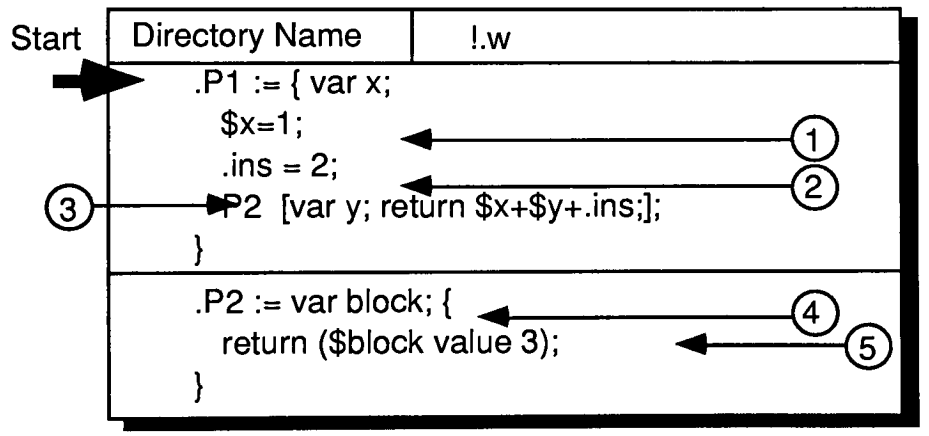


Figure 3.9 an example for the evaluation of a Block object

In the example, there are two methods: P1 and P2. A Block object is passed as an argument for P2, and P2 evaluates that Block object by sending "value".

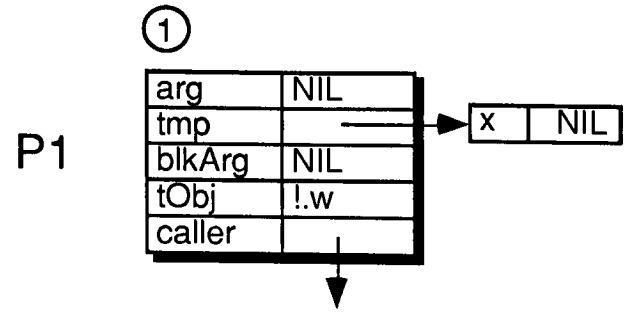


Figure 3.9 (a) An activation record for the Block object's evaluation

Figure 3.9 (a) shows the activation record when the program counter is located at ① in Figure 3.9. When the variable x gets set to 1, the activation record is changed as shown in Figure 3.9 (b).

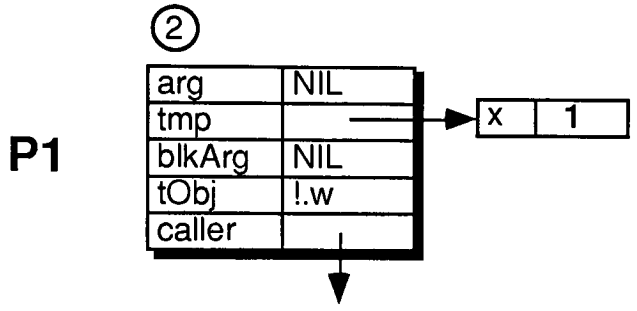


Figure 3.9 (b) An activation record for the Block object's evaluation

Before P2 receives the Block object as an argument, there is a change inside the Block object. The Block object, receiving a message "create" (internally inserted by the HOPL parser), creates a new Block object which shares an executable object (method) with the original Block object and initializes two slots inside: a target object and an environmental object. They are copied from the current activation record (Environment object). Figure 3.9 (c) illustrates the moment in which the new Block object is created.

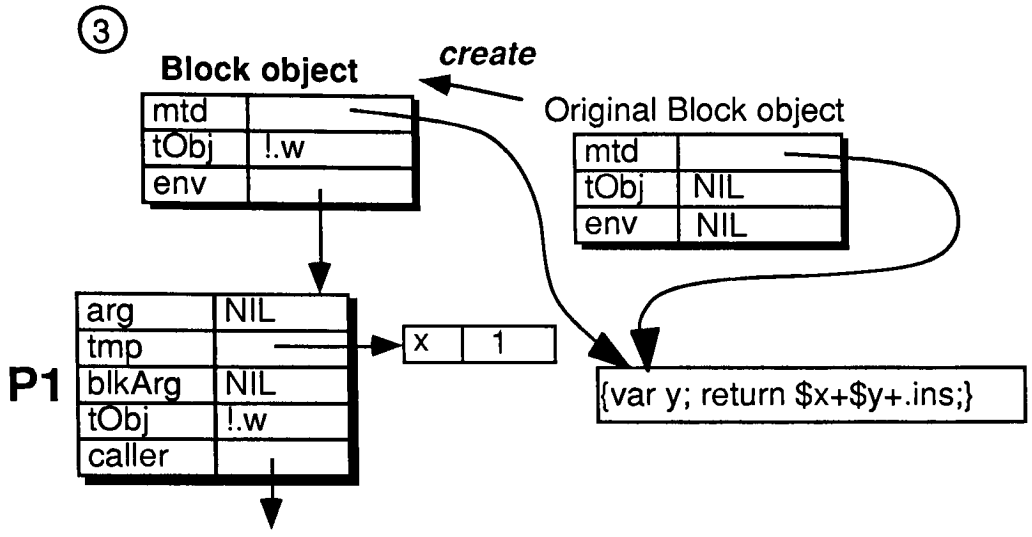


Figure 3.9 (c) An activation record for the Block object's evaluation

The initialized Block object is bound to a variable "block" in a new activation record as shown in Figure 3.9 (e).

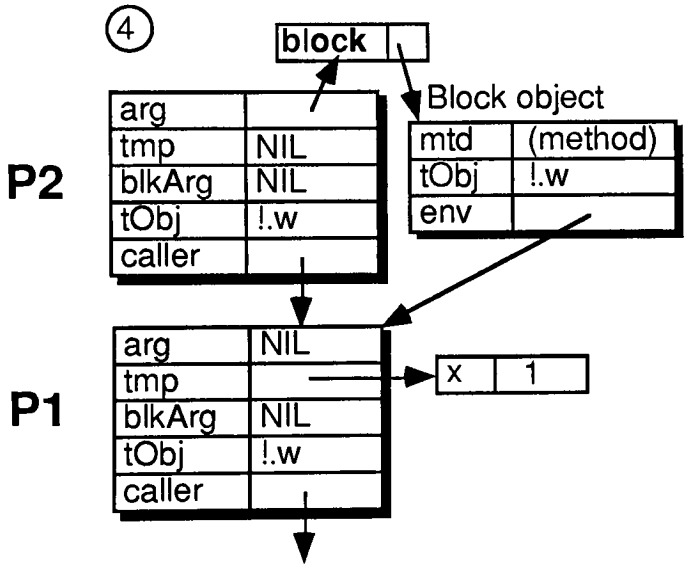


Figure 3.9 (d) Activation records for the Block object's evaluation

Now P2 sends "value" to the Block object. However, the method inside the Block object demands accesses to the variables under the environment where the Block object received "create". Therefore, the Block object, receiving "value", constructs another activation record which shares temporary variables and argument variables with the previous record,

pointed to by one of the slots in the Block object. The new record also includes "!.w" as its target object, since it was the target object for the previous record. Finally, the Block object can execute its method according to the rules of lexical scoping, as shown in Figure 3.9 (f).

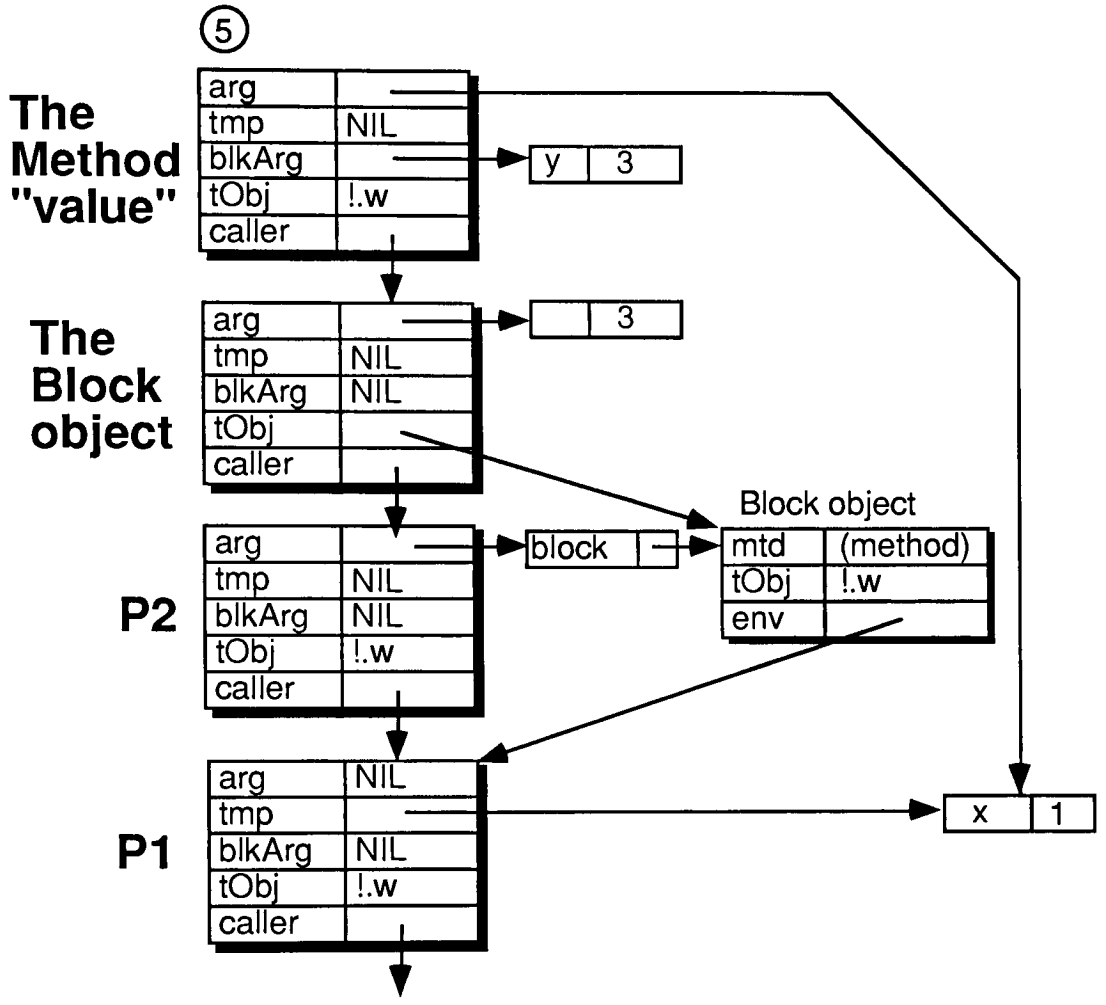


Figure 3.9 (e) Activation records for the Block object's evaluation

3.3.10. Scope Rules

(1) Instance Variables: Instance variables are syntactically specified as follows:

```
instance-variable:
        . identifier
        . character-string-object

hierarchical-object:
        instance-variable hierarchical-object
        ^ hierarchical-object
        instance-variable
        ^

constant-object:
        ! hierarchical-object opt
        ...
```

Semantically, there is no difference between retrieving instance variables' values and sending messages. For instance, the expressions

```
!.dir.x
.p
^.prog.x
```

are translated into the following codes (not allowed syntactically).

```
(! . "dir"). "x"
SELF . "p"
((SELF ^). "prog"). "x"
```

Here "." is a message symbol accompanied by a variable name and "^" is a message symbol with no arguments. The two messages are included in the method dictionary for a Directory object. Because of this translation, instance variables can be referenced in any part of a method that is declared in the directory where those instance variables appear. Basically, if a full path name (starting with "!") is used, all the instance variables can be visible. An instance variable that belongs to an inherited directory is *not* visible to an inheriting directory for the same reason described above. In addition, an instance variable that belongs to an inherited directory is *not* visible even to the inherited directory itself when a method containing references to the variable is invoked as an inherited method. Consider the following methods.

```
.p:={.x=1;}          //a method for "!.demo"

.r:={.x=3;}          //a method for "!.sup"
```

Suppose that "!.demo" inherits methods from "!.sup". If "r" is sent to "!.sup", then the instance variable "x" is assigned to 3 under "!.sup". If "p" is sent to "!.demo", then the instance variable "x" is assigned to 1 under "!.demo". But if "r" is sent to "!.demo", then the instance variable "x" is assigned to 3 *under "!.demo"*. So the user has to make sure that every instance variable which is located in the inherited directories is initialized correctly under an inheriting directory. For this purpose, it is encouraged that an inherited directory have a method "init" which initializes all the instance variables that should be declared under a directory. That method must be called right after the creation of a compound object (directory).

(2) Local Variables: Local variables (including parameters and temporary variables inside a method) are instantiated upon entry to a method and are bound to an activation record. When exiting from the method, the reference count of the current activation record is decremented by "DecObj" (described in the previous section "Reference Counts") and if it is zero, then the record is abandoned and "DecObj" spreads to all the objects bound to that record recursively.

(3) Block Arguments: A Block object provides a lexical scope. That is, the activation record created under an environment where a Block object appears is preserved, as long as the Block object is alive, as described in the previous section "Block Object". A Block argument name hides a local variable name if their names are identical. In addition, there is a special scoping rule for a Block object: if a Block object, say B, appears inside another Block object, say A, the Block argument declared inside A is not visible to B. The following shows illegal statements for a method.

```
.p:={var x;
     return [var y;
               $x=$y;                    //OK
               return [var z;
                         $x=$z;          //OK
                         $y=$z;          // syntax error
                                         //"y" not found
                       ];
            ];
}
```

(4) Class Name An object corresponding to Class in Smalltalk-80 is implemented simply as a Dictionary object in HOPS. All those Dictionary objects are arranged in a directory "*!.dict*", as described in the previous section "Creation of Objects". Each dictionary accepts a message "new" and then returns a new instance object which has that dictionary as its class. If the user wants to create a new List object containing 1,2, and 3 and assign it to an instance variable "x", for instance, then the following statement will do.

```
.x=!.dict.List new 1,2,3
```

The method "new" for a List object class can have a variable number of arguments, which become elements of a new List object. However, there is an abbreviation for this expression. Because the directory "!.dict" is inherited by the root directory (!) and every method available on the root directory is inherited by every directory, the following expression is exactly the same as the above.

```
.x = List:new 1,2,3
```

Here, the identifier "List" works as a message symbol. That message is finally sent to "!.dict" through the inheritance mechanism, the HOPL interpreter (SendMsg) returns it as it is (because a Dictionary object is constant throughout the evaluation process), and the result receives the message "new". The message "new"should follow a colon ':', which enables it to cascade messages. This convention gives a more concise format for creating a new instance object.

3.3.11. Debugger

The debugger designed for HOPS provides a break at the beginning and end of any message-passing, so that the user can see the sequence of message-passing and the content of activation records. Its implementation was quite easy. Every action of sending messages is done through the C language function SendMsg described in the previous section. The debugger routine is called right before SendMsg body starts and right after SendMsg returns. This simplifies the design of the debugger.

You can toggle a debug mode on and off by sending "debug" to a directory. With the debug mode on, the system waits for a debug command input every time SendMsg is called or returned. The debugger employs commands as follows:

(1) 'Leap': The debugger usually keeps track of the system status before and after message-passings. However, it is sometimes useful to see only the result from several message-passings without tracing each message-passing. The command 'l' allows the user to leap to the same depth of SendMsg's recursive calls. That means that if a control is just before the execution of SendMsg in the debug mode and 'l' is typed in, then the next break point is right after that execution without breaking any nested calls of SendMsg.

(2) Information about Activation Records: You can see the chain of activation records by typing in 'i' at any time in the debug mode. A target object, temporary variables, arguments, and Block arguments are printed out for each activation record.

(3) Detection of 'ILLEGAL' objects: HOPS does not abort an execution in which a method is not found. While Smalltalk-80 or Little Smalltalk automatically terminates the execution of an application program as soon as a method is not found, HOPS has two modes:the mode in which HOPS keeps silent and the mode in which HOPS goes into a debug mode as soon as a system constant object ILLEGAL is detected as the result value from "SendMsg". Those modes can be toggled by sending the message "verbose" to the

HOPL interpreter. The ILLEGAL objet is an object that returns ILLEGAL when receiving any message. This allows the user to check if a method or an instance variable is defined on a directory. For example, the following method 'setup' is a method that compiles a program under a directory. First, "setup" checks for an object ".SETUP". If there is no such object, then a File object "main" receives "trans". That File object includes method definitions which are translated into executable objects by "trans". Otherwise, "setup" sends "run" to a File object "SETUP" to run commands in it sequentially.

```
.setup := {
     if (.SETUP == ILLEGAL)
          .main trans;
     else
          .SETUP run;
}
```

In the debug mode, there is a command to break the execution as soon as the ILLEGAL object is detected as the result of SendMsg. This command makes it easy to find an error in a program that does not have any checks for ILLEGAL.

3.4. Inheritance Mechanism

There are two kinds of objects in HOPS, a *simple object* and a *compound object* . They are different in the way the system searches for a method. When a simple object, such as an Integer object, receives a message, it is looked up first in its method dictionary (called magic), then in the public method dictionary, which contains operations common to all objects. A compound object is a Directory object, which is the only object that is allowed to inherit methods. A message to a compound object is looked up first (1) in the Directory object's method dictionary, (2) in a user's methods defined under a directory, (3) in user's methods defined under any inherited directories, (4) in the methods defined under the root directory, (5) in the methods defined under the root directory's inherited directories, and finally (6) in the public method dictionary. As seen in this sequence, the user's methods are inherited through (3). Let us take a look at the inheritance mechanism in HOPS.

Figure 3.10 shows a sample program for an inheritance mechanism. Here, a directory "!.demo" inherits a method "hello" from a directory "!.sup". This demo program starts by sending "say" to !.demo. The method "say" sends "hello" to SUPER, that is, !.sup. So the program control moves to the method "hello" inside !.sup. That method prints out "sup\n" and then sends "hello" to SELF, that is, !.demo *not !.sup*. (Note that the method for "hello" inside "!.sup" does not work alone. In other words, if "!.sup" receives "hello" directly, not through the inheritance such as this example, then the method calls itself recursively forever.) The program control moves to the method "hello" inside !.demo and "demo\n" is printed out.

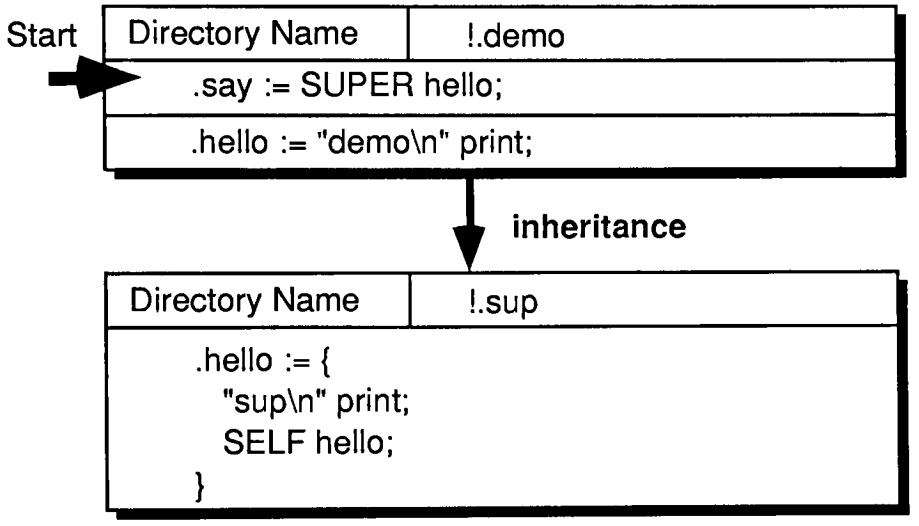


Figure 3.10 An inheritance mechanism

This process is explained together with activation records by Figure 3.11. An activation record (Environment object) has a slot "tObj", which holds a target object, and a slot "hObj", which holds an object possessing the current running method. When the method "say" is invoked, both tObj and hObj hold !.demo because "say" is sent to !.demo and "say" belongs to !.demo. However, when the method "hello" for !.sup is invoked, hObj changes to !.sup because the method "hello" belongs to !.sup. Finally, when the method "hello" for !.demo is invoked, hObj changes to !.demo because the method "hello" belongs to !demo in turn. Thus, tObj always reflects an object corresponding to SELF, and hObj is

used to searching for methods belonging to SUPER. Everytime a method is found, hObj is updated by the HOPL interpreter.

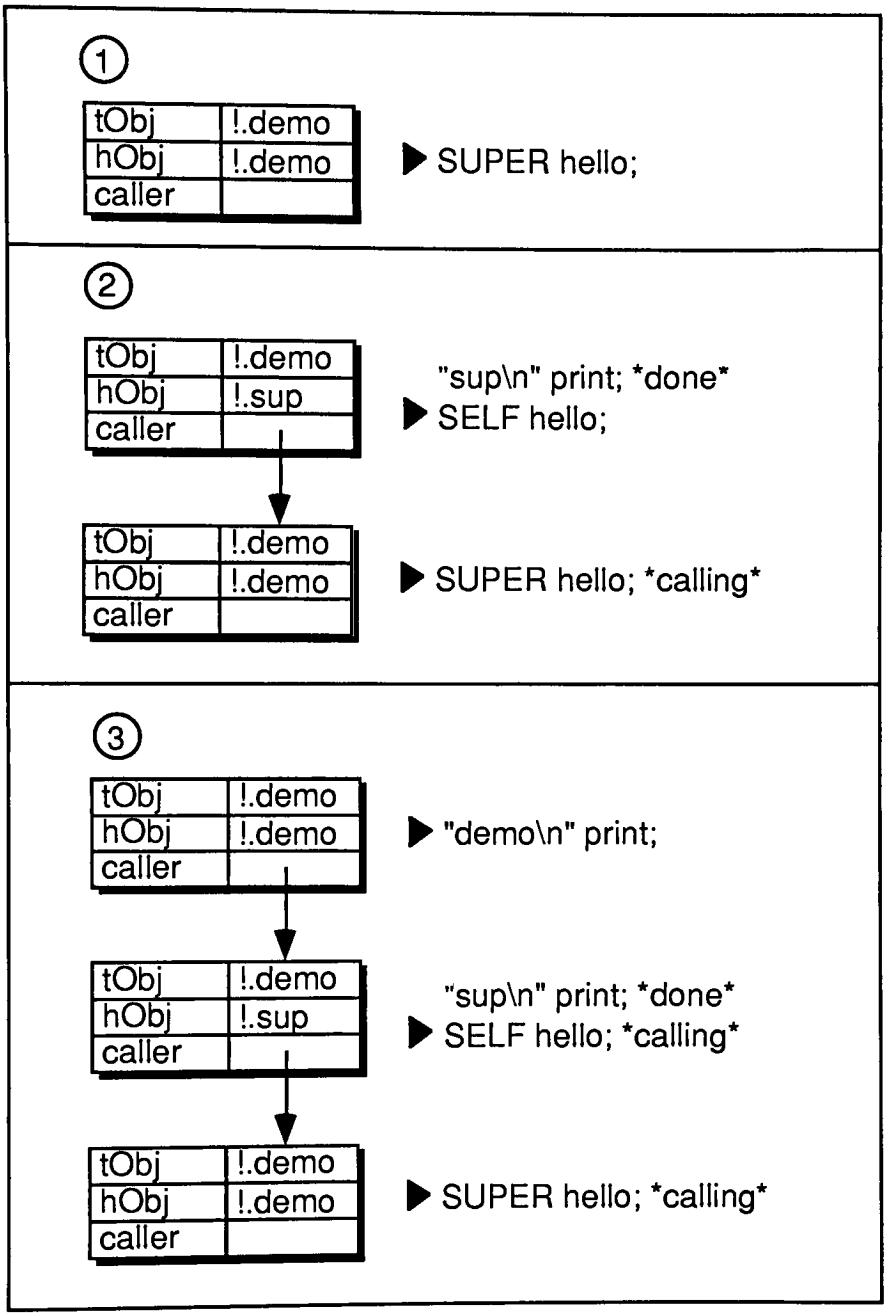


Figure 3.11 Inheritance and activation records

3.5. Hierarchical Arrangement of Objects

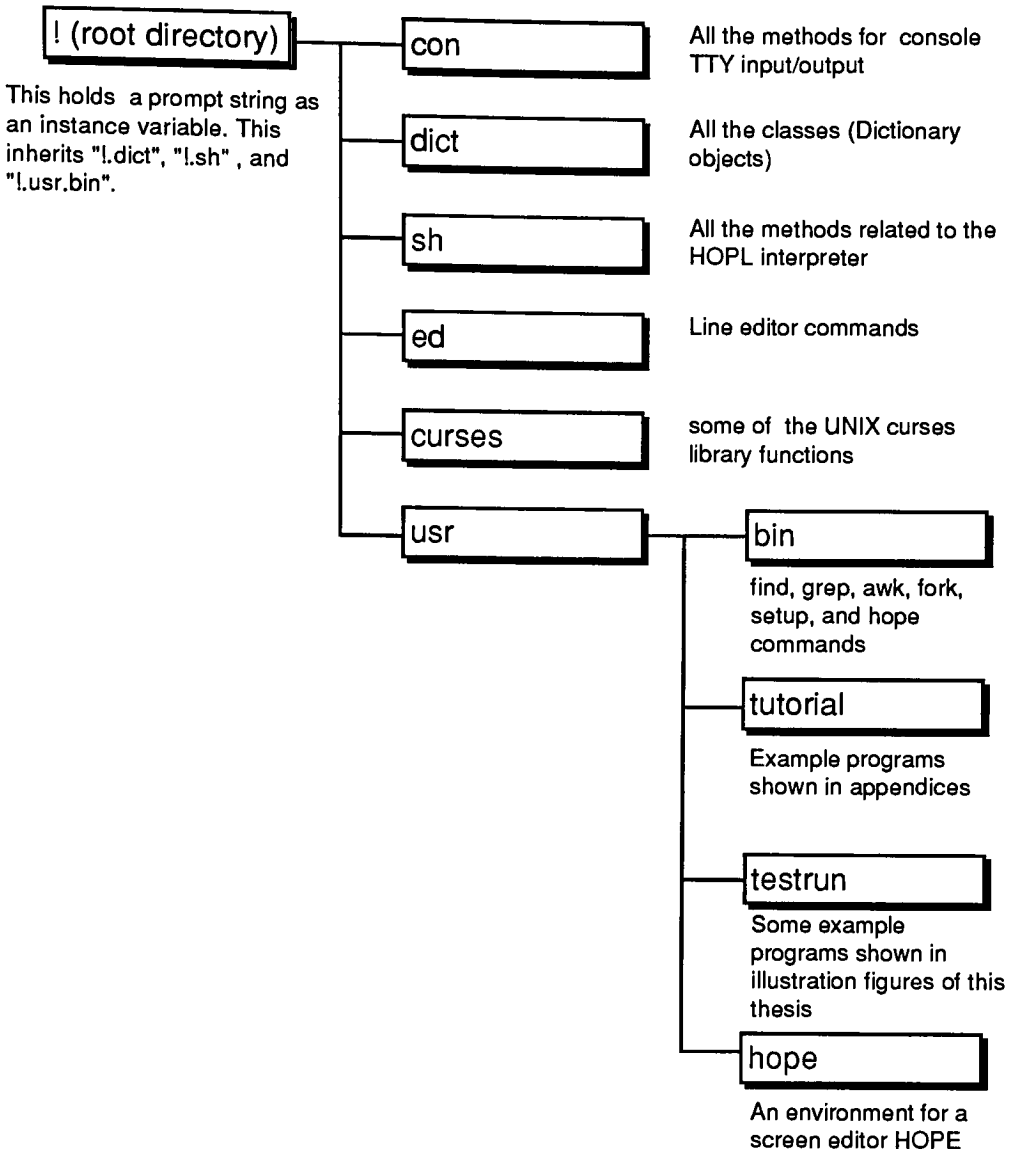


Figure 3.12 Hierarchically arrangement of objects

Figure 3.12 illustrates the initial arrangement of HOPS objects. In addition, the users can locate their programs on any directory or create a new directory in which to embed those programs. Programs in HOPS are normally saved in a text file. The text file is also implemented as an object called a File object. The following sections discuss the File object and its relationship to UNIX files.

3.5.1. A File Object

A File object is a text file which is often used for saving programs written in HOPL. That object has no slots in it; only the object ID number itself becomes a file name in the UNIX file system. Every file created for a File object is pooled under the UNIX directory "./HOSTFILE". For example, suppose that a new File object is numbered 2345, which is actually the object ID. First, the HOPS system creates a new UNIX file named "./HOSTFILE/2345". The methods sent to that File object reference "./HOSTFILE/2345". If this File object is discarded, the system sends a special message "!.kill" to the File object and the method for !.kill *unlinks* (UNIX system call, which removes a file) the file "./HOSTFILE/2345". This feature comes from an object flag KILL described in the section "Structure of Objects."

There are some important methods for a File object: "run", "trans", "vi", and "new". A File object receiving "run" should contain a sequence of commands. The method "run" sequentially executes those commands. A File object receiving "trans" should contain a sequence of method definitions. After "trans" is sent to such a File object, the methods defined in the File object are compiled and ready to be executed. "Vi" allows the user to edit the file content by using UNIX *vi*. If the user wants to create a new File object, then the following statement might be the most appropriate.

```
.newFile=File:new:vi
```

If the method "new" for a File object receives a String object as its argument, then the argument is regarded as a UNIX file path name. The created File object is linked to the UNIX file path name by a *link* (UNIX system call), so that every modification is reflected onto the linked UNIX file.

3.5.2. Coordination with the UNIX File System

There is a special instance variable called "!hostdir", which holds the UNIX directory path name. If a File object is created under those directories which have

"!hostdir", then the UNIX file with the File object's name is automatically created under that UNIX directory path. If a Directory object is created under such directories that have "!hostdir", then the UNIX directory with the Directory object's name is automatically created under that UNIX directory path and the created Directory object has "!hostdir" that includes the new UNIX directory path name.

The method "mount" for a Directory object is involved in this "!hostdir" variable. Let us consider the following example:

```
mount "usr"
```

where the argument "usr" means a UNIX directory under the current UNIX directory. What this statement does is copy to HOPS the tree structure under the UNIX directory "usr". If a UNIX file is copied to HOPS, a File object is created with the same name under the corresponding Directory object and is automatically linked to the UNIX file. If a UNIX directory is copied to HOPS, a Directory object is created with the same name and "!hostdir" containing the corresponding UNIX directory path name is automatically added to that Directory object. These procedures are recursively repeated throughout the tree structure of the UNIX file system. This "mount" provides a *canned* hierarchical file system environment for the user, taking advantage of the foundational UNIX file system.

3.6. HOPE

3.6.1. The Line Editor

The original design of the line editor implemented in HOPS comes from Kernighan [8]. Based on a scratch file which stores lines in an editing text, the editor enables several basic operations on the lines. Those operations are implemented as methods in HOPS and described in Appendix C. This line editor is used for constructing the screen editor HOPE.

3.6.2. The Curses Library in C language

The *curses* library, available in the UNIX environment, contains functions to control a screen cursor and to edit characters or lines displayed on a screen. This was necessary in order to implement a screen editor. There are some methods that are directly connected to the curses library. Those are described in Appendix C.

There is a useful function called a *command line editor*, which also makes use of the curses library. It enables the user to edit the previous input line like the *Korn Shell* [9] in the UNIX system. The command line editor is written in C language, and built in the line input routine which supplies a character string line to the lexical analyzer. The command line editor is invoked by hitting the ESCAPE key and terminated by hitting the RETURN key or the NEWLINE key.

3.6.3. The HOPE Main Program Structure

Figure 3.13 illustrates the format of the screen editor HOPE. It consists of two separated TTY windows: an editor window for displaying and editing a text, and an execution window for executing commands and for seeing the result. The user can switch the cursor from the editor window to the execution window by typing the character '!' in the editor window. If the user types in "quit" in the execution window, then the cursor moves back to the previous position in the editor window. Since the editor is based on the curses library, it supports many kinds of terminals. The editing commands are a subset of the UNIX *vi* editing commands.

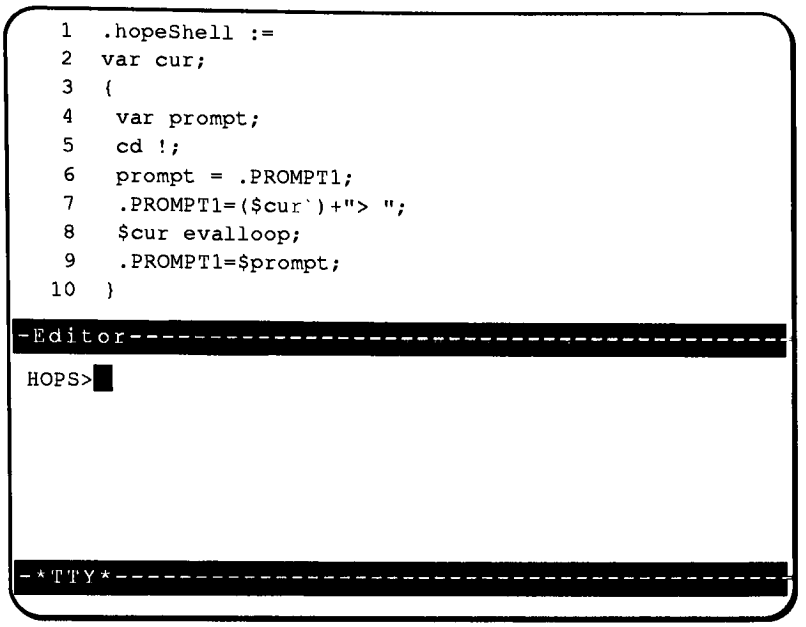


Figure 3.13 The HOPE screen editor

The main program for HOPE is included in "!.usr.hope". Since "!.usr.hope" is not inherited by the root directory, the method "hope" described above cannot be directly accessed. Therefore, "!.usr.bin" (inherited by the root directory) includes the following method.

```
.hope :=
var file;
{
     !.usr.hope hope SELF, file;
}
```

3.7. Software Development using HOPS

3.7.1. Prototyping

HOPS provides a convenient way to define methods instantly, without preparing a file to be compiled. Suppose that a method to print out "hello, world" is needed. The following command creates a new method for that purpose in the current directory.

```
.hello:={"hello, world\n" print;}
```

The quoted assignment operator ":=" assigns the right-hand-side expression to the left-hand-side instance variable *without evaluating that expression.* That is, the executable

object for "{"hello, world\n" print;}" is directly assigned to the instance variable. This approach gives great facility for prototyping.

Here is another sample program containing two arguments.

```
.add:=var x,y;{return $x+$y;}
```

The method "add" receives two arguments "x" and "y", sends "+" to "x" with an argument "y", and returns the result.

Finally, here is another example program to show the use of local variables.

```
var tmp;$tmp=.x;.x=.y;.y=$tmp
```

This program swaps two instance variables, ".x" and ".y". This is not the definition of a method, but a simple compound statement that is typed in. The grammar of typed-in statements allows the local variable declaration at the beginning of the line, so that those variables can be used only when the typed-in commands are working.

3.7.2. A "setup" Message

When the user is involved in a project, it might be convenient to locate every source file on one directory and to be able to compile it by invoking just one command. In UNIX, the command 'make' helps the programmer with documentation of a program linkage, the "instant" program construction, and minimization of the compiling time. Not having the information about modification time of a File object, HOPS employs a command called "setup" only for the linkage documentation and the instant program construction.

The command "setup" is a method for a Directory object. It is inherited from the directory "!.usr.bin" and written in HOPL as follows:

```
.setup := {
     if (.SETUP == ILLEGAL)
          .main trans;
     else
          .SETUP run;
}
```

This command was introduced already, in the section "Debugger". The File object ".SETUP" must include commands to compile all the source files under the current directory and to inherit methods from appropriate objects. If the user does not need such a large, organized project, then s/he can define all methods in a File object ".main" without having ".SETUP".

For example, consider a program to print out "hello, world". Let us use the ".SETUP" file first. The content of the ".SETUP" file is just one line:

```
.hello trans;
```

To create a new File object using an editor, type:

```
.SETUP=File:new:vi
```

The other file to be prepared is ".hello" which includes:

```
.hello:=
{
     "hello, world\n" print;
}
```

We created two File objects in the current directory: ".SETUP" and ".hello". If the "setup" message is sent to the current directory, the ".SETUP" is run, and finally ".hello" becomes translated (into a Method list) by receiving the message "trans". Now we can send a message "hello" to the current directory. When "hello" is typed, two lines will be printed out.

```
hello                  // type it under the current directory
hello, world           // the method for "hello" prints this out
hello, world           // HOPS prints out the last evaluation.
```

The instance variable ".hello" holds the Method list for "hello", not the File object including the method definition text. The File object for the definition of ".hello" seems to be gone, but there is a slot inside the Method list, grabbing that File object. Actually, there are several messages which are sent to that File object via the Method list. The message "vi" is

such a case. When the Method list receives "vi", it sends "vi" to the File object grabbed by one of its slots.

```
.hello vi
```

At this point, we can edit the definition of "hello" using the UNIX screen editor "vi". In order to recompile it, "setup" can be sent to the current directory again. In the "setup" process, "trans" is sent to ".hello", which is a Method list object this time. The message "trans" is sent to the edited File object via the Method list object, just like the message "vi".

The case of the "hello" program can be handled without using the ".SETUP" file. A File object ".main" is created which includes the same content as the File object ".hello" and "setup" is sent. The Method list object for the "hello" method is created and named ".hello". If the user wanted to change the content of the source code, "vi" would be sent to ".main" and "setup" would be sent to the current directory again. In this case, the structure inside the directory is more concise, since only one file ".main" is sufficient to construct a method. To summarize the above procedures,

1. For a large project, prepare a File object ".SETUP" including all compilation commands and other File objects including method definitions. To compile or recompile them, just send "setup" to the directory to which those objects belong.

2. For a small project, prepare only one file ".main" including all method definitions. To compile or recompile them, just send "setup" to the directory which ".main" belongs to.

If the directory is "mounted", then those files created on that HOPS directory are automatically linked to the UNIX files under a UNIX directory corresponding to that HOPS directory. As a result, those files are permanently stored.

3.7.3. A Project Using Inheritance

The easiest way to inherit methods from other objects is to use a message "inherit" for a Directory method. The following example shows the inheritance from a simple object, an Integer object.

```
inherit 5
+2              » 7
*3              » 15
```

Because the current directory inherits an Integer object "5", if it receives "+" with "2", then it returns "7". Generally, "inherit" appends its argument at the end of an inheritance list that belongs to a directory. The argument of "inherit" can be any kind of object: a simple object or a compound object (a Directory object).

Let us consider a more complicated example of a *prioritized set.* To make the example simple, the number of implemented methods is limited to three: "<", "add", and "del". The message "<" literally means "less than" according to the priority that a set has. The user can add an element to a set by "add" and delete an element from a set by "del". The method for "<" is implemented in a directory, '!.project.Prio', different from the directory, '!.project.PSet', in which "add" and "del" are defined. The following program is written in the File object "!.project.Prio.main".

```
.init:=
var p;
{
     .priority=$p;
}
."<":=
var arg;
{
     (priority) < ($arg priority);
}
```

The message "init" has to create and initialize all the instance variables used under the inherited object. The following program is written in a File object "!.project.PrioSet.main".

```
.init:=
var p;
{
     className "PrioSet";
          /* the message className creates
          the instance variable "className"
          whose value is the argument.*/
     .aSet=List:new;
     SUPER init $p;
}
.add:=
var element;
{
     if ((.aSet lookup $element) == ILLEGAL)
          .aSet add $element;
}
.del:=
var element;
{
     if ((.aSet lookup $element) <> ILLEGAL)
          .aSet del $element;
}
```

The content of "!.project.PSet.SETUP" follows:

```
^.Prio setup;          // compilation for "!.project.Prio"
setup;                 // compilation for "!.project.PSet"
inheritList (List:new ^.Prio);
```

While the message "inherit" appends an object to an inheritance list, the message "inheritList" switches the inheritance list to the given argument. This ".SETUP" file initializes the inheritance list with a one-element List object including only "!.project.Prio".

The following statements are typed under "!.project".

```
.PrioSet setup
.s1=PrioSet:fork 1
.s2=PrioSet:fork 3
```

The message "fork" is a built-in message for a Directory object (defined in "!.usr.bin" of the standard HOPS environment).

The message "fork" creates a compound object whose class name is copied from the callee's "className" instance variable. The compound object inherits all the methods

which the object receiving "fork" possesses or inherits. Then, it receives "init" having the same arguments as "fork". Using ".s1" and ".s2", the following statements can be typed.

```
.s1 < .s2              »TRUE
.s2 < .s1              »NIL
.s1 add "abc"          »Set of "abc"
.s2 add 345,'a'        »Set of 345 and 'a'
.s2 del 345            »Set of 'a'
```

If the user wants to see inside of ".s1", then the command "cd .s1" allows it. After invoking that command, all the instance variables can be examined by using messages for a Directory object, such as "l" and "ll".

Finally, the method for "fork" is written in HOPL as follows:

```
.fork:=
{
      var x, ih;
      $x = new;          // a new Directory object
      if ( (className) <> ILLEGAL )        // if className is defined
           $x className (className);       //copy the name
      $ih = List:new SELF;
      $x inheritList ($ih + (inheritList));
                         // append SELF at the top of the inheritance list
      sendmsg $x, "init", (argv);
                         // invoke "init" with the same arguments
      return $x;
}
```

4. Sample Programs in HOPL — find, grep, and awk

Let us take a look at practical programs written in HOPL. *Find*, *grep*, and *awk* are the famous, useful commands that UNIX provides to the user. Figure 3.14, 3.15, and 3.16 show how shortly they can be implemented. *Find* implemented here receives no arguments and behaves like "find . -print" in UNIX. Although *awk* should be implemented as a method for a File object, it is a method for a Directory object. It receives the first argument as a File object (to which the *awk* command is applied) and the second argument as a Block object (which contains *awk* commands written in HOPL). Figure 3.17, 3.18, and 3.19 show the result of those executions.

```
.find :=
{
     do [ var name, obj;
               /* The current directory responds to "do"
               by passing a name and value of each instance
                variable.  This block is repeatedly called for
                each variable.*/
          if ( ($obj isDir) not ) {
                    /* if the instance variable is
                    not a Directory object */
               pwd;
                    /* print out the path name */
               "."+$name+"\n" print;
                    /* print out the variable name */
          } else {
                    /* if the instance variable is a
                    Directory object */
               $obj pwd; ".\n" print;
                    /* print out the directory path name */
               $obj find;
                    /* recursively calls "find" for that directory */
          }
     ];
     return TRUE;
}
```

Figure 3.14 A program "find"

```
.grep :=
var pattern;
/* must be a String object */
{
      var filename, lno, file, matchNo;
      matchNo = 0;
      do[ var name, obj;
            filename = $name;
                  /* so that the name is accessible in another Block */
            if ( $obj isInstanceOf (File) )
                  file = $obj;
            else if ( $obj isInstanceOf (Mtdlist) )
                  /* if it is Mtdlist, it contains its documentation */
                  file = $obj doc;
            else return;
                  /* returns from this block, continues the iteration */
            lno=0;
            $file lineDo [
                  /* "lineDo" passes each line in the file as an argument*/
                  var line;
                  lno = lno + 1;
                  /* count up */
                  if ( $pattern in $line ) {
                        /* if that line includes the pattern */
                        ".%O:line %O: %O\n"
                        /* print it with a format */
                        format filename,lno,line:
                        print;
                        matchNo = matchNo+1;
                  }
            ];
      ];
      return ("%O match(es) found" format matchNo);
/* returns the information as a string rather than explicitly printing it out.
It will be printed by the HOPL interpreter as the result of this method */
}
```

Figure 3.15 A program "grep"

```
.awk :=
var file, blk;
/* receives a File object and a Block object that includes awk commands */
{
      var list;
      $file lineDo [ var oneline;
            $list = $oneline tokenlist " \t";
                  /* "Tokenlist" dissolves a String object while
                  the characters included in the argument String object
                  are acting as delimiters for tokens.*/
            $list = List:new:append $oneline:+list;
                  /* According to the SPEC of awk, the first argument
                  must be a whole line. */
            $blk self:sendmsg $blk,"value",$list;
/* sends "value", with elements of the list being arguments. "$Blk self" is a
directory under which the block is created. */
      ];
}
```

Figure 3.16 A program for a pseudo "awk"

```
!.usr.testrun.Nproject.
!.usr.testrun.Nproject.Magnitude.
!.usr.testrun.Nproject.Magnitude.main
!.usr.testrun.Nproject.Number.
!.usr.testrun.Nproject.Number.main
!.usr.testrun.blockEx.
!.usr.testrun.blockEx.main
!.usr.testrun.inheritEx.
!.usr.testrun.inheritEx.demo.
!.usr.testrun.inheritEx.demo.SETUP
!.usr.testrun.inheritEx.demo.main
!.usr.testrun.inheritEx.sup.
!.usr.testrun.inheritEx.sup.goodbye
!.usr.testrun.inheritEx.sup.main
!.usr.testrun.pipe.
!.usr.testrun.pipe.main
!.usr.testrun.project.
!.usr.testrun.project.SETUP
!.usr.testrun.project.hello
!.usr.testrun.project.profile
```

Figure 3.17 The result of a command 'find' under "!.usr.testrun"

```
.find:line 4: if ( ($obj isDir) not ) {
.fork:line 8: if ( (className) <> ILLEGAL )
.grep:line 8: if ( $obj isInstanceOf (File) )
.grep:line 10: else if ( $obj isInstanceOf (Mtdlist) )
.grep:line 17: if ( $pattern in $line ) {
.select:line 10: if ( $blk value x, y )
.setup:line 6: if ( .SETUP <> ILLEGAL )
```

Figure 3.18 The result of a command 'grep "if"' under "!.usr.bin"

```
.awk
var
{
var
$file
$list
$list
$blk
];
}
```

Figure 3.19 The result of a program:
'awk (.awk doc),[$2+'\n" print;]'
(Note: the message "doc" retrieves a File object from a Method list object)

5. Conclusions

5.1. Summary

Object-oriented programming has powerful advantages that enable programmers to package a functional element of their programs as an *object* In order to strengthen its power, HOPS (Hierarchical / Object-oriented Programming Environment System) , developed in this thesis project, not only adopts the object-oriented programming for its paradigm, but also hierarchically arranges all the resource objects (such as files, directories, simple integer data and so on), so that the user can access and manage the resources as easily as in the UNIX file system.

An object-oriented language HOPL (Hierarchical / Object-oriented Programming Language) allows the user to design application programs or to manipulate the system resources. The programs written in HOPL can be typed in as commands, compiled, and stored as methods, which can be inherited from other objects. The HOPL grammar is quite similar to C language.

The interpretation process is implemented as message-passing rather than the typical *byte code* implementation, . This makes it easier to understand the system's dynamic behavior, which can be observed by using the HOPS built-in debugger.

HOPS can share the UNIX directories and files as its resources. As a result, the user can store and modify those resources without worrying about the duplication of resources.

5.2. Alternative Approaches for an Improved System

Although the *byte code* was not adopted for the reason described above, its benefit in execution speed is still attractive. It eliminates some processes for traversing the executable objects (such as Method list objects). In addition, it can implement a process

easily. HOPS does not support a process object (allowing multi-tasking), because of an inherent difficulty. The HOPL interpretation calls the C language function 'SendMsg' recursively; it makes use of the UNIX user program stack. To implement a process object, the stacks for process objects should be switched with each other. Since the byte code simulates a stack, it is easier to switch the stack.

However, the HOPS executable objects can help in developing the byte code, because those objects apparently represent the parsing tree for input messages. The byte code can be created by traversing those objects.

5.3. Suggestions for Future Extensions

Currently, HOPS uses a 16-bit-address to identify an object. Because the most significant bit of that address indicates whether or not the object is a system constant, HOPS address space is limited to 15-bit. As an expansion in the future, a 32-bit-address could be considered.

In addition, a process object could be implemented as described in the previous section. One of the interesting features to be achieved about a process object is the coordination with the host operating system's processes (which means the UNIX processes in the current version of HOPS). There might be a special directory, which all the processes created in a certain condition belong to as instance variables, so that the user can monitor those processes by sending messages. Some of those processes could be the host operating system's processes. The implementation of a multi-user environment would be the most practical use of those processes. The "init" process and "getty" process in UNIX can be written by using HOPL process objects.

Another facility that HOPS could equip in the future is a read/write protection for each instance variable under a directory. Now, encapsulation is partially attained by allowing only those modifications of instance variables which are invoked from the

methods inside the current working directory which owns the instance variables. To strengthen the encapsulation, a read/write protection flag could be attached to each instance variable. This would enable the user to make a instance variable unchangeable by others. Furthermore, combined with the multi-user environment described above, that flag would protect the references and modifications to instance variables in different ways: by allowing the modifications for a certain group of users, or by allowing the references to instance variables for only the user who created them.

Bibliography

[1] Aho, A. V., R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] Budd, T., *A Little Smalltalk*, Addison-Wesley, 1987.

[3] Goldberg, A. and D. Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, 1983.

[4] Goldberg, A., "The influence of an Object-Oriented Language on the Programming Environment," *Proceedings of 1983 ACM Computer Science Conference*, (1983): 35-54.

[5] Ingalls, D. H., "The Smalltalk-76 Programming System: Design and Implementation," *Proceedings of the Fifth Principles of Programming Language Symposium*, Jan. 1978: 9-16.

[6] Kernighan, B. and D. Ritchie, *The C Programming Language Second Edition*, Prentice Hall, 1988.

[7] Kernighan, B. and J. Mashey, "The UNIX Programming Environment," *IEEE Computer*, 14:4 (1981): 25-34.

[8] Kernighan, B.and P. Plauger, *Software Tools in Pascal*, Addison-Wesley, 1981.

[9] Korn, David G. and Morris Bolsky, *Korn Shell Command & Programming Language*, Prentice-Hall, 1988.

[10] McCarthy, J., *LISP 1.5 Programmer's Manual, 2d ed.*, M.I.T. Press, Cambridge, Mass, 1965.

[11] Pratt, T. W., *Second edition PROGRAMMING LANGUAGES Design and Implementation*, Prentice-Hall, 1984.

[12] "Special Issue on Smalltalk," *BYTE*, Jun. 1981: 14-378.

[13] Stallman, R., "EMACS: The Extensible Customizable, Self-Documenting Display Editor," *proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Jun. 1981: 147-156.

[14] Steel, G. L., *Common LISP: The Language*, Digital Press, 1984.

[15] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.

[16] Tanenbaum, A. S., *COMPUTER NETWORKS second edition*, Prentice-Hall, 1988.

[17] Ungar, D.and R. B. Smith, "Self: The Power of Simplicity," *OOPSLA '87 Proceedings SIGPLAN Notices 22*, Dec. 1987: 227-242.

Appendix A
C Language Interface Example

This appendix shows a sample program which implements some of the methods for a List object. The following procedures are essential to build a simple object written in C language.

1. Prepare a source file including at least three things: a *magic* variable which holds the simple object's *magic* (a method dictionary), a *token struct* which is initialized by a sequence of pairs containing a token number and a message symbol string, and a *method function* whose interface is as follows:

```
MethodFunction(obj, token, arg)
OBJ obj, arg;
int token;
```

The method function is called when the simple object receives a message and that message is found in the method dictionary. The argument "obj" is the target object. The argument "token" is the one associated with the message symbol which the interpreter looks for in the method dictionary. The token number serves as a reference telling which program should be executed. The argument "arg" is the Environment object (as described in "An Environment object") which was created for this particular method function.

2. A method function will receive some special tokens:

-99: When HOPS terminates, -99 is passed as a token. A method function will perform procedures to terminate the simple object. For example, the line editor objet includes in its -99 program the procedures for deleting all the *scratch* files.

-1: If this is passed as a token, a method function must return a String object as the simple object's class name. The system automatically associates this -1 token with the message symbol "class", which does not have to be defined in the token struct.

0: When the system installs a simple object, its method function is called with a token of 0's. The program corresponding to the token 0's must be as follows:

```
magic = InstallObject(tokenStruct,
                      MethodFunction, 0);
```

By interpreting the token struct, the function "InstallObject" creates a method dictionary and returns it as the result. The returned value can be used for the class identification. The second argument of "InstallObject" must be the address of the method function, and the third argument must always be zero.

3. Insert the following statement into the program "InitMethods" defined in "installer.c".

```
MethodFunction(0,0,0);
```

Even if many method functions have already been called in "InitMethods", the order of those calls has no meanings.

4. Insert the following statement into the program "Epilogue" defined in "interpreter.c", only if the method function includes the -99 program.

```
MethodFunction(0,-99,0);
```

The following sample program shows how to construct a program for a simple object.

```
#include <stdio.h>
#include "object.h"          /* The type OBJ is defined */

extern OBJ printLit,         /* the String object for "print" */
       evalLit,              /* the String object for "eval" */
       valueLit;             /* the String object for "value" */

extern int returnFlag;

                             /* token number definition */
enum {L_NEW=1, L_APPEND, L_AT, L_PR, L_DO,
L_ATPUT, L_LEN};
                             /* token struct */
static struct {
     short token;
     char *str;
} list_m[] = {
{L_NEW, "new"}, {L_APPEND, "append"},
{L_AT, "at"}, {L_PR, "print"},
{L_DO, "do"}, {L_ATPUT, "atPut"},
{L_LEN, "len"},
{0,NULL} };                  /* end mark */
                             /* magic */
short list_magic;

CreateList()
{
     OBJ obj;
/* "CreateObject" allows the creation of a new object. The first argument must be a
magic, and the second must be the initial word size of the new object. */
     obj = CreateObject(list_magic, 0);
     return obj;
}
IsObjList(obj)
OBJ obj;
{
/* "ObjectMagic" is a macro defined in "object.h". */
     return ObjectMagic(obj) == list_magic;
}
```

```
                                   /* method function */
ListMethods(obj, token, arg)
OBJ obj, arg;
int token;
{
      OBJ x, y, z, nl;
      int i, l;
      switch(token)
      {
      case -1:
                                   /* "CreateLiteral" creates a String object*/
            return CreateLiteral("List");
      case 0: /* installer */
            list_magic = InstallObject(list_m, ListMethods, 0);
            break;
      case L_NEW:
      /* "CountArgN" tells the number of arguments for the current message */
            if ( CountArgN(arg) != 0 ) {
                  l = CountArgN(arg);
                  nl = CreateList();
                  for ( i=0;i<l;i++ ) {
      /* "GetArgN" retrieves one argument */
      /* "AppendSlot" appends a slot */
                        AppendSlot(nl, GetArgN(arg, i));
                  }
                  return nl;
            }
            return CreateList();
            break;
       case L_APPEND:
            x = GetArgN(arg, 0);
            if ( x == ILLEGAL )  x = NIL;
            if ( obj == x )
                  return ILLEGAL;
            AppendSlot(obj, x);
            return obj;
      case  L_LEN:
      /* "ObjectLength" is a macro defined in "object.h" */
            x = ObjectLength(obj);
      /* "CreateInteger" creates an Integer object */
            return CreateInteger(x);
       case L_ATPUT:
      /* "PutobjectSlot" is one of the primitives.  It puts an object into a slot of
an object.  The reference counts method works inside. */
      /* "ObjToInteger" converts an Integer object to an integer representation in C
language. */
            PutObjectSlot(obj,
                  ObjToInteger(GetArgN(arg, 0)),
                  GetArgN(arg, 1));
            return obj;
       case L_AT:
      /* "GetObjectSlot" is another primitive.  It gets an object bound to a slot of
an object. */
            GetObjectSlot(obj,
                  ObjToInteger(GetArgN(arg, 0)), &x);
            return x;
      case  L_PR:
             PrintList(obj, arg);      /* defined later */
```

```
            return obj;
      case  L_DO:
            return DoList(obj, arg);        /* defined later */
      default:
            return ILLEGAL;
      }
}
DoList(obj, arg)
OBJ obj, arg;
{
      int i, l;
      OBJ x, nobj, blk;
      blk = GetArgN(arg, 0);
      l = ObjectLength(obj);
      x = NIL;
      for (i=0;i<l;i++) {
            GetObjectSlot(obj, i, &x);
/* "SendMsgWithArg" sends a message with arguments.  The first argument of
"SendMsgWithArg" is a target object, the second is a message symbol, the third is the
argument number, and the fourth and subsequent arguments specify the arguments for the
message being sent. */
            x = SendMsgWithArg(blk, valueLit, 1, x);
            if ( returnFlag ) break;
      }
      return x;
}
static PrintList(obj, arg)
OBJ obj, arg;
{
      int i, l;
      OBJ x;
      l = ObjectLength(obj);
/* "Wprintf" is the same as "printf", but it prints out to the TTY controlled by HOPS.
*/
      wprintf("List(");
      for (i=0;i<l;i++) {
            GetObjectSlot(obj, i, &x);
            if ( i != 0 ) wprintf(",");
/* "SendMsg" sends a message with no arguments. */
            SendMsg(x, printLit, arg);
      }
      wprintf(")");
}
```

Appendix B
Setting Up HOPS under UNIX

First of all, the empty UNIX directory "HOSTFILE" must be created in the directory where the user starts HOPS. This is used to pool the files which are linked to File objects created in HOPS.

When HOPS is started, it executes the commands included in the special file ".hopsrc" under the UNIX current working directory, before entering the read-eval-write loop. Normally, those commands create or mount HOPS directories, print out titles, and set up the prompt string. The following shows a sample program for ".hopsrc".

```
/*-------------------------------------------------------*
    ".hopsrc"
 *-------------------------------------------------------*/

system "rm -f HOSTFILE/*";    // clear all the linked files

!.curses init;           /* curses mode ON (if the curses is
not supported, then delete this line) */

cd !.sh;             // common commands setup
.Console=!.con;
/*-------------------------------------------------------*
     'usr' directory mount
 *-------------------------------------------------------*/
cd !;
.usr=new;
cd !.usr;
mount "HOPSUSR";
mount "BIN";
mount "HOPE";
! inherit !.dict;        // inheritance
! inherit !.usr.bin;          // inheritance

cd !.usr.bin;                // .usr.bin compile
.SETUP run;
!.usr.hope setup;            // HOPE compile
cd !;
.".hopsrc"=File:new ".hopsrc";
/*-------------------------------------------------------*
     TITLE
 *-------------------------------------------------------*/
"\tHops: Hierarchical Object-oriented Programming System\n"
     print;
"\t\t\tVersion 1.00, Oct. 2, 1989\n" print;
.PROMPT1="HOPS> ";           //the prompt string set up
```

This ".hopsrc" requires that there be three directories under the current working directory: HOPSUSR, BIN, and HOPE. These directories, having sample programs and useful commands, are provided together with the HOPS load module. Among them, at the minimum BIN must be mounted, so that "setup" and "fork" (defined under BIN) are available in HOPS.

Appendix C
HOPS User's Manual (Message Descriptions)

The load module name for HOPS is "hops". To start HOPS, type "hops" in the UNIX directory, which includes some directories and files described in Appendix B. If the standard ".hopsrc" is used, you will see the following.

```
     Hops: Hierarchical Object-oriented Programming System
               Version 1.00, Oct. 2, 1989
HOPS>
```

The initial current working directory is the root directory "!". If you type in "l", then you will see the content of the root directory as follows:

```
HOPS> l
.!CurrentEnv      .PROMPT1          .con              .ed
.!CurrentStatem   .PROMPT2          .curses           .sh
..hopsrc          .PrevResult       .dict             .usr
!.sh
!.dict
!.usr.bin
TRUE
HOPS>
```

To terminate the HOPS session, type in "quit". The HOPS process quits.

The following are message descriptions for each object built in HOPS. Each description provides sample programs, which can be typed in from a TTY. The objects in the following manual are listed in alphabetical order.

The sample programs use the following notations.

1. Comments: They start with double slash "//" and continue to the end of the line.

2. Evaluated values: They are written to the right of the mark "»".

3. Printed values: Although the HOPS responses to the typed-in commands are frequently omitted, those responses are underlined if they are not omitted.

4. Class names: All the class names in HOPS start with a capital letter. This notation comes from Smalltalk-80. Although it is not mandatory in HOPS, this is sometimes necessary for clarification of the difference between UNIX resources and HOPS resources, such as a *directory* in UNIX and a *Directory* object in HOPS.

Assign An Assign object is used to accommodate the translated code for an assignment statement. There are two slots inside an Assign object: r-val and l-val. The l-val slot holds the left-hand-side expression in an assignment statement and the r-val slot holds the right-hand-side expression in an assignment statement. The object grabbed by the l-val slot must be one of the expressions for an instance variable, a parameter, and a temporary variable.

Message Symbols

class	Returns a String object for "Assign".
eval	Executes the assignment operation. The object grabbed by the r-val slot is evaluated (by sending the "eval" message), then it is assigned to the variable designated by the l-val object.
new	Creates an empty Assignment object.
print	Sends "print" to the l-val object, prints out " = ", and sends "print" to the r-val object.

Examples

```
.a:=.x=1
.a                                    » Assign object .x=1
a   // evaluation of the Assign object
.x                                    » 1
.a eval   // evaluation of the Assign object again
.a class                              » "Assign"
```

Block

A Block object is a special evaluatable object. It is a constant throughout the evaluation by the "eval" message, but it responds to the "value" message, evaluating the statements inside the Block object.

The "value" message can be accompanied by arguments, if the Block object includes parameter declarations. In addition, the Block object has a lexical scope, which means that it maintains the environment where the Block object appears.

A Block object has tree slots: one for a Method list (translated codes of statements), one for a target object, and one for an Environment object. The second and third slots are initialized by the message "create".

Message Symbols

class	Returns a String object for "Block".
create	Creates a new Block object whose first slot is a Method list copied from the original Block object, whose second slot becomes the current working directory, and whose third slot becomes the current Environment object. This message is automatically inserted by the HOPL parser when a Block object is parsed.
getlist	Returns the first slot object: a Method list.
new	Creates an empty Block object.
print	Prints out "[\n", sends "print" to the first slot object, and prints out "]\n".
self	Returns the second slot object: a target object.
value	Evaluates a Block object by sending "eval" to the first slot object (a Method list object).

Examples

```
.a=var x;{[var y;$x+$y];}
// the message "a" returns a Block object
.block=(a 12)
// The Block object maintains the environment for "x"
// "(a 12)" initialize "x" to 12.
.block value 3                    »15
.block value 1                    »13
```

Character (SystemObject)

A Character object is one of the constant objects which are called SystemObjects. SystemObjects also include the other constant objects: TRUE, NIL, FALSE, SELF, and SUPER.

A character constant is enclosed in single quotes. It supports the same escape sequence as C language.

Message Symbols

class	Returns a String object for "SystemObject".
int	Returns an Integer object coerced by the target Character object.
isalnum	Returns TRUE if the target Character object is alphabetic or a digit. Otherwise, it returns NIL.
isalpha	Returns TRUE if the target Character object is alphabetic. Otherwise, it returns NIL.
iscntrl	Returns TRUE if the target Character object is a CONTROL character. Otherwise, it returns NIl.
isdigit	Returns TRUE if the target Character object is a digit. Otherwise, it returns NIl.
islower	Returns TRUE if the target Character object is a lower case letter. Otherwise, it returns NIL.
isprint	Returns TRUE if the target Character object is printable. Otherwise, it returns NIl.
ispunct	Returns TRUE if the target Character object is punctuation. Otherwise, it returns NIL.
isupper	Returns TRUE if the target Character object is a capital letter. Otherwise, it returns NIL.
print	Prints the target Character object. If it is not visible, then the escape sequence is used to represent it.
tolower	Returns a lower case letter corresponding to the target Character object.
toupper	Returns a capital letter corresponding to the target Character object.

Examples

```
'a' toupper                    »'A'
'B' lower                      »'b'
'1' isdigit                    »TRUE
```

Cmethod

A Cmethod object is evaluatable. It includes an address to a method function and a token number. When it is evaluated, that method function is called with the token number. In that sense, a Cmethod object is a representation of a C language function. Normally, the method dictionary for a simple object initially contains the Cmethod object corresponding to a message symbol.

Message Symbols

class	Returns a String object for "Cmethod"
eval	Calls a method function according to its interface. The address of that method function is included in the Cmethod object. The second argument for the method function is also included in the Cmethod object as a token number.
new	Creates an empty Cmethod object.
print	Prints out a Cmethod object in the following format. ex. Cmethod(0x88393,3) : 0x88393 is the address to a method function. 3 is the token number.

Examples

```
Directory:lookup "."                »Cmethod(0x...,..)
// The method dictionary for a Directory object has a
Cmethod object corresponding to the message symbol
".".
```

Control

A Control object internally represents an if-statement, a while-statement, a for-statement, and a return-statement. It is evaluatable. The first slot inside a Control object holds a value which designates which statement the Control object represents out of four statements. Depending on that value, the remaining slots are used differently.

Message Symbols

class	Returns a Symbol object for "Control".
eval	In the case of an if-statement: first, the conditional expression is evaluated. If the result is NIL, then the else-statement is evaluated. Otherwise, the then-statement is evaluated. In the case of a while-statement: while the conditional expression is evaluated as a non-NIL value, the statement continues to be evaluated. In the case of a for-statement: first, the init-statement is evaluated. Then, while the conditional expression is evaluated as a non-NIL value, the statement and the repeated statement continue to be evaluated. In the case of a return-statement, a special flag gets set to 1's. This flag is checked for everywhere a loop evokes; such as at the while-statement evaluation and at the for-statement evaluation. At that check point, if the flag is on, then the loop is exited. This flag is cleared at the point where SendMsg terminates.
print	Prints the control statement image.

Examples

```
.a:=if (.x==1)"one"else if(.x==2)"two"else"else"
// a Control object for an if-statement
.x =1
.a eval                                  »"one"
a                                        »"one"
.x=3
a                                        »"else"
.b:=while (.x<5) {.x=.x+1;}
// a Control object for a while-statement
b
.x                                       »5
var i;for($i=0;$i<10;$i=$i+1) $i print;
0123456789
```

Dictionary

A Dictionary object is fundamentally important in HOPS. Every simple object points to a Dictionary object as its method dictionary (*magic*).

The structure of a Dictionary object contains an even number of slots, whose even-numbered slots specify key objects, and whose odd-numbered slots specify value objects associated with its key objects.

Message Symbols

add	Adds the second argument as associated with the first argument key (which must be a String object) to the Dictionary object.
class	Returns a String object for "Dictionary".
lookup	Looks up the value which has the given argument as a key. If there is no such value, then it returns ILLEGAL.
new	Is the trickiest message. Since every method dictionary is a Dictionary object, this method is invoked when "new" is sent to a method dictionary. However, this method checks whether or not a target object is the method dictionary for a Dictionary object. If it is, then a new empty Dictionary object is created and returned. If it is not, then this method evaluates the value object associated with "new" in that target object. Eventually, this method creates a simple object which has that target object as a method dictionary.
print	Prints the target object as follows: ex. Dictionary(a b c) The Dictionary object has "a", "b", and "c" as its keys.

Examples

```
var x;Integer:add "**",($x=var a;{if($a==0)1;else
SELF*(SELF**($a-1));}) // must be typed in one line

//This adds a method for "**" (power) to the
// method dictionary for an Integer object.

2**3                                    »8
3**3                                    »27
```

Directory

A Directory object is another name for a compound object. While a simple object cannot inherit methods from other objects, a compound object can.

Since most Directory objects are hierarchically arranged, the user can move through the tree structure, using the command "cd". "Cd" changes the current working directory, which is regarded as SELF when a message is sent without any specified target object.

Message Symbols

^	Looks up the value which is bound to "!pa" in the directory.
.	Looks up the value which is bound to the given argument (a String object).

add	Adds the second argument as a value which is associated with the first argument as a key. The following two statements are equivalent.
	
	`add "x",12`
	`.x=12`

class	Returns a String object for "Directory".
className	Returns a value which is currently bound to ".className" if there is no argument. If there is an argument, the argument is assigned to ".className".
del	Deletes the instance variable specified by the argument (a String object).
do	Has to have a Block object as its argument. That Block object receives two arguments: an instance variable name (a String object) and an instance variable value. Inside this "do" method, the Block object is repeatedly evaluated for each instance variable.
inherit	Appends the given argument to the inheritance list.
inheritList	Returns the current inheritance list if no argument is given. If it is given, then this "inheritList" method takes in that argument as a new inheritance list.
l	Prints out all the instance variable names in a columned format and prints out all the inherited objects.
li	Prints out all the inherited objects.

ll	Prints out all the instance variables with the class names of their values.
ls	Returns a list (a List object) of the instance variable names.
mount	Mounts a UNIX directory to HOPS. The argument must be a String object which designates a path name for the UNIX directory. If there is a file in the UNIX directory, it is linked to a File object in HOPS. If there is another directory in the UNIX directory, the corresponding Directory object is created in HOPS. These are recursively repeated.
new	Creates an empty Directory object.
print	Prints out the Directory object as follows: ex. Directory(project) The parenthesized name comes from the value bound to "!.me".
pwd	Prints out the path name starting with "!".
stat	Reports the status of the object block queue use.

Examples

```
pwd
!.project  // the current working directory
.aDir=new  // a new directory
cd .aDir
pwd
!.project.aDir
.x=1
^                                    »Directory(project)
.x                                   »1
del "x"
.x                                   »ILLEGAL
```

Environment

An Environment object is an activation record for each call of SendMsg. In an Environment object, two slots (caller and callee) form bi-directional links, simulating a stack.

Message Symbols

class	Returns a String object for "Environment".
print	Prints all the Environment object which are linked from the current Environment object by the caller slot.

Examples

```
// In a debug mode, a view of the current stack
// can be seen (by using  the command 'i'). That view
// is printed by sending "print" to the current
// Environment object.
```

File

A File object contains a text, which can be viewed by sending "type" to the File object. A File object can be linked to a specified UNIX file when it is created, so that all the modification on that File object can be permanently valid.

Message Symbols

!kill	Deletes the corresponding UNIX file pooled in "HOSTFILE/". *The user is not allowed to send this message.* The message "!kill" is called by HOPS only when the reference counter of the File object becomes zero.
+	Appends the argument to the end of the File object. The argument must be a String object.
cat	Returns a new File object, which concatenates the target File object and the File object given as the argument.
charDo	Has to have a Block object as its argument. The Block object receives one argument: a character from the File object. This method continues to evaluate the Block object, passing each character in each argument inside the File object.
class	Returns a String object for "File".
clear	Clears the content of the File object. This method is destructive.

cp	Returns a new File object, whose contents are copied from the target object.
getm	Returns an executable object as the result of compilation of the target File object.
hc	Performs the same thing as "getm" does.
id	Returns a String object for the pooled UNIX file name corresponding to the target File object.
lineDo	Has to have a Block object as its argument. The Block object is given one argument: a line (a String object) from the target File object. This method continues to evaluate the Block object, passing each line of the target File object.
monDo	Has to have a Block object as its argument. This method performs the Block object statements and all the things which the Block object prints out are appended to the target File object. While the Block object is running, those appended characters do not appear on a TTY screen.
new	Returns a new, empty File object if no argument is passed. If there is an argument, it is regarded as a String object for a UNIX file path name, which is linked to the new File object.
print	Prints "File".
run	Requires that the content of the target File object be a statement list. Each statement in the list is sequentially executed while the current working directory remains unchanged.
teeDo	Performs the same thing as "monDo" does, except that this method allows for printing to a TTY.
trans	Requires that the content of the target File object be a method-definition list. Each definition is interpreted and, finally, all the methods defined in the target File object are created under the current working directory.
type	Prints out the content of the target File object.
vi	Enables the user to edit the target File object using the UNIX *vi*.
wc	Returns a list containing three integers: a line count, a word count, and a character count for the target File object.

Examples

```
.aFile=File:new
.aFile append "hello\n"
.aFile type
hello
pwd
!.project
.aFile monDo[pwd;]
.aFile type
hello
!.project
```

Float

A Float object acts as the double floating point number in C language. The form of a positive Float constant object starts with a positive integer or an optional zero, which is immediately followed by a period (.), an optional sequence of digits, and an optional exponential expression. The exponent expression consists of a letter 'E' or 'e', an optional sign symbol '+' or '-', and a positive integer.

Message Symbols

*	Returns the product of the target object and the argument. The argument must be a Float object.
+	Returns the sum of the target object and the argument. The argument must be a Float object.
/	Returns the result of the target object divided by the argument. The argument must be a Float object.
<	Returns TRUE if the target object is less than the argument. Otherwise, it returns NIL.
<=	Returns TRUE if the target object is less than or equal to the argument. Otherwise, it returns NIL.
<>	Returns TRUE if the target object is not equal to the argument. Otherwise, it returns NIL.
>	Returns TRUE if the target object is greater than the argument. Otherwise, it returns NIL.
>=	Returns TRUE if the target object is greater than or equal to the argument. Otherwise, it returns NIL.
@	Returns a Point object, whose x value is the target object, and whose y value is the argument.
abs	Returns the absolute value for the target object.

ceil	Returns the smallest integer not less than the target object.
class	Returns a String object for "Float".
cos	Returns cosine of the target object.
exp	Returns the result of the exponential function $e^{\text{target object}}$.
int	Coerces the target object as an Integer object.
log	Returns the result of the natural logarithm.
log10	Returns the result of the base 10 logarithm.
neg	Returns (-1.0)*(the target object).
new	Returns a new Float object for 0.0.
pow	Returns the target object raised to the power of the argument.
print	Prints out the Float object using a format "%g" in "printf" notation.
sin	Returns the sine of the target object.
sqrt	Returns the square root of the target object.
tan	Returns the tangent of the target object.
–	Returns the value of the target object minus the argument.

Examples

```
1.2+1.3                          »2.5
.a=34.198
.a neg                           »-34.198
```

Integer An Integer object represents a long integer in C language (four-byte representation).

Message Symbols

~	Returns the target object's complement.
%	Returns the remainder when the target object is divided by the argument.
&	Returns the result of bitwise-and operation between the target object and the argument.
*	Returns the product of the target object and the argument.
+	Returns the sum of the target object and the argument.
/	Returns the result of the target object divided by the argument.
<	Returns TRUE if the target object is less than the argument. Otherwise, it returns NIL.
<<	Returns the bitwise-left-shifted integer of the target object by the argument.
<=	Returns TRUE if the target object is less than or equal to the argument. Otherwise, it returns NIL.
<>	Returns TRUE if the target object is not equal to the argument. Otherwise, it returns NIL.
==	Returns TRUE if the target object is equal to the argument. Otherwise, it returns NIL.
>	Returns TRUE if the target object is greater than the argument. Otherwise, it returns NIL.
>=	Returns TRUE if the target object is greater than or equal to the argument. Otherwise, it returns NIL.
>>	Returns the bitwise-right-shifted integer of the target object by the argument.
@	Returns a Point object, whose x value is the target object, and whose y value is the argument.
char	Returns a Character object, regarding the target Integer object as ASCII.
class	Returns a String object for "Integer".

float	Returns a Float object coerced from the target object.
neg	Returns (-1)*(the target object).
new	Returns an Integer object zero.
print	Prints out the target Integer object.
timesRepeat	Requires that the argument be a Block object. The Block object is repeatedly evaluated just N times, where N is the target object.
to	Returns an Interval object, whose start number is the target object, and whose end number is the argument. The step number for that Interval object is 1 if the target object is less than the argument. Otherwise, it is -1.
toBy	Returns an Interval object as the message "to" does, except that this method has a second argument, which specifies the step number.
xor	Returns the bitwise-exclusive-or operation of the target object and the argument.
|	Returns the bitwise-inclusive-or operation of the target object and the argument.
–	Returns the value of the target object minus the argument.

Examples

```
1+2                                    »3
.a=3-4                                 »-1
.a neg                                 »1
3 timesRepeat["hello\n" print;]
hello
hello
hello
1+2*3                                  »(1+2)*3»9
// The parser interprets the expression
// from left to right.
```

Interval

An Interval object has three slots in it: a start number, a end number, and a step number. Normally, this object is created by sending "to" or "toBy" to an Integer object (see Integer). The only meaningful message to this object is "do", which performs iteration using the argument Block object.

Message Symbols

class	Returns a String object for "Interval".
do	Requires that its argument be a Block object. The Block object receives an Integer object as its argument. This method continues to evaluate the Block object as the argument Integer object (starting with the start number) is increased by the step number, until the Integer object becomes greater than the end number.
print	Prints out the target Interval object.

Examples

```
1 to 4:do[$1 print;]
1234
3 to -1:do[$1 print;]
3210-1
2 toBy 7,2:do[$1 print;]
246
```

List

A List object is the most basic object, because its messages reflect a set of basic operations on an object in HOPS. A List object is simply a sequence of objects.

Message Symbols

+	Returns the concatenation of the target object and the argument. The argument object must be a List object.
append	Appends the argument to the end of the target List object.
at	Gets the element in the List object which is indexed by the argument Integer object.
atPut	Requires two arguments. This method puts the second argument object at the position indexed by the first argument Integer object.
class	Returns a String object for "List".

del	Deletes the element indexed by the argument Integer object.
do	Requires one Block object argument. The Block object receives an element from the target List object. The Block is repeatedly evaluated, receiving each element of the target List.
insert	Requires two arguments. This method inserts the first argument into the position indexed by the second argument Integer object.
len	Returns the element number.
lookup	Looks for the argument in the target List object. If not found, ILLEGAL is returned.
new	Creates a new empty List object if no arguments are provided. If there are arguments, those become the initial elements for the new List object.
print	Prints out the List object together with its content as follows: ex. List(1,2,3)
sort	Returns the sorted List object. It sends the message "<" to each element in order to decide the order. The *quick sort* method is employed inside.

Examples

```
.aList=List:new "b","c"          »List("b","c")
.aList atPut 0,"a"               »List("a","c")
.aList insert 0,"b"              »List("b","a","c")
.aList sort                      »List("a","b","c")
```

Method

A Method object is a basic evaluatable object. When an object receives a message, that action is interpreted as a Method object. A Method object includes a slot for a target object, a slot for a message symbol, and a slot for a list of arguments.

When it is evaluated, a target object and all the arguments are evaluated, then SendMsg is called.

Message Symbols

class	Returns a String object for "Method".

eval	First, evaluates a target object in a slot, first. A new Environment object is then created, so that it accommodates evaluated arguments. The evaluated target object, the message symbol, and the newly created Environment object are passed to SendMsg. The result of SendMsg is bound to one of the slots in the old Environment object and is returned to the caller.
new	Creates an new empty Method object.
print	Prints out the Method object.

Examples

```
.aMethod:=1+2  // "1" receives "+" with "2"
aMethod                              »3
```

Mtdlist

A Mtdlist object (called a Method list object in the thesis sections) is a sequence of Method objects. It is frequently used because the typed-in commands should be a statement-list which is translated into a Mtdlist object. In addition, a compound statement is translated into a Mtdlist object, too.

A Mtdlist object includes a slot for a sequence of evaluatable objects, and a slot for a File object, which contains a source program. If a File object receives the "tarns" message, methods are created under the current working directory. Those methods are implemented by Mtdlist objects whose File object slot is set to that translated File object.

Message Symbols

class	Returns a String object for "Mtdlist".
doc	Returns a File object which is bound to a source file slot. It indicates that File object is used to create the target Mtdlist object.
eval	Evaluates each evaluatable object sequentially.
new	Creates an new empty Mtdlist objet.
print	Prints out the target Mtdlist object.
trans	Sends "trans" to the File object which is bound to a slot of the target Mtdlist object. If there is no such File object, then an error message is printed.
type	Sends "type" to the File object which is bound to a slot of the target Mtdlist object. If there is no such File object, then an error message is printed.

vi	Sends "vi" to the File object which is bound to a slot of the target Mtdlist object. If there is no such File object, then an error message is printed.

Examples

```
.aMtdlist:={.x=.x+.x;}
.x=1
aMtdlist
.x                                    »2
aMtdlist
.x                                    »4
.x="abc"
aMtdlist
.x                                    »"abcabc"
aMtdlist
.x                                    »"abcabcabcabc"
```

Point

A Point object is normally created by sending "@" to an Integer object or a Float object (see *Integer* and *Float*). Basically, it implements a two-dimensional coordinate.

Message Symbols

+	Returns a new Point object, whose x value is the sum of the target object's x and the argument object's x, and whose y value is the sum of the target object's y and the argument object's y.
class	Returns a String object for "Point".
new	Requires two arguments: x value and y value for a new Point object.
print	Prints <x value>@<y value>.
x	Returns the x value for the target Point object.
y	Returns the y value for the target Point object.
–	Returns a new Point object, whose x value is the difference between the target object's x and the argument object's x, and whose y value is the difference between the target object's y and the argument object's y.

Examples

```
1@3+(4@3)                          »5@6
1.2@4.5-(2.3@3.4)                  »-1.1@1.1
.aPoint=3@6
.aPoint x                          »3
.aPoint y                          »6
```

Pub

The Pub method dictionary collects a built-in inherited methods for every object (simple objects and compound objects). It mostly includes logical-operations.

Message Symbols

&&	Returns TRUE if both the target object and the argument are non-NIl values. Otherwise, it returns NIL.
<<>>	Returns TRUE if the magic of the target object and the magic of the argument are not equal. Otherwise, it returns NIL.
<>	Returns TRUE if the target object and the argument are not equal. Otherwise, it retums NIL.
==	Returns TRUE if the target object and the argument are not equal. Otherwise, it retums NIL.
===	Returns TRUE if the magic of the target object and the magic of the argument are equal. Otherwise, it returns NIL.
`	Returns a String object for the image which is printed when the target object receives "print".
copy	Retums a new object which is shallow-copied from the target object.
error	Prints an error message with the argument.
isDir	Returns TRUE if the target object is a Directory object. Otherwise, it retums NIL.
isExec	Returns TRUE if the target object is evaluatable. Otherwise, it returns NIL.
isInstanceOf	Returns TRUE if the magic of the target object is equal to the argument. Otherwise, it retums NIL.
isKill	Returns TRUE if the KILL flag of the target object is on. Otherwise, it returns NIL.

isMounted	Returns TRUE if the MOUNT flag of the target object is on. Otherwise, it returns NIL.
isNil	Returns TRUE if the target object is NIL. Otherwise, it returns NIL. It is equivalent to the message "== NIL".
IsPrim	Returns TRUE if the CG flag of the target object is on. Otherwise, it returns NIL.
not	Returns TRUE if the target object is NIL. Otherwise, it returns NIL.
notNil	Returns NIL if the target object is NIL. Otherwise, it returns TRUE. It is equivalent to the message "<> NIL".
size	Returns a word size which the target object occupies for its slots.
||	Returns NIL if both the target object and the argument are NIL. Otherwise, it returns TRUE.

Examples

```
NIL isNIL                          »TRUE
NIL notNIL                         »NIL
1 not                              »NIL
.x=1
.y="abc"
.x==1                              »TRUE
.y<>"abc"                          »NIL
(.x==1)&&(.y<>"abc")               »NIL
(.x==1)||(.y<>"abc")               »TRUE
.x isInstanceOf (Integer)          »TRUE
.y isInstanceOf (Float)            »NIL
```

String

Basically, a String object accommodates a practically infinite length of a character string. This takes advantage of the infinity in the object's basic structure. If the length of a character string is odd, then a null character is put at the end in order to allocate memory in words (16 bits). If you send "size", then you will see the word size of that string. If you send "len", then you will see the precise character size of that string.

Message Symbols

+	Returns a new String object which concatenates the target object and the argument.
<	Returns TRUE if the target object is less than the argument in ASCII order. Otherwise, it returns NIL.

<=	Returns TRUE if the target object is less than the argument in ASCII order or equal to the argument. Otherwise, it returns NIL.
<>	Returns TRUE if the target object is not equal to the argument. Otherwise, it returns NIL.
==	Returns TRUE if the target object is equal to the argument. Otherwise, it returns NIL.
>	Returns TRUE if the target object is greater than the argument in ASCII order. Otherwise, it returns NIL.
>=	Returns TRUE if the target object is greater than the argument in ASCII order or equal to the argument. Otherwise, it returns NIL.
append	Returns a new String object which appends the argument Character object to the end of the target object.
at	Returns a Character object indexed by the argument Integer object in the target String object. The index starts with zero.
charDo	Requires a Block object as its argument. The Block object receives a Character object from the target String object. The Block is repeatedly evaluated while it receives each Character object.
class	Returns a String object for "String".
format	Returns a String object created in a similar manner to the "sprintf" function in C language. The format is the target String object which may include conversion specifications. The conversion specification begins with '%', and ends with the conversion character 'O'. Between the two characters are an optional '-', an optional number specifying a minimum field width, and an optional period following a number called the precision. The arguments for "format" can be only Integer objects and String objects.
getm	Returns the translated code (a Mtdlist object) for the target String object.
in	Returns TRUE if the argument includes a pattern specified by the target object. Otherwise, it returns NIL.
len	Returns the character size of the target String object.

new	Returns a new empty String object.
print	Prints the target String object.
tokenlist	Searches the target String object for tokens delimited by characters from the argument String object, and returns a list (a List object) of those tokens.

Examples

```
"ab:cd ef" tokenlist " :"          »List("ab","cd","ef)
"abc" in "lkjasabcasjd"            »TRUE
"abc" len                          »3
"abc" size                         »2
.p=".a=1;" getm
p // evaluation
.a                                 »1
"abc"+"def"                        »"abcdef"
"abc" append 'd'                   »"abcd"
"abc" charDo[$1 toupper:print;]
ABC
":%5.3O:%5.3O:" format "hello", 123:print
:  hel:  123:
```

QuotedAssign

A QuotedAssign object is a translated code for a quoted assignment statement, which uses ":=" rather than "=" in an assignment statement. In the evaluation of a QuotedAssign object, the right-hand-side of the operator ":=" is not evaluated and then it is assigned to the variable specified by the left-hand-side expression. This object has two slots: l-val and r-val.

Message Symbols

class	Returns a String object for "QuotedAssign".
eval	Assigns the value of the r-val slot to the variable specified by the expression bound to the l-val slot. The value of the r-val slot is not evaluated.
new	Returns a new empty QuotedAssign object.
print	Prints the target QuotedAssign object in a format.

Examples

```
.a:=.b:=.x=5
//.a is the QuotedAssign object ".b:=.x=5"
a // the evaluation of ".b:=.x=5"
//Now .b is an Assign object for ".x=5"
b // the evaluation of ".x=5"
.x                                    »5
```

The following are message descriptions for built-in directories.

Bin (!.usr.bin)

The following methods are defined in the directory "!.usr.bin". Since this directory is inherited by the root directory, every directory can invoke those methods.

The methods in the directory "!.usr.bin" are all written in HOPL. You can see the source codes by sending "type" to the methods.

Message Symbols

awk

Acts as the UNIX command *awk*. It requires two arguments: a File object and a Block object. The File object is checked using a program which is written in the Block object. The arguments to the Block object are a list (a List object) of a whole line, and the words included in that line. Those arguments can be accessed by using "barg", "bargc", and "bargv" (see Interpreter). The evaluation of the Block repeatedly continues for each line in the File object.

find

Acts as the UNIX command "find . -print". It prints all the instance variables which are hierarchically arranged from the current working directory. They are printed in a full path name.

fork

Creates a new clone object generated from a compound object. If you want to make an instance object of your own class, then send "fork" to the class (a Directory object). All the arguments for "fork" are passed to the message "init", which is supposed to initialize all the instance variables.

grep

Acts as the UNIX command *grep*. It takes one argument: a pattern string. "Grep" looks for File objects in the current working directory, or File objects included in Mtdlist objects as the source code. Each line in those File objects is examined whether or not it contains the pattern string. If it contains the string, then that line is reported together with the line number.

hope

Invokes the screen editor HOPE. If an argument is provided, HOPE regards it as a File object and reads its content into a scratch file before HOPE starts.

select	Takes a Block object as its argument. That Block object receives two arguments: an instance variable name and an instance variable value. The evaluation of the Block object continues for each instance variable in the current working directory. This method returns a list of instance variable names, for which the Block object returns TRUE (non-NIL).
setup	Compiles all the source files in the directory. The user must prepare the File object ".SETUP" or ".main". ".SETUP" includes procedures to compile all the source files in the directory. If the user prepares only ".main", all the method definitions should be included in ".main".

Examples

```
.aFile type
abc 123
def 456
abc 789
grep "def"
.aFile: line 2:def 456
1 match(es) found
awk .aFile, [$2+"\n" print;]
abc
def
abc
```

Console (!.con)

This directory includes all the methods for a TTY input / output.

Message Symbols

clear	Clears the TTY screen. It is effective only in the curses mode.
cursorX	Returns the cursor x position (column number). It is effective only in the curses mode.
cursorY	Returns the cursor y position (line number). It is effective only in the curses mode.
getc	Reads one character and returns it.
getm	Reads one line, parses it, and returns the translated code.
gets	Reads one line, and returns it.

move	Moves the cursor to the (x,y) position, where x is the first argument and y is the second argument.
nullDo	Requires a Block object as its argument. This method discards all the things which are printed during the evaluation of the Block object. That means nothing is printed on a TTY.
putc	Writes one character which is given as an argument.
puts	Writes one string which is given as an argument.

Examples

```
Console:nullDo["abc" print;]  // nothing printed
// "Console", which is defined in "!.sh", simply
// returns "!.con", so that the user can access
// the Console object easily rather than typing
// "!.con".
Console:putc 'a'
a
.p=Console:getm
.a=2 // typed in
p  // the evaluation of ".a=2"
.a                                    »2
```

Curses (!.curses)

This directory provides some functions of the UNIX *curses* library as methods. To make those methods effective, "init" should be sent to "!.curses". This makes the curses mode be on. The curses mode can be turned off by sending "end" to "!.curses".

Message Symbols

addch	Prints one character which is the argument.
addstr	Prints one string which is the argument.
beep	Beeps.
clrtobot	Clears the screen from the current cursor position to its bottom.
clrtoeol	Clears the screen from the current cursor position to the end of the current line.

cookedInp	Edits a command line. There is memory to store up to 10 previous command lines. Those lines can be accessed by typing the ESCAPE character followed by "k" or "j". The key "k" puts the previous line into the line edit buffer. The key "j" puts the recent line into the line edit buffer. After typing the ESCAPE character, the user can use "l", "h", "x", "i", "a", "A" and "0" as edit commands. Their meanings are the same as the UNIX *vi*. When you finish editing the line, type the NEW-LINE character. The method for "cookedInp" returns a String object for the edited line. This method is also implemented inside usual command input routine. If the curses mode is on, the user can go back to the previous command line and make use of it, when typing a command.
curMotion	Used in HOPE. Basically, this method moves around the screen on which the edited text is displayed. The effective commands are "l", "k", "j", "h", "G", "/", "?", "n", "N", "w", "i", "I", "a", "A", and "x". Their specification is the same as the UNIX *vi*. This method returns the character that was hit by the user and was not any of the above commands. Before using this method, the user should call "initLineDb".
curline	Returns the current line number which is being used in HOPE.
end	Terminates the curses mode.
getch	Gets one character and returns it.
getmaxxy	Returns a Point object whose x and y are the max width and the max height for the current using terminal.
getxy	Returns a Point object whose x and y are the column number and the row number of the cursor.
gotoLineCo	Requires two arguments: the line number and the column number. This method redisplays the edit screen for HOPE to relocate the current edit position to the given position.
init	Turns on the curses mode.
initLineDb	Initializes all the stuff necessary for the edit screen display. This is used in HOPE.
insch	Inserts a character which is given as its argument.

insertln	Inserts a line which is given as its argument.
move	Takes a Point object as its argument. This moves the cursor to the position designated by the Point object.
redisplay	Redisplay the edit screen for HOPE.
refresh	Updates the screen, flushing all the modification to the screen.
scrrng	Changes the scroll range. There are two arguments which specify the lower line number and the upper line number. The screen is scrolled only between those two lines.
standend	Turn off the standout mode.
standout	Turn on the standout mode, in which every character is printed in highlight.
status	Reports whether the curses mode is on or off. If it is on, then it returns TRUE. Otherwise, it returns NIL.

Interpreter (!.sh)

This directory is inherited by the root directory. That means every directory can share the methods defined in "!.sh".

Message Symbols

arg	Requires an Integer object as its argument. This method retrieves the argument indexed by the Integer object from the current environment.
argc	Returns the number of arguments in the current environment.
argv	Returns a list (a List object) of arguemtns in the current environment.
barg	Requires an Integer object as its argument. This method retrieves the block argument indexed by the Integer object from the current environment.
bargc	Returns the number of block arguments in the current environment.
bargv	Returns a list (a List object) of block arguments in the current environment.

cd	Changes the current working directory to a directory specified by the argument. It there is no argument, then it changes the current working directory to the root directory.
Console	Returns simply "!.con".
debug	Toggles the debug mode between on and off.
dumpOk	Returns the current mode for dump listing if no argument is given. If the mode is TRUE, then every evaluatable object is printed in details. If the mode is NIL, then every evaluatable object is printed in a short format. If you give an argument, that TRUE/NIL value is assigned to the current mode. This mode is useful to make it easy to see the debugger messages. If the mode is TRUE, then those messages might be too wordy to skim the executions.
endtime	Prints the time which has been spent since "starttime" (described below) was invoked. The format includes user CPU time, system CPU time, and the real time spent for the HOPS execution. This implementation was originated by Rochkind, Marc J., *Advanced UNIX programming*, Prentice-Hall, 1985.
env	Prints the current Environment objects.
evalloop	Is the main body for the interpreter cycle of read, evaluate, and write. This can be recursively called. To exit from this loop, send "quit".
evaluate	Requires two arguments: a target object and a HOPL expression. That expression is evaluated while keeping the current working directory being that target object.
postwrite_toggle	Toggles the postwrite mode between on and off. If it is on, then the "evalloop" cycle includes the writing process for the result. If it is off, the system keeps silent. The default is on.
quit	Terminates the current "evalloop" cycle.
sendmsg	Requires three arguments: a target object, a message symbol, and a list (a List object) of arguments. This method sends the message symbol with the list of arguments to the target object.

starttime	Starts to measure the time spent by the HOPS execution. See "endtime".
stat	Reports the status of the object block use.
unix	Enters a UNIX shell session.
verbose	Returns the current verbose mode if there is no argument. If there is an argument, it is assigned to the mode as a Boolean value. If the mode is on and SendMsg returns ILLEGAL as the result of evaluation, then the system automatically goes to the debug mode. If the mode is off, the system keeps silent.

Examples

```
sendmsg 1,"+",(List:new 2)          »3 (1+2)
.x=12
sendmsg SELF,".", (List:new "x")  »12
```

Line Editor (!.ed) This directory provides the line editor commands as methods. Those methods are used internally in HOPE. As usual, this line editor maintains the current line number. The command usually operates on the line pointed to by the current line number unless parameters specify the position for that command..

Message Symbols

&	Returns the number of lines included in the scratch file.
*	Returns the current line number. The line number starts with one. If there is no lines in the scratch file, it returns zero. If there is an argument, the user can change the current line number.
a	Appends the string (given as its argument) right after the current line.
clred	Clears the scratch file content. All the edited text is gone.
cp	Takes a Point object as its argument. This method copies a block (a sequence of lines) to the current line. The lower line number of the block is specified by the x-value of the Point object. The upper line number of the block is specified by the y-value of the Point object.

d	Takes a Point object as its argument. This method deletes a block (a sequence of lines). The lower line number of the block is specified by the x-value of the Point object. The upper line number of the block is specified by the y-value of the Point object.
getl	Returns a String object for the current line.
i	Inserts a String object (given as its argument) into the current line.
mv	Takes a Point object as its argument. This method moves a block (a sequence of lines) to the current line. The lower line number of the block is specified by the x-value of the Point object. The upper line number of the block is specified by the y-value of the Point object.
p	Prints all the lines which are being edited in the scratch file. If the line number is specified as its argument, only that line is printed.
r	Takes a File object as its argument. This inserts the content of the File object into the current line.
sw	Switches the scratch files. There two scratch files, which can be used for editing a file independently. Th current line number is preserved for each scratch file.
w	Returns a new File object whose content is the same as edited in the scratch file. If two arguments are given, then the user can specify from which line to which line the new File object should contain.

Examples

```
cd !.ed // suppose the scratch file is empty.
a "hello"
p
    1*hello // '*' means the current line.
a "hello2"
a "hello3"
p
    1:hello
    2:hello2
    3*hello3
cp 1@2
p
    1:hello
    2:hello2
    3*hello3
    4:hello
    5:hello2
.aFile=w // a File object is created.
.aFile type
hello
hello2
hello3
hello
hello2
```

Root Directory (!)

The root directory inherits "!.sh", "!.dict", and "!.usr.bin". When HOPS starts, the initial current working directory is the root directory.

Message Symbols

!CurrentStatement	Holds the translated code for the current input command.
PROMPT1	Holds the prompt string.
PrevResult	Holds the previous result.

Examples

```
HOPS> .PROMPT1="hello> "
hello>
```