THE SKI MANUAL

by

Gray Clossman

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT No. 110

THE SKI MANUAL

GRAY CLOSSMAN

JULY, 1981

This material is based upon work supported by the National Science Foundation under Grant MCS80-09201.

page 1. Title page. Contents by section name.

The SKI Manual

by Gray Clossman

This report is divided into the following sections:

A. INTRODUCTION

SKI is an implementation of the programming language Scheme [Steele and Sussman 78] and is based directly on the IUSCHEME implementation by Mitchell Wand [Wand 80a]. The Ski (see note on capitalization conventions in section B, "CONTENTS AND NAMING CONVENTIONS") language includes, in addition to the kernel forms of Scheme, the data abstraction constructs CLASSES and OBJECTS and kernel forms for expressing parallel computations. The SKI run-time system is an interactive program and consists of an editor, a compiler, and a "microcode-like" interpreter which executes compiled Ski programs and simulates parallel execution of parallel computations as in [Wand 80b]. SKI runs in conjunction with UCI-LISP [Bobrow et al] and uses the ILISP s-expression oriented editor, the ILISP break package, and a file editor. SKI has grown out of our work on the artificial intelligence project SEEK-WHENCE (SW) [Hofstadter et al 79] for which we are implementing a program architecture that uses data-driven parallel processes as in the HEARSAY II program [REDDY et al 76]. This report is a manual and tutorial for the SKI implementation of Scheme. We assume that the reader is, at the very least, familiar with some implementation of Lisp. We strongly recommend that the reader keep handy for reference:

1) documentation of the Scheme language, for example, [Steele and Sussman 78],

2) some documentation of ILISP, for example, [Bobrow et al] or [Meehan 79], and

3) "Continuation-Based Multiprocessing", [Wand 80b].

Anyone who wants to modify SKI should have all of the above and also the manual [Wand 80a] for SKI's predecessor, IUSCHEME.

Type "RU SKI[1,GAC]" to run SKI at SAIL. This text is SKI.MAN[1,GAC].

overview of SKI

SKI runs on top of the UCI-LISP system. SKI was implemented on the version of ILISP documented in [Bobrow et al] on a DEC-10 at Indiana University under the TOPS-10 operating system and at Stanford Artificial Intelligence Lab under the WAITS operating system. Questions, observed bugs, suggestions, extensions, or improvements are invited. Correspondance to:

Gray Clossman
Computer Science Dept.
101 Lindley Hall,
Indiana University
Bloomington, Indiana 47401

SKI resembles LISP programming systems in many ways. A programmer interactively writes, tests, and edits a system of functions to compose a Ski program. All of the capabilities of the underlying ILISP system are available to a SKI programmer, and any ILISP program may be run from SKI. We have avoided redefining Ilisp primitives when implementing SKI. The underlying Ilisp system is undisturbed. We hope that any program that runs in ILISP will run in SKI.

SKI consists of:

1) a READ-EVAL-PRINT LOOP which reads input from a terminal or a file and prints values to a terminal or file.

2) a COMPILER. Scheme expressions are compiled into an assembly language for a virtual stack machine. This assembly language is list-structured and is called the STACK LANGUAGE.

3) a STACK LANGUAGE INTERPRETER which executes the stack language. The interpreter is the virtual stack machine and is written in Ilisp.

4) an S-EXPRESSION-ORIENTED EDITOR. SKI uses the ILISP STRUCTURE EDITOR [Bobrow et al] [Meehan 79] to edit Scheme programs and file definitions.

5) a FILE EDITOR. A file is represented in SKI by a FILE DEFINITION: a list of the names of functions, procedures, data, and comments on a file. You edit a file by editing its file definition.

The file editor and extensions to the ILISP editor are separable from SKI and are appropriate for use in ILISP systems. The source language for the editor extensions, a versatile pretty-printer, and the file editor is on the file FILHAN.IL [1,GAC] at SAIL.

Debugging facilities for Ski programs include:

1) COMPILE-TIME ERROR CHECKING.

2) the ILISP BREAK PACKAGE [Bobrow et al] [Meehan 79]. When an error condition occurs, the SKI system enters a BREAK, i.e., the computation is frozen and control is thrown to the break package. In a break, a programmer may examine variable bindings and function definitions, edit them, and continue the computation after corrections are made.

3) the functions ERR, ERROR, and ERRSET. These functions let the SKI programmer detect and generate errors in order to direct the flow of control in a Ski program.

4) the functions TRACE and UNTRACE, compatible with the ILISP trace package, for tracing Scheme procedures.

background

In the summer of 1980, we began the implementation of the SEEK-WHENCE project. The architecture of the SW program requires a variable number of processes that run in parallel, communicate both by messages to each other and through a global data structure, can execute arbitrary expressions indivisibly, and can be scheduled by one or more schedulers which are also processes. We had to choose a programming language in which to implement SW. The language had to be easy to write, debug, and extend as needed, and had to have support software (file handling, editing, etc.) for convenient programming. For many reasons -- familiarity, availability, expressive power, extensibility -- Lisp seemed right. UCI-LISP was familiar to us, and provides outstanding editing and debugging facilities. We were aware that Mitchell Wand at Indiana University had implemented a dialect of Scheme, called IUSCHEME, which included the data abstraction constructs CLASSES and OBJECTS (similar to those of SIMULA [Dahl et al 70] and Smalltalk [Shoch]). Wand had written an operating system in IUSCHEME using the Scheme primitive CATCH for process saving [Wand 80b]. IUSCHEME had many of the qualities we were looking for. It had data abstraction; it was written in Lisp1.6 and so was modifiable and extensible; it supported parallelism using timing interrupts to simulate parallel execution of processes; it implemented Scheme's tail recursion; and it ran acceptably fast -- half as fast as interpreted Lisp1.6. We decided that IUSCHEME looked promising for SW. SKI, which very roughly stands for "Scheme at Indiana University", took shape in the fall of 1980.

B. CONTENTS AND NAMING CONVENTIONS.

capitalization conventions

This report describes the integration of two programs -- Mitchell Wand's implementation of Scheme and the UCI-LISP implementation of Lisp. It talks about several very similar languages -- Scheme, Ski, Lisp1.6, and Ilisp. The naming conventions used in this report will often be important. The name of a program or system will usually be written in all capitals (e.g., ILISP, SKI), while the name of a language supported by or interpreted by a program will be written with its first letter capitalized (e.g., Ilisp, Ski). Names of individual Ilisp or Scheme functions and variables will be printed uppercase (e.g., COND, LABELS, INVENTORY). Definition types -- named by the property names (e.g., EXPR, MACRO, DEF) under which they appear on the Ilisp property lists of identifiers -- will be printed lowercase (e.g., expr, macro, def). Section P of this manual is a glossary of terms. The naming conventions used for functions and variable names in the SKI source files are explained in section O, "THE SKI SOURCE FILES".

naming conventions in the file editor

Most of the functions implementing the file editor come in pairs of a macro and a subr (e.g., INVENTORY - INVENTORYX, GETFILE - GETFILELX). The macro of the pair usually provides the more convenient way to call the function from the top level. It typically will not require its arguments to be quoted, and may accept a variable number of arguments. (But see individual function specifications in section J, "EDITING A FILE".) The subrs of the pairs do evaluate their arguments (as subrs always do), and if you program your own extensions to the file editor, feel free to use them as subroutines. The subr of each pair is named by suffixing "X" or "LX" to the name of the macro. In general, the suffix "X" indicates a subr that takes single file names or variable names as arguments. For example, RENAMEFILEX takes two arguments, the names of two files. But the suffix "LX" indicates a subr that takes at least one list of things as an argument. For example, GETFILELX takes one argument, a list of file names.

Some global Ilisp variables used by the file editor are:

GRINPROPS — the list of property names of definition types that may be pretty-printed, edited, and stored on files.

DEFINERS — the list of definers -- functions that store definitions on property lists of identifiers -- and the property names that correspond to them.

TRACEDFNS — the list of names of functions that are currently traced.

SCREENSIZE — the number of printed lines that fit on the user's terminal screen. See explanation of TY in section J, "EDITING A FILE".

ERRVALEXP
ERRVALFLAG — special variables used by the file editor.

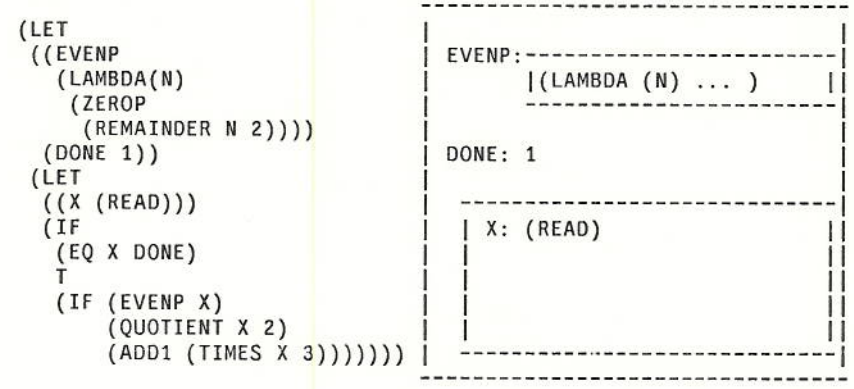
C. SHORT DESCRIPTION OF THE SCHEME LANGUAGE AND PARALLEL PROCESSES

Scheme is a lexically scoped dialect of Lisp. Scheme, Algol, and Pascal are all examples of lexically scoped languages. Each is a block-structured language where identifiers declared within a block are local to that block, i.e., references to an identifier are valid only in expressions written in the block in which that identifier is declared. Scheme uses lambda binding to declare identifiers. So the scope of an identifier -- the parts of a program in which references to that identifier have meaning -- can be determined by examining the source-language text of the program. The identifier's scope does not depend on the program's run-time flow of control. Flow of control in a given program will change from one execution of the program to the next because it is data-dependent, but the source-language text of that program will not change. Roughly speaking, then, a lexically scoped language can be compiled. Based solely on the source text of a Scheme program, a compiler can determine whether an identifier reference is valid, and if so, exactly where the value associated with that identifier will be at run time. Since this determination can be made on the basis of a static form of a program -- its source text -- "statically scoped" is synonymous with "lexically scoped". Identifiers in a lexically scoped language are said to be "statically bound" and are called "static identifiers" or "static variables". Identifiers in a dynamically scoped language are said to be "fluidly bound" and are called "fluid identifiers" or "fluid variables". SKI provides both static and fluid environments, but in Scheme programs, any identifier that is not marked "fluid", e.g., (FLUID X), is treated as a static identifier. In the following example Scheme program, the Scheme magic word "LET" is an alternate syntax for lambda binding. The LET expression here binds the variables EVENP and DONE. In general:

"(LET ((<var1> <exp1>) ... (<varn> <expn>)) <body>)"

is equivalent to

"((LAMBDA (<var1> ... <varn>) <body>) <exp1> ... <expn>)".



The Scheme program above reads a number, x. The program returns the value T if x is equal to DONE; if x is even, it returns x divided by two; but if x is odd, it multiplies x by three and adds one. The program contains three blocks: the outermost block where the identifiers EVENP and DONE are bound, a block in which N is bound, and a block in which X is bound. The identifiers EVENP and DONE are meaningful everywhere in the program, because the block in which they are bound contains every other block in the program. Note the uses of EVENP and DONE in the bottom inner block. The identifier N is not within the lexical scope of the bottom LET expression and so cannot be referenced in the bottom inner block. Similarly, X cannot be mentioned in the top inner block. The following equivalent Lisp program DOES refer to X in its function EVENP.

```
((LAMBDA(EVENP DONE)
  ((LAMBDA(X)
    (COND ((EQ X DONE) T)
          ((EVENP) (QUOTIENT X 2))
          (T (ADD1 (TIMES X 3)))))
   (READ)))
 (QUOTE (LAMBDA NIL (ZEROP (REMAINDER X 2))))
 1))
```

The reference to X in the Lisp program depends on the run-time flow of control of the program. The Lisp function EVENP here MUST be called from some other expression in which X is bound to a number or an error will result. The validity of the reference to X in the Lisp program cannot be determined from the text of the program. Because of this, there are some properties of the program that cannot be proved, and a compiler for Lisp cannot replace the identifier X in this Lisp expression by a specification of the location where its value will be found at run time. References of this type are illegal in Scheme. The SKI compiler replaces each identifier in a Scheme program by a pair of small integers that code for the location of its value at run time.

A Scheme identifier may have as its value EITHER a datum (constant, atom, or list) OR a procedure. (PROCEDURE is a data type in Scheme, like ATOM, LIST, NUMBER, etc. The predicate PROCP returns T if its argument has a procedural value, just as NUMBERP returns T if its argument has a numeric value. Procedures may be passed as arguments to other procedures and returned as the values of other procedures.)

A VALUE is either a
DATUM, which is a constant, an atom, a list, or a string;
or a
PROCEDURE, which is a closure, a continuation, or an object.

In Scheme, a procedure is always the VALUE of some identifier or expression. This contrasts with ILISP and most LISP systems, where an identifier may simultaneously have a value and one or more functional properties. Scheme procedures are closed functions (called "closures" -- closures, objects, and continuations are all closed functions in Ski and may all be called "closures".) Because all functions are closed, there is no funarg problem in Scheme as there is with Lisp. All Scheme functions are automatically full upward and downward funargs. Procedures in Ski, including primitives (e.g., CAR, CDR, CONS), user-defined Scheme procedures, and Lisp functions (any Ilisp function may be called from SKI), are the values of variables. When one wants to know the value of a variable in SKI, one types that variable to the SKI top-level read-eval-print loop. SKI then prints on the terminal screen the value of the variable, if it has one. So if one types "CAR", then the value of the variable CAR -- the procedural value that computes the function CAR -- is printed on the screen.

What does a procedural value -- a closure -- look like? By definition in [Steele and Sussman 78], the run-time representation of a procedure (a closure) is implementation-dependent, while Scheme programs (which, of course, use procedures) are implementation-independent. For example, one way to generate a closure is to evaluate a lambda expression. The syntax for lambda expressions is part of the Scheme language; that syntax is implementation-independent. But a procedural value -- a closure -- generated at run time from a lambda expression, has an implementation- dependent representation. Therefore, PROCEDURE is a protected data type; procedural values should not be explored or decomposed with, say, CAR or CDR or PRINT. When a procedural value is printed on the terminal screen, it is shown as the atom "**CLOSURE**" (or "**OBJECT**" or "**CONTINUATION**"). If you type "CAR" to the top-level, it prints "**CLOSURE**", which is all you get to see of the representation of the closure. But if you type "(CAR (QUOTE (A B C)))", it works: it returns "A".

Like Lisp programs, Scheme programs are composed entirely of expressions. (Expressions, unlike GOTOs and DO loops in FORTRAN for example, return values when executed.) All expression are in prefix functional notation, and the CAR of an expression is called the "function" of that expression and is said to be in the "function position". When a Scheme expression is evaluated, each element of the

expression is evaluated once; the element in the function position must evaluate to a procedural value. The other elements of the expression are the arguments to this procedure and should evaluate to appropriate values. For example, in the expression, "(F (ADD1 X) (CAR Y) (GRAPPLE Z))", the identifier "F" must evaluate to a closure that is a functions of three arguments, and "(ADD1 X)", "(CAR Y)", and "(GRAPPLE Z)" are the arguments to that function.

The only Scheme expressions whose functions do not necessarily evaluate to closures are those with Scheme "magic words" in their function positions. Magic words are reserved words (key words). The SKI compiler has special routines to generate stack language for magic words. Magic words are used where one would typically use fexprs in Lisp, e.g., IF, ASETQ, and BLOCK, which correspond roughly to the Ilisp functions COND, SETQ, and PROGN. A SKI user may define additional magic words.

In many implementations of Lisp, the environment in which a function is called is always retained until the conclusion of the evaluation of that function call. This is because it is very difficult to analyze and anticipate the flow of control and, in particular, the fluid environment, in a dynamically scoped language like Lisp. However, since Scheme is a lexically scoped language, it is possible for a compiler for Scheme to determine when environments will and will not be needed at run time to complete the evaluation of any pending function calls. The data structures containing unneeded environments can be garbage-collected at the earliest possible moment. Because of this, iterative algorithms may be written in Scheme as recursive procedures without risk of run-time recursion stack overflows. Such procedures are called "tail-recursive" procedures and are useful for algorithms for which one would use a DO loop in Fortran, or a WHILE or FOR statement in Algol or Pascal.

A Scheme procedural value -- a closure -- carries with it ALL of the information (non-local variable bindings) necessary to compute a result EXCEPT for its actual arguments. A closure is represented in the SKI implementation as a 3-tuple:

1) the reserved flag-word "**CLOSURE**" (or "**OBJECT**" or "**CONTINUATION**" depending on the type of closure)

2) the body of the procedure

3) the static environment in which values of identifiers (including references to other functions, since they, too, are the values of identifiers) are to be found.

There are several kinds of expressions that evaluate to closures at run time:

1) lambda expressions
2) primitive function names (e.g., ADD1, CAR, CONS)
3) user-defined Scheme procedure names
4) Lisp function names
5) CATCH variables (which evaluate to continuations)
6) CLASS expressions (which evaluate to objects)

Any of these expressions evaluates to a Scheme procedural value and may be written in the function position (CAR position) of a Scheme expression. Arbitrary Scheme expressions that return as their values closures that are ultimately generated by one of the above types of expressions may be used as functions. Magic words are also written in the function positions of expressions. No other type of expression may be used as a function in a Scheme expression.

One way that closures can be created is with CATCH expressions. A catch expression has the syntax: (CATCH <var> <body>), where <var> is a variable, and <body> is an arbitrary Scheme expression. Within the lexical scope of a CATCH expression, <var> is automatically bound to a special kind of closure called a CONTINUATION or CONTINUATION OBJECT. A continuation is a procedural value and may be saved and invoked later. A continuation is usually saved by assigning it to some variable. A continuation may be thought of as a partly evaluated expression which may be restarted to resume evaluation at a later time. So, one may think of

a continuation as a process that has been suspended and is "asleep", but which may be "awakened" later to continue its computation. With continuations, then, one can simulate parallel processing. Processes, represented as Scheme procedures, may be packaged into continuations, and these continuations may be run little by little, one after another, round and round. The time slice that each process gets (the amount of computing time a process is allowed to run before it is put back to sleep) may be set initially by the SKI user and changed dynamically during a computation if desired.

D. SPECIFICATION OF THE SKI LANGUAGE

The material on the following seven pages is freely adapted from "The Revised Report on SCHEME, A Dialect of Lisp", [Steele and Sussman 78]. We have modified the excerpts here to be stylistically consistent with the rest of this report. Explanations of differences between Steele and Sussman's SCHEME and SKI have been inserted into the text.

A. THE REPRESENTATION OF SCHEME PROCEDURES AS S-EXPRESSIONS

SCHEME programs are represented as LISP s-expressions. The evaluator interprets these s-expressions in a specified way. This specification constitutes the definition of the language.

Atoms that are not atomic symbols (identifiers) evaluate to themselves. Typical examples of such atoms are numbers and strings (character arrays). Symbols are treated as identifiers (variables). They may be lexically bound by lambda expressions. There is a global environment containing values for (some) free variables. Many of the variables in this global environment initially have as their values primitive operations such as, for example, CAR, CONS, and *PLUS. SCHEME differs from most LISP systems in that the atom CAR is not itself an operation (in the sense of being an invocable object, e.g., a valid first argument to APPLY), but only has one as a value when considered as an identifier.

Non-atomic forms are divided by the evaluator into two classes: combinations and "magic forms". Magic forms are recognized by the presence of a "magic word" (a reserved word) in the CAR position of the form. All non-atomic forms that are not magic forms are considered to be combinations. The system has a small initial set of magic words; there is also a mechanism for creating new ones.

When a combination is to be evaluated, it is considered to be a list of subforms. These subforms are all evaluated. The first value must be a procedure; it is applied to the other values to get the value of the combination. There are four important points here:

(1) The procedure position (also called the "function position") is always evaluated just like any other position. (This is why the primitive operators are the VALUES of global identifiers.)

(2) The procedure is never "re-evaluated"; if the first subform fails to evaluate to an applicable procedure, it is an error. Thus, unlike most LISP systems, SCHEME always evaluates the first subform of a combination exactly once.

(3) The argument forms (and procedure form) may in principle be evaluated in any order. This is unlike the usual LISP left-to-right order.

(4) The arguments are all completely evaluated before the procedure is applied; that is, Scheme, like much of Lisp, is an applicative-order language.

B. CATALOGUE OF MAGIC FORMS

B.1 KERNEL MAGIC FORMS

The magic forms in this section are all part of the kernel of Scheme, and so must exist in any SCHEME implementation.

(LAMBDA (<identifier list>) <body>)

Lambda expressions evaluate to procedures (to closures). Unlike most LISP systems, SCHEME does not consider a lambda expression (an s-expression whose CAR is the atom LAMBDA) to be a procedure. A lambda expression only EVALUATES to a procedure. A lambda expression should be thought of as a partial DESCRIPTION of a procedure; a procedure and a description of it are conceptually distinct objects. A lambda expression must be closed (associated with an environment) to produce a procedure. Evaluation of a lambda expression performs such a closure operation.

The resulting procedure takes as many arguments as there are identifiers in <identifier list> of the lambda expression. When the procedure is eventually invoked, the intuitive effect is that the evaluation of the procedure call is equivalent to the evaluation of <body> in an environment consisting of (a) the environment in which the lambda expression had been evaluated to produce the procedure, plus (b) the pairing of the identifiers of <identifier list> with the arguments supplied to the procedure. The pairings (b) take precedence over the environment (a), and to prevent confusion no identifier may appear twice in <identifier list>. The net effect is to implement ALGOL-style lexical scoping, and to solve the notorious funarg problem.

(IF <predicate> <consequent> <alternative>)

This is a primitive conditional operator. The <predicate> is evaluated. If the result is non-NIL, then <consequent> is evaluated, and otherwise <alternative> is evaluated. The resulting value is the value of the IF form.

(IF <predicate> <consequent>)

As above, but if <predicate> evaluates to NIL, then NIL is the value of the IF form.

(QUOTE <s-expression>)

As in Lisp, this quotes the argument form so that it will be passed verbatim as data; the value of this form is <s-expression>.

(LABELS (<labels list>) <body>)

where <labels list> is ((<id1> <lambda exp1>) ... (<idn> <lambda expn>))

This has the effect of evaluating <body> in an environment where all the identifiers <idi> (which, as for LAMBDA, must all be distinct) in <labels list> evaluate to the values of the respective lambda expressions <lambda expi>. Furthermore, the procedures which are the values of the <lambda expi> are themselves closed in that environment, and not in the outer environment; this allows the procedures to call themselves and each other recursively.

```
(CLASS <basis>
       (<msg1> <lambda-exp1>)
       (<msg2> <lambda-exp2>)
       ...
       (<msgn> <lambda-expn>))
```

CLASS is not specified in [Steele and Sussman 78]. CLASS is discussed in section I, "CLASSES AND OBJECTS".

B.2 SIDE EFFECTS

These magic forms produce side effects in the environment.

(DEFINE <identifier> <lambda expression>)

DEFINE is used for defining a procedure in the global environment permanently, as opposed to LABELS, which is used for temporary procedure definitions in a local environment. DEFINE takes a name <identifier> and a lambda expression; it evaluates <lambda expression> in the global environment and causes the result to be the global value of <identifier>.

(DEFINE <identifier> (<identifier list>) <body>)
(DEFINE (<identifier> <identifier list>) <body>)

These alternative syntaxes permitted by DEFINE are equivalent to:

(DEFINE <identifier> (LAMBDA (<identifier list>) <body>))

These alternative forms are provided to support stylistic diversity. The Ski DEFINE has two more alternate syntaxes. It may be used to define both global procedures and global values. The added syntax for defining values is:

(DEFINE <identifier> <datum>)

where <datum> is an arbitrary s-expression.

A procedure or datum DEFINITION, defined with DEFINE, can be overridden only by another DEFINE statement. But global procedure and datum VALUES can be changed during program execution by ASETQ (described below). The definition of an identifier is its INITIAL value in the Scheme global environment. It is this value (which can change) that is used in Scheme computations. One can always force an identifier to "back up" to its DEFINITION -- i.e., override its current value and "reinitialize" it -- by calling DEFINE with no <body>:

(DEFINE <identifier>)

This last DEFINE syntax reinitializes the value of <identifier> to its most recent definition.

(ASETQ <identifier> <form>)

This is analogous to the LISP primitive SETQ. Ski environments are separate from the Ilisp environment. ASETQ and SETQ are analogous, not synonymous. They never refer to the same variable bindings.

B.3 DYNAMIC MAGIC

These magic forms implement continuations and fluid (dynamic) variables.

(FLUIDBIND (<fluidbind list>) <form>)

where <fluidbind list> is ((<id1> <exp1>) (<id2> <exp2>) ... (<idn> <expn>))

This evaluates <form> in the environment of the FLUIDBIND form, with a DYNAMIC environment to which dynamic bindings of the identifiers <idi> in <fluidbind list> have been added. Any procedure dynamically called by <form>, even if not lexically scoped within to the FLUIDBIND form, will use this dynamic environment (unless modified by further FLUIDBINDS, of course). The dynamic environment is restored on return from <form>.

Most LISP systems use a dynamic environment for ALL variables. SCHEME provides two distinct environments. The fluid variable named FOO is completely different from a normal lexical variable named FOO, and the access mechanisms for the two are distinct. In SKI, the fluid and lexical environments intersect in the global environment; all global identifiers are also in the outermost level of the fluid environment.

(FLUID <identifier>)

The value of this form is the value of <identifier> in the current dynamic environment.

(FLUIDSETQ <identifier> <form>)

The value of <form> is assigned to <identifier> in the current dynamic environment.

(STATIC <identifier>)

The value of this is the lexical value of <identifier>; writing this is the same as writing just <identifier>.

(LISP <identifier>)

In SKI, evaluation of this form returns the value of <identifier> in the current Ilisp environment.

(CATCH <identifier> <form>)

This evaluates <form> in an environment where <identifier> is bound to a continuation (also called an "escape object" by Sussman and Steele). A continuation can be invoked as if it were a procedure of one argument. When the continuation is so invoked, control proceeds as if the CATCH expression had returned with the value of the supplied argument as its value.

The semantics of CATCH and FLUIDBIND are intertwined. When a continuation is called, the dynamic environment is restored to the one that was current at the time the CATCH form was evaluated.

C. SYNTACTIC EXTENSIONS

SCHEME has a syntactic extension mechanism that provides a way to make an identifier into a magic word, and to associate a macro with that word. The macro accepts the magic form as an argument, and produces a new form; this new form is then evaluated in place of the original (magic) form.

C.1 SYSTEM-PROVIDED EXTENSIONS

Some standard syntactic extensions are provided by the system for convenience in ordinary programming. They are distinguished from other magic words in that they are defined in terms of others rather than being primitive. For expository purposes they are described here in a pattern-matching/production-rule kind of language. The matching is on s-expression structure, not on character string syntax, and uses the definition of lists in dot notation, for example:

(A B C) = (A .(B .(C . NIL))).

Thus the pattern (x . r) matches (A B C), with x = A and r = (B C). The ordering of the productions is significant; the first one that matches is to be used.

(BLOCK <x1> <x2> ... <xn>)

BLOCK sequentially evaluates the subforms <xi> from left to right. For example:

(BLOCK (ASETQ A 43) (PRINT A) (PLUS A 1))

returns 44 after setting A to 43 and then printing it. The following rules show how BLOCK is implemented tail-recursively:

(BLOCK x) → x
(BLOCK x . r) → ((LAMBDA (A B) (B)) x (LAMBDA NIL (BLOCK . r)))

(LET ((<v1> <x1>) (<v2> <x2>) ... (<vn> <xn>)) <body>)

LET provides a convenient syntax for binding variables <vi> to the values of the arbitrary expressions <xi>. It allows the forms <xi> to appear textually adjacent to their corresponding variables. The variables are all bound simultaneously, not sequentially, and the initialization forms <xi> may be evaluated in any order. LET expressions expand into lambda expressions:

(LET ((<v1> <x1>) (<v2> <x2>) ... (<vn> <xn>)) <body>)
→ ((LAMBDA (<v1> <v2> ... <vn>) <body>) <x1> <x2> ... <xn>)

(TEST <predicate> <function> <alternative>)

<predicate> is evaluated; if its value is non-NIL then <function> should evaluate to a procedure of one argument, which is then invoked on the value of <predicate>. Otherwise <alternative> is evaluated. TEST is of use with predicates or sequencing operators, e.g., AND, OR, BLOCK, that return useful non-NIL values. For example, AND returns NIL as the truth value "false", but returns the value of its last argument (instead of the atom T as in some LISPs) as the truth value "true". This non-NIL value may be useful to the calling program.

```
(TEST <predicate> <function> <alternative>)
→ ((LAMBDA (P F A) (IF P ((F) P) (A)))
    <predicate>
    (LAMBDA NIL <function>)
    (LAMBDA NIL <alternative>))
```

(COND (<pred1> <exp1-1> <exp1-2> ... <exp1-i>)
...
(<predn> <expn-1> <expn-2> ... <expn-j>)

This COND is an extension of the Lisp COND. A singleton clause returns the value of <predicate> (if that value is non-NIL), and a clause with two or more forms treats the first as the predicate and the rest as constituents of a BLOCK, thus evaluating them in order.

The extension to COND made in Scheme is flagged by the atom "=>" as the second form in a COND clause. (It cannot be confused with the more general case of BLOCK constituents because having the atom "=>" as the first element of a BLOCK is not useful.) In this situation the form following "=>" should have as its value a function of one argument. If the value of <predicate> is non-NIL, this function is determined and invoked on the value returned by <predicate>. The use of this extension is as explained for TEST above.

```
(COND) → NIL
(COND (p) . r) → ((LAMBDA (V R) (IF V V (R))) p (LAMBDA NIL (COND . r)))
(COND (p => f) . r) → (TEST p f (COND . r))
(COND (p . e) . r) → (IF p (BLOCK . e) (COND . r))
```

(OR <x1> <x2> ... <xn>)

This standard Lisp primitive evaluates the forms <xi> in order, returning the first non-NIL value, and ignoring all following forms. If all of the <xi> evaluate to NIL, then NIL is returned.

```
(OR) → NIL
(OR x) → x
(OR x . r) → (COND (x) (T (OR . r)))
```

(AND <x1> <x2> ... <xn>)

This standard Lisp primitive evaluates the forms <xi> in order. If any of the <xi> evaluates to NIL, then NIL is returned, and succeeding forms <xi> are ignored. If all forms produce non-NIL values, the value of the last is returned.

```
(AND) → T
(AND x) → x
(AND x . r) → (COND (x (AND . r)))
```

(AMAPCAR <f> <list1> <list2> ... <listn>)

AMAPCAR is analogous to the Lisp MAPCAR function. The function <f>, a function of n arguments, is mapped simultaneously down the lists <list1>, <list2>, ..., <listn>; that is, the value of <f> is applied to tuples of successive

elements of the lists. The values returned are collected and returned as a list. Note that AMAPCAR of a fixed number of arguments could easily be written as a function in Scheme. It is a syntactic extension only so that it may accommodate any number of arguments, which saves the user the trouble of defining an entire set of functions AMAPCAR1, AMAPCAR2, ... where AMAPCARn takes n+1 arguments.

C.2 USER-PROVIDED EXTENSIONS

SKI provides the function DSYNM, which stands for "define syntactic macro", as a way for the user to extend the inventory of magic words. Magic words defined with DSYNM are known as "synmacs" because these macro definitions are stored on the property lists of identifiers under the property name SYNMAC.

D. PRIMITIVE SCHEME FUNCTIONS

All of the usual Lisp subrs are available in Scheme as procedures which are the values of global variables. The particular primitives CONS, CAR, CDR, ATOM, and EQ are part of the kernel of Scheme. Others, such as *PLUS, *DIF, *TIMES, *QUO, EQ, EQUAL, RPLACA, RPLACD, etc. are quite convenient to have.

Although there is no way in Scheme to write a lexpr (a function of a variable number or arguments), Lisp lsubrs and lexprs are available to the Scheme user. Only lsubrs and lexprs that take fixed numbers of arguments may be called directly from SKI. Other lsubrs and lexprs may called through ILISP.

The primitive functions PROCP and ENCLOSE are part of the kernel of Scheme.

(PROCP <thing>)

This predicate is true of procedures and not of anything else. Thus if (PROCP X) is true, then it is safe to invoke the value of X. More precisely, if PROCP returns a non-NIL value, then that value is the number of arguments accepted by the procedure. Ski PROCP is true of closures, objects, and continuations.

(ENCLOSE <fnrep> <envrep>)

ENCLOSE takes two arguments, one representing the code for a procedure, the other representing the (lexical) environment in which the procedure is to run. ENCLOSE returns a closed procedure -- a closure -- which can be invoked.

The representation of the code is the standard s-expression description of a procedure, i.e., a lambda expression. The representation of the environment is an association list (an a-list):

((<var1> <value1>) (<var2> <value2>) ... (<varn> <valuen>)).

NIL represents the global lexical environment. (The format of the value of <envrep> in Ski differs slightly from the specification in [Steele and Sussman 78]. Their SCHEME requires dotted pairs, (<vari> . <valuei>), in the value of <envrep>.)

This description of ENCLOSE should not be construed as describing how SKI represents either environments or code internally. A closure returned by ENCLOSE has an implementation-dependent representation and should not "explored" with functions like CAR and CDR. All that ENCLOSE guarantees to do is to compute a procedural value given a description of its desired behavior. The description must be in the prescribed lambda-expression form, but the resulting closure is in a compiled form that is convenient to the implementation, and must be treated as an indivisible value.

The multiprocessing primitives CREATE!PROCESS, START!PROCESS, STOP!PROCESS, and TERMINATE, all described in [Steele and Sussman 78], are not implemented in SKI. SKI provides the more general and elegant multiprocessing constructs of [Wand 80b], see section K, "SIMULATING PARALLEL PROCESSING".

E. DEFINITIONS AND PRETTY-PRINTING THEM

The word "definition" will be used in a special way in this manual. Definitions in SKI are standard ways to represent Scheme and Ilisp functions and values, comments, and files descriptions. You can think of DEFINITION as a data type, and the kinds of definitions -- Scheme functions, Ilisp functions, values, comments, files, etc. -- as subtypes. Standardizing the representation of definitions allows convenient and uniform operations on definitions. Common operations on definitions are creation, viewing, testing, and editing. A definition is an s-expression of the form:

(<definer> <identifier> <exp1> <exp2> ... <expn>)

A definer is an Ilisp function that stores an s-expression, pieced together from the arbitrary s-expressions <expi>, on the property list of <identifier>. Familiar definers are DE for defining Ilisp exprs, DF for fexprs, DM, etc. DEFINE is the definer for Scheme procedures. Some definers may take as arguments specific numbers of <expi>. Others may take arbitrarily many. A definer never evaluates any of its arguments. It just has the side effect of putting an expression onto the property list of <identifier>. It returns <identifier>. A list of all definers, paired with their corresponding property names, is kept on the Ilisp global variable DEFINERS. DEFINERS is initialized to:

```
((DEFINE DEF)   (DSYNM SYNMAC)  (DE EXPR)       (DM MACRO)
 (SETQQ VALUE)  (DF FEXPR)      (FILE FILE))
```

Here are some sample definitions. The prompt of the SKI top level is ">". The evaluation of each definition returns <identifier>, which is printed as the value of the definition.

```
>(DEFINE RAC (L) (IF (CDR L) (RAC (CDR L)) (CAR L)))
RAC
>(DEFINE X (IF RAC WORKS IT WILL RETURN ELEPHANT))
X
>(FILE RAC (PAGE (> DEFINE RAC) (> DEFINE X)))
RAC
>
```

(The syntax for file definitions, as in "(FILE RAC (PAGE ...))" above, is given in section J, "EDITING A FILE".)

Notice that the identifier RAC is given two different definitions, as a def and as a file. This seems to violate the spirit of the Scheme language, where an identifier has only a single value, and that value may be a procedure or a datum. The pretty-printing and file-handling functions used by SKI are Ilisp functions and use Ilisp property lists to store definitions of functions, values, comments, and files. The rule for evaluating Ski expressions is roughly this:

> Any function call (<f> <arg1> <arg2> ... <argn>) in which <f> has a value in the Scheme environment is evaluated within SKI. In any call where <f> is not bound in the Scheme environment, <f> must be an Ilisp function; its arguments are evaluated within SKI and then the call is handed to the ILISP interpreter for evaluation.

Thus, calls to the pretty-printing and editing functions are evaluated by ILISP and follow ILISP conventions for the attributes of identifiers.

Given these definitions for RAC and X, we can call the function RAC to test it:

```
>(RAC X)
ELEPHANT
```

The function PP (for "pretty-print") lets you view all of the definitions on the property list of an identifier. RAC has definitions as a def and a file:

```
>(PP RAC)
(DEFINE RAC (L) (IF (CDR L) (RAC (CDR L)) (CAR L)))

(FILE RAC (PAGE (DEFINE RAC) (DEFINE X)))

NIL
```

The value of PP is always NIL, the last thing printed above. For PP to pretty-print definitions, the definer for that definition type, e.g., DE, DM, DEFINE, must be in the list DEFINERS, and the property name that appears on the property list of <identifier>, e.g., EXPR, MACRO, DEF, must be in the list GRINPROPS. GRINPROPS is a global Ilisp variable and is initialized to (DEF SYNMAC VAL EXPR MACRO VALUE FEXPR FILE). There is one definer in DEFINERS for each property name in GRINPROPS.

PP (PP <id1> <id2> ... <idn>)

PP pretty-prints the definitions of the identifiers <idi>. PP does not evaluate its arguments. Recall that the macro versions (fexpr-like versions) of pretty-printing and editor functions typically do not evaluate their arguments, and may take variable numbers of arguments. The corresponding subr versions have an "X" or an "LX" appended to their names, as here with "PPX". (See section B, "CONTENTS AND NAMING CONVENTIONS".) PP returns NIL.

PPX (PPX <exp>)

PPX takes one argument, a single identifier or list, and pretty-prints it. If <exp> evaluates to an identifier, that identifier's definitions are pretty-printed. PPX pretty-prints all of an identifier's definitions whose property names appear in GRINPROPS. If <exp> evaluates to a list, that list is pretty-printed. PPX evaluates its argument. PPX returns NIL.

defs and vals

The definer DEFINE stores the definition of a Scheme source-language procedure or datum on the property list of an identifier under the property name DEF. When an identifier is given a Scheme source-language procedure or datum definition with DEFINE, it keeps that definition until it is changed with another DEFINE. The DEF property of an identifier (if there is one) is used as the initial Scheme value for that identifier. The Scheme primitive ASETQ may be used to change a global Scheme value (a global Scheme value is kept on the property list of an identifier under the property name VAL), but does not change the def of the identifier.

The val of an identifier is its value used in Scheme computations. The def of that identifier is compiled to provide the initial val for the identifier. If a def is ever redefined with DEFINE or edited with EDIT, the new definition is automatically recompiled to provide a new val. Lambda binding of (with LAMBDA, LET, or LABELS) and assignment to (with ASETQ) an identifier only affect its val.

Only defs can be written onto files with FILE or EDIT (section J, "EDITING A FILE"). This is because vals are the compiled, implementation-dependent representations of Scheme language constructs, and may contain unprintable circular lists. Defs are the source-language, implementation-independent representations of Scheme language constructs. Data (things of which PROCP is false, e.g., numbers, atoms, lists, strings) have identical VAL and DEF properties. When your Ski program computes a val that you wants to save on a file, you may use the function DEFSETQ to reset the def of the identifier to its current val. The syntax is:

(DEFSETQ <identifier> <expression>)

The value of <expression> is used as the body of a DEFINE expression for <identifier>. For example, (DEFSETQ X (LIST 1 2 3)) has the same effect as (DEFINE X (1 2 3)), but has the advantage that the value of <expression> is computed in the current environment. DEFSETQ, then, is the same as ASETQ for a global identifier, except that it also updates the def property of the identifier. DEFSETQ is provided as a way to prepare definitions to be written

onto files. The user must take care to avoid writing the compiled versions of procedural values onto files, because they may not be Ilisp printable, i.e., they may contain circular lists that cause the Ilisp subr SPRINT to loop.

PP pretty-prints Ilisp values and functions as well as Ski definitions. To obtain the values of Ilisp variables for use in Ski computations, the function LISP may be used. Like (STATIC <var>) and (FLUID <var>), (LISP <var>) specifies the environment from which the value of <var> is retrieved. The function LISP, though, is more general than STATIC and FLUID, because (LISP <exp>) calls the Ilisp interpreter to evaluate <exp>, any Ilisp expression.

a sample Ski program: the 3n+1 problem

We are now ready to present a sample Ski program which we will develop throughout this manual to give glimpses of Scheme, tail-recursion, the editor, the break package, classes and objects, tracing, the file editor, and simulated multiprocessing. Suppose that a new SKI user, let's call her KIM, wants to write a program to investigate the 3n+1 property of some integers. This property is the number of steps required to reach the value 1 from the starting integer n, n≥1, by the following rules:

if n is even, then divide it by 2;
if n is odd, then multiply it by 3 and add 1.

So, for example, starting with 3, the steps are:

```
step 1          3  is odd so    3 * 3 + 1 is    10
step 2          10 is even so   10 / 2 is       5
step 3          5  is odd so    5 * 3 + 1 is    16
step 4          16 is even so   16 / 2 is       8
step 5          8  is even so   8 / 2 is        4
step 6          4  is even so   4 / 2 is        2
step 7          2  is even so   2 / 2 is        1
```

The answer is 7. Seven steps were required to get from 3 to 1. This property varies wildly for different integers: for 26 it is 10, for 27 it is 111, for 28 it is 18, etc.. Our programmer, Kim, defines a simple Scheme function, called "TNPO" (for "three n plus one"). (TNPO <n>) will return the 3n+1 property of <n>. TNPO calls a help function, TNPO-H, to calculate the answer.

```
>(DEFINE TNPO (N) (TNPO-H N 0))
TNPO
>
```

The value of the DEFINE expression is TNPO, the name of the defined function. Now, because she is really a Lisp programmer at heart, Kim decides to write TNPO-H, the help function, as an Ilisp expr.

```
>(DE TNPO-H (N A) (COND ((EQ N 1) A)
                        ((EVENP N) (TNPO-H (QUOTIENT N 2) (ADD1 A)))
                        (T (TNPO-H (ADD1 (TIMES N 3)) (ADD1 A)))))
TNPO-H
>(DE EVENP (N) (ZEROP (REMAINDER N 2)))
EVENP
>
```

To view her definitions Kim calls the function PP with the names of her functions as its arguments:

page 19. Example of tail recursion. Sample run-time error.

```
>(PP TNPO TNPO-H EVENP)

(DEFINE TNPO (N) (TNPO-H N 1))

(DE TNPO-H (N A)
   (COND ((EQ N 1) A)
         ((EVENP N) (TNPO-H (QUOTIENT N 2) (ADD1 A)))
         (T (TNPO-H (ADD1 (TIMES N 3)) (ADD1 A))))))

(DE EVENP (N) (ZEROP (REMAINDER N 2)))

NIL
>
```

Now, Kim tries out her program:

```
>(TNPO 3)
7
>(TNPO 28)
18
>(TNPO 27)
⊗SPEC
PUSHDOWN CAPACITY EXCEEDED
LAST INPUT: (TNPO 27)

(TNPO-H BROKEN)
1:
```

An error occurred because the Ilisp recursion stack overflowed. Ilisp did not execute the expr TNPO-H tail-recursively. It tried to stack up all 111 function invocations, and didn't have room. (Of course, it is possible to allocate more space to the ILISP recursion stack, but that is not the point here.) Control of the interrupted computation is now in the break package. A break occurs whenever there is an error. The top level of the break package is a read-eval-print loop. The prompt for the break package is the number of the level of the break, here 1, and a colon, ":". Kim types an up-arrow, "↑", to get out of the break package.

```
1:↑
SKI-ERROR
>
```

The prompt is now the SKI top-level prompt again, ">", signaling that everything is back to normal.

When a recursive function is executed in ILISP (or most other LISP systems), the recursion stack grows to a size proportional to the number of recursive calls executed. However, algorithms written as tail-recursive Scheme procedures are executed by SKI with a recursion stack of constant size. TNPO-H happens to use a tail-recursive algorithm. Kim wisely decides to redefine TNPO-H as a Scheme function.

```
>(DEFINE TNPO-H (N A)
          (IF (EQ N 1) A
              (IF (EVENP N) (TNPO-H (QUOTIENT N 2) (ADD1 A))
                            (TNPO-H (ADD1 (TIMES N 3)) (ADD1 A)))))
TNPO-H
>
```

Now, two definitions for TNPO-H exist, an expr and a def. Kim tries her program again:

```
>(TNPO 27)
⊗SPEC
PUSHDOWN CAPACITY EXCEEDED
LAST INPUT: (TNPO 27)

(TNPO-H BROKEN)
1:
```

What's wrong? The same error occurred. SKI compiles a Scheme function only once -- the first time it is executed. The call to TNPO-H from TNPO was already compiled in the stack language for TNPO as a call to an Ilisp function, not as a call to a Scheme function. Kim must recompile TNPO so that the reference to TNPO-H will be compiled as a call to a Scheme function. (This works because Scheme functions have higher priority that Ilisp functions. The priorities of different function types are listed in section L, "THE SKI COMPILER".) The easiest way to recompile a Scheme function is to call DEFINE (DEFINE is documented in section D, "SPECIFICATION OF THE SKI LANGUAGE"). Kim types a DEFINE statement with no definition body. This statement has no effect other than removing the compiled VAL property of the identifier TNPO. The next time TNPO is used, it will be recompiled.

```
1:↑
>(DEFINE TNPO)
TNPO
>(TNPO 27)
111
>
```

F. EDITING A DEFINITION

EDIT (EDIT <id> <prop>)

Any definition may be edited with EDIT. EDIT does not evaluate its arguments. If <id> is an atom, then the definition found under the <prop> property of <id> will be edited. The editor used is the ILISP structure editor documented in [Bobrow et al] and [Meehan 79]. Before the ILISP editor is called, EDIT untraces <id> so that, if <id> is a traced function, the source version and not the traced version of the definition will be edited.

The final argument, <prop>, is optional. If <prop> is not provided, then the first property in GRINPROPS that occurs on the property list of <id> will be used. If no such property exists, then EDIT generates an error. The s-expression that is edited is a DEFINING EXPRESSION, i.e., a list whose first element is a function name from DEFINERS. For example, if the atom SEVEN has on its property list the s-expression

(LAMBDA (X) (CADDDR (CDDDR X)))

under the property DEF, then the call

(EDIT SEVEN DEF)

will call the editor on the list

(DEFINE SEVEN (X) (CADDDR (CDDDR X))).

You can change the type of a definition by changing the definer, here DEFINE, which is the first element in the form being edited. You can change the name of the definition by changing the second element of the form, here SEVEN. If the definer or name is changed during editing, the old version of the definition will still exist under the former property type and name. You can remove it explicitly with REMPROP. (It is important to do this if, for example, a def is changed to an expr during an editing session. Unless the def is removed from the property list of the name of the definition, the SKI compiler will continue to generate stack language to access the def, because defs have higher precedence than exprs do.) After the definition of <id> has been edited, EDIT returns <id>.

EDITX

Same as EDIT but does evaluate its arguments.

Let's pick up the sample Scheme program from the last section, the 3n+1 problem. Our programmer, Kim, first defined a def called TNPO and an expr called TNPO-H. Let's back up to the point where she wants to change TNPO-H, an Ilisp expr, into a Scheme def. To do this, she calls the editor:

```
>(EDIT TNPO-H EXPR)
EDIT
#PP
(DE TNPO-H
    (N A)
    (COND ((EQ N 1) A)
          ((EVENP N) (TNPO-H (QUOTIENT N 2) (ADD1 A)))
          (T (TNPO-H (ADD1 (TIMES N 3)) (ADD1 A)))))
>
```

In SKI, the entire definition, including the definer (here DE) and name (here TNPO-H), can be edited. This allows Kim to change the type and name of the function in the editor.

```
#(1 DEFINE)
```

This command says "replace the first thing in the s-expression by DEFINE".

page 22. Sample editing session, continued.

```
#R TNPO-H TNPO-HELP
```

The R command, for "replace", changes every occurrence of TNPO-H to TNPO-HELP.

```
#R A ANSWER
```

This replaces all occurrences of A with ANSWER.

```
#PP
(DEFINE TNPO-HELP
        (N ANSWER)
        (COND ((EQ N 1) ANSWER)
              ((EVENP N) (TNPO-HELP (QUOTIENT N 2) (ADD1 ANSWER)))
              (T (TNPO-HELP (ADD1 (TIMES N 3)) (ADD1 ANSWER)))))
#OK
TNPO-HELP
>
```

Now Kim changes the definition of TNPO to call TNPO-HELP instead of TNPO-H.

```
>(EDIT TNPO DEF)
EDIT
#R TNPO-H TNPO-HELP
#PP
(DEFINE TNPO (N) (TNPO-HELP N 0))
#OK
TNPO
>(TNPO 27)
111
>
```

G. TRACING PROCEDURES

TRACE and UNTRACE are SKI synmacs that may be used to trace the execution of Scheme defs and synmacs, i.e., procedures defined with DEFINE or DSYNM. When a def F is traced by evaluating the form (TRACE F), F is redefined so that it will print out a message whenever it is called. This entry message tells the names of the formal parameters of F, and the values of their actual arguments in that call. A list of all currently traced function names is kept on the global ILISP variable TRACEDFNS. Tracing of Scheme procedures differs slightly from the tracing of Ilisp functions as documented in [Bobrow et al] and [Meehan 79]. Because tail recursion in Scheme allows recursive and nested function calls that do not nest environments, the depth of nesting of calls is not bounded by the depth of a recursion stack as it is in ILISP. The depth of nesting of calls to traced Scheme procedures is not displayed by indenting trace messages as it is in calls to nested ILISP functions.

It may happen that a Ski function does not return, i.e., execution of the function may be interrupted by evaluation of an continuation (a CATCH variable). No exit messages from traced defs that are so interrupted can be printed. Furthermore, if a def were redefined to print an exit message, the redefined version of the def could no longer be tail recursive. Such a redefinition could significantly change the behavior of an algorithm. So no exit messages from traced defs are printed.

A traced synmac prints an entry message when it is called, and also prints the expansion it computes in an exit message.

TRACE (TRACE <fn1> <fn2> ... <fnn>)
traces the functions <fni>. Each <fni> is an identifier. If no def or synmac definition is found for <fni>, Ilisp's TRACE is called to trace <fni>.

UNTRACE (UNTRACE <fn1> <fn2> ... <fnn>)
untraces the functions <fni>. Each <fni> is an identifier. If no def or synmac definition is found for <fni>, Ilisp's UNTRACE is called to untrace <fni>.

Suppose there is an error in Kim's definition of TNPO-HELP. She has misspelled the variable ANSWER as "ANWER" in the first COND clause.

```
>(PP TNPO-HELP)
(DEFINE TNPO-HELP
        (N ANSWER)
        (COND ((EQ N 1) ANWER)
              ((EVENP N) (TNPO-HELP (QUOTIENT N 2) (ADD1 ANSWER)))
              (T (TNPO-HELP (ADD1 (TIMES N 3)) (ADD1 ANSWER)))))
```

But she doesn't notice the error:

```
>(TNPO 5)
ANWER is an unbound Scheme variable.
SKI-ERROR
>
```

and gets an error message from SKI. Kim traces TNPO-HELP to watch it and to see what goes wrong.

```
>(TRACE TNPO-HELP)
(TNPO-HELP)
>(TNPO 5)
Entering TNPO-HELP with:
        (N = 5)
        (ANSWER = 0)
Entering TNPO-HELP with:
        (N = 16)
        (ANSWER = 1)
```

page 24. Sample traced procedure, continued.

```
Entering TNPO-HELP with:
        (N = 8)
        (ANSWER = 2)
Entering TNPO-HELP with:
        (N = 4)
        (ANSWER = 3)
Entering TNPO-HELP with:
        (N = 2)
        (ANSWER = 4)
Entering TNPO-HELP with:
        (N = 1)
        (ANSWER = 5)
ANWER is an unbound Scheme variable.
SKI-ERROR
>
```

Kim now knows that the error does not occur until the last call to TNPO-HELP. She pretty-prints TNPO-HELP and notices and corrects the error.

```
>(EDIT TNPO-HELP DEF)
EDIT
#PP
(DEFINE TNPO-HELP
        (N ANSWER)
        (COND ((EQ N 1) ANWER)
              ((EVENP N) (TNPO-HELP (QUOTIENT N 2) (ADD1 ANSWER)))
              (T (TNPO-HELP (ADD1 (TIMES N 3)) (ADD1 ANSWER)))))
#R ANWER ANSWER
#OK
TNPO-HELP
>(TNPO 5)
5
>
```

Editing a function with EDIT automatically untraces it and causes it to be recompiled the next time it is called. Just for fun, Kim traces TNPO-HELP again to watch a complete trace.

```
>(TRACE TNPO-HELP)
(TNPO-HELP)
>(TNPO 5)
Entering TNPO-HELP with:
        (N = 5)
        (ANSWER = 0)
Entering TNPO-HELP with:
        (N = 16)
        (ANSWER = 1)
Entering TNPO-HELP with:
        (N = 8)
        (ANSWER = 2)
Entering TNPO-HELP with:
        (N = 4)
        (ANSWER = 3)
Entering TNPO-HELP with:
        (N = 2)
        (ANSWER = 4)
Entering TNPO-HELP with:
        (N = 1)
        (ANSWER = 5)
5
>(UNTRACE TNPO-HELP)
(TNPO-HELP)
```

H. INTERACTIVE HELP AND ERROR MESSAGES

The SKI run-time system provides help and on-line documentation to the interactive user in two ways:

I) pages of this manual may be typed to the terminal screen, and
II) detailed diagnostics are printed when errors occur.

I) The initial values of the Scheme variables "?" and "HELP" are messages telling how to type parts of this manual on the terminal screen. For example, the message at SAIL is:

```
SKI documentation is on file SKI.MAN[1,GAC].
Type '(TY (SKI-MAN@1-GAC 3))' to view the directory page.
Then type '(TY (SKI-MAN@1-GAC <pagea> <pageb> ... <pagen>))',
where the <pagei> are integer page numbers,
to view specific pages of documentation.
```

The Ilisp function TY types files to the terminal screen and is discussed in section J, "EDITING A FILE".

II) Because SKI is an interactive system, all errors, including compiler errors, are treated as run-time errors. The SKI compiler is called as needed as a subroutine to compile Ski expressions and procedures. The SKI stack language interpreter calls the compiler through the Scheme primitive ENCLOSE. ENCLOSE is implemented as an Ilisp function that returns the executable (compiled) form of a Scheme expression. Thus all compiler errors are run-time errors incurred when the interpreter calls the Ilisp function ENCLOSE. A global Ski procedure is compiled not when it is defined -- when a DEFINE expression is evaluated -- but when it is first called in the course of the execution of a Ski program. More accurately, a global Ski procedure is compiled when the value of the identifier that names it is first needed. For example, if you type

```
>(DEFINE RAC (L) (IF (CDR L) (RAC (CDR L)) (CAR L)))
RAC
```

then RAC is defined but is not yet compiled. You can cause it to be compiled by asking for the value of the identifier RAC.

```
>RAC
**CLOSURE**
```

The error-handling routines in SKI were written to meet the following goals:

a) When an error occurs in the execution of the compiled, stack-language form of a Ski expression, the programmer should have immediate access to the corresponding source-language Ski expression in order to edit it and to make permanent changes to it at the time and site of the error.

b) When possible, execution should continue after changes to the source-language Ski expression are made. The corrected Ski expression should be recompiled and the computation should continue from where it left off.

c) Ski errors and Ilisp errors should be compatible, i.e., errors generated by the SKI interpreter and compiler, by Ski programs, by the ILISP interpreter, and by Ilisp programs should all be treated uniformly.

d) Error detecting and handling should not slow the SKI stack language interpreter.

To meet the above goals, the SKI compiler can produce stack-language output in three different flavors -- with three different levels of error-handling information built into the stack-language instructions themselves. These three levels -- modes of compilation -- are called "streamlined mode", "normal mode", and "sleuth mode".

1) streamlined mode

Stack language compiled in this mode contains no error-recovery information at all. This stack language is very compact. It occupies less memory space than the other flavors. Thoroughly debugged procedures should probably be compiled in this mode.

2) normal mode

This is the default mode. Error-recovery information is inserted into stack language generated from Scheme expressions that the USER wrote -- i.e., for Ski expressions that are parts of the globally defined Ski procedures that constitute the user program -- but NOT for Ski expressions that result from macro expansions of expressions in the user program. In general, it does little good for a user to make editing changes to macro-expanded Ski expressions because these are not parts of her Ski program. The only changes worth making are those that make permanent improvements to the Ski procedures that the programmer wrote as part of her program.

3) sleuth mode

In this mode, error-recovery information is compiled into ALL stack language generated by the compiler, even that generated from macro-expanded Ski expressions. Code generated in this mode is approximately twice as bulky -- uses about twice as many CONS cells -- as streamlined code. This mode should probably be used only to track down errors that normal mode does not reveal in sufficient detail.

The stack language produced in these different modes is compatible. Procedures compiled in different modes may call each other or be executed concurrently. The stack language interpreter "does not notice" the differences between code compiled in different modes until an error occurs. The following functions are provided to switch among the three modes:

STREAMLINE (STREAMLINE <proc1> <proc2> ... <procn>)

The <proci> are identifiers that are procedure names. Each of the <proci> is recompiled in streamlined mode. STREAMLINE returns the list of procedure names, (<proc1> ... <procn>). STREAMLINE does not evaluate its arguments.

UNSTREAMLINE

Called like STREAMLINE above, recompiles the <proci> in normal mode.

SLEUTH

Called like STREAMLINE above, recompiles the <proci> in sleuth mode.

UNSLEUTH

Called like STREAMLINE above, recompiles the <proci> in normal mode.

error messages

All errors can be controlled by the function ERRSET documented in [Bobrow et al] and [Meehan 79]. The Ski synmac ERRSET is modeled after the Ilisp function ERRSET. Its syntax is (ERRSET <exp> <flag>) where <flag> is a literal atom -- either T or NIL. If an error occurs anytime during the evaluation of the arbitrary Scheme or Lisp expression <exp> -- during the entire dynamic scope of the ERRSET expression -- then the evaluation of <exp> is abandoned and the call to ERRSET returns immediately. The value that ERRSET returns depends on the error. A programmer can use ERRSET to regain control of a computation when an error occurs, and to direct the flow of control in a computation according to which error occurs. If no error occurs during the evaluation of <exp>, then ERRSET returns the value of <exp> with an extra set of parentheses around it. By convention, the value returned by ERRSET when an error occurs is usually an atom, so it can be distinguished from a value returned when no error occurs.

If <flag> is NIL, then ERRSET returns immediately when an error occurs, with no user interaction. If <flag> is T, then a break is entered when an error occurs so that the user may examine the context of the error.

page 27. Illegal opcode. Lookup error. Wrong number of arguments.

When an error occurs during the execution of streamlined stack language, there is generally no way for the user to correct the state of the program and let the computation continue. There are two error types that the user can never correct within the context of the computation, even if the error occurs in stack language compiled in normal or sleuth mode. Both of these errors indicate that the stack language being executed or the virtual machine's registers have been altered (perhaps accidentally edited, or damaged by imprudent use or REPLACA or RPLACD). These errors also may indicate that the compiler is producing bad code.

1) "Illegal SKI opcode: <exp>"
This error occurs when the interpreter tries to execute an illegal stack language instruction, i.e., an opcode that is not one of the stack language opcodes. The interpreter's program counter has reached an opcode that is not an atom, or is an atom that does not have an INSTR property on its Ilisp property list. Some stack language has been damaged or the INSTR property has been removed from the property list of the opcode. NIL is returned to the nearest ERRSET or to the SKI top level.

2) "Look-up error in static environment."
The interpreter has tried to execute a variable reference that specified a location that does not exist in the static environment data structure. This error could result from damage to the contents of the ENV# register, to the compiler, or to some stack language, causing the interpreter to try to reference a location in the wrong environment. NIL is returned to the nearest ERRSET or to the SKI top level.

When any other error occurs during execution of stack language compiled in normal or sleuth mode, the Ski source language expression that is the (likely) cause of the error is made available for the user to edit. It is possible to correct and to continuation the computation from these errors. After any of the errors messages below is printed, the user is given instructions telling how to call the editor to edit the source-language expression responsible for the error. After the user edits the problematic expression, that expression is recompiled and the computation continues by executing the new stack language for the entire edited expression. If side effects in the user program prohibit re-evaluation of the expression, then the user should notice that fact and exit the break with the up-arrow command "↑" instead of the "OK" command.

If the user chooses not to try to recover from the error within the context of the error, she may type the up-arrow command to the break package. This will return control and the value NIL to the nearest ERRSET or to the SKI top level.

3) "Declare the number of arguments required by <subr or lsubr>. Type:
(PUTPROP (QUOTE <subr or lsubr>) <n> (QUOTE NUMBER-OF-ARGS))
where <n> is the number of arguments."
The SKI compiler must know how many arguments an Ilisp subr or lsubr takes. (SKI permits only lsubrs that take a specific number of arguments.) This information is kept on the property list of the name of the subr or lsubr under the property name NUMBER-OF-ARGS. For example, the following Ilisp statements correctly declare the number of arguments for the subrs CAR and CONS:

```
>(PUTPROP (QUOTE CAR) 1 (QUOTE NUMBER-OF-ARGS))
1
>(PUTPROP (QUOTE CONS) 2 (QUOTE NUMBER-OF-ARGS))
2
```

If a subr or lsubr does not have a NUMBER-OF-ARGS property, then this error occurs. The user can make the appropriate declarations with a PUTPROP as above and continue the computation with an "OK" command.

4) "Mismatch of parameters and arguments in <call>"
There are too many or too few arguments in a call to a def or Ilisp function. After the above diagnostic is printed, the user may edit the call or the function definition involved and let compilation continue.

5) "Compiler error in <exp>"
If a compiler routine is called to compile a malformed Scheme expression, e.g., "(LET)", "(LAMBDA A (ADD1 A))", then if you're lucky it will print this message. (If you're unlucky it will generate incorrect stack language.) Errors of this type may occur during execution of compiler routines, synmacs, or macros. After this message is printed, the user may edit the call or macro definitions involved, and then let compilation continue.

6) "<var> is an unbound Scheme variable."
This error can occur when a variable (or function name) has not been bound or set, or has been misspelled, or when parentheses in a Scheme expression are misplaced. The user is given the opportunity to edit the source-language call in which the variable appears. The spelling of the variable may be changed, the variable may be deleted, it may be changed to an expression, etc.

7) errors generated in Ilisp functions called from SKI
This Ski error occurs when evaluation of a call from SKI to an expr, subr, lexpr, lsubr, fexpr, or fsubr causes an Ilisp error. This error is first handled by the Ilisp interpreter and break package as an Ilisp error. In other words, unless an (ERRSET ... NIL) has been set, the broken function prints a diagnostic, a break occurs, and the error is handled in the break package by ILISP. After the user returns from that break with either an "OK" or "↑" command, another break occurs in which she can edit the Ski expression that called the troublesome Ilisp function. Execution continues after she exits from this break with an "OK" or "↑".

8) "Value in function position is not a closure.
Error with function: <exp> and arguments: <list>"
This error occurs when the function <exp> to be applied is not a closure, an object, or a continuation.

9) errors generated by Ski and Ilisp functions
ERR and ERROR are subrs that generate errors in Ski and Ilisp programs. ERR and ERROR are explained in detail in [Bobrow et al] or [Meehan 79]. The error values they return depend on their arguments. (ERR <exp>) returns the value of <exp> to the nearest ERRSET or to the SKI top level. It may or may not cause a break, depending on the value of <flag> in the nearest ERRSET. ERROR always causes a break in the computation.

A word about why and how ERR, ERROR, and ERRSET are implemented in SKI is in order here. Ilisp functions often use ERR, ERROR, and ERRSET to direct flow of control. Some Ilisp primitives, e.g., READ, TYI, INC, predictably generate errors when certain conditions, e.g., end of file, illegal dot notation, etc., occur. Because Ilisp functions are called from SKI, there are times when Ilisp errors propogate up into Scheme computations. We decided that ERRSET should be available in Ski procedures to catch, control, examine, and use Ilisp errors. The stack language interpreter intercepts Ilisp errors generated by Ilisp ERR or ERROR and converts them to Ski errors. Ski ERRSET intercepts Ski errors just as the Ilisp ERRSET intercepts Ilisp errors.

The fluid and global variable ERRPOINT is used by the interpreter to hold error return points set by ERRSET in Ski program. ERRPOINT should be treated as a reserved word.

in a break

The ILISP break package freezes or "breaks" the execution of an Ilisp program, typically when an error occurs or is generated by the program itself, and allows the programmer to examine the context of the error, to edit functions or values, and then either to supply an expression to be substituted as the result of the broken expression, or to return to some higher level in the computation. The ILISP break package is documented in [Bobrow et al] and [Meehan 79].

page 29. Exiting from a break. The read loop. The function SKI. Nested SKIs.

Any definitions, e.g., Scheme functions, magic word definitions, Ilisp functions, and Scheme or Ilisp values, may be edited in a break. Editing a macro or magic word definition DOES NOT automatically update already compiled Scheme procedures with references to that macro or magic word. Such Scheme procedures must be recompiled to expand the new macro definitions and generate new stack language. The easiest way to recompile a Scheme procedure is to call DEFINE with no body as explained in section B, "SPECIFICATION OF THE SKI LANGUAGE".

It is important to remember that when an error (either an Ilisp or Ski error) causes a break, the break package uses the ILISP interpreter and the Ilisp environments, not the SKI interpreter and Scheme environments.

There are several ways to exit from a break. The "OK" command causes SKI to try to continue the interrupted computation. The up-arrow command, "↑", generates an (ERR NIL) and bounces control out to the nearest ERRSET or to the SKI top level. For more information on the break package, see [Meehan 79].

the read loop and nested instantiations of SKI

One of the convenient features of ILISP is that the major subprograms of the ILISP interpreter support nested calls to each other and to themselves. For example, a user can find herself typing to an invocation of the ILISP editor which was called from a break which was called from a higher-level break which was called from the editor. As soon as an error occurs in the evaluation of some expression by ILISP or SKI (unless an (ERRSET ... NIL) is in effect), control is thrown to the break package where the programmer can examine the context of the error, edit functions, change values, and of course cause more errors!

SKI reflects this programming feature. The easiet way to invoke the SKI read-eval-print loop is to call the function SKI with no arguments. SKI's registers and flags (e.g., #CSTACK, #PCTRACE, #VALSTACK, etc.) are local to each invocation of SKI. The up-arrow character, typed as an atom to the read-eval-print loop, causes SKI to return. The value of SKI is always NIL. When SKI is called as a subroutine, its local registers and flags are reinitialized to default values. When an error occurs, you may enter a lower-level SKI, test functions, trace execution, etc. and then return to the upper level to continue the computation.

SKI may also be called as a subroutine without user interaction. SKI may take a series of Ski expressions as arguments, e.g.,

```
>(SKI (DEFINE NEQ (A B) (NOT (EQ A B)))
      (IF (NEQ (QUOTE THIS) (QUOTE THAT)) (PRINT (QUOTE NOT-EQUAL)))
      (IF (NEQ (QUOTE THIS) (QUOTE THIS)) (PRINT (QUOTE EQUAL)))
      ↑ )
NOT-EQUAL
NIL
>
```

This example calls SKI to define the function NEQ, to make two tests of it, and to return. If an up-arrow command is not present in the list of arguments in a call to SKI, that SKI invocation will remain in effect (will not return) until an up-arrow is typed interactively.

I. CLASSES AND OBJECTS

A central feature of Ski, not found in Steele and Sussman's Scheme, is the data-abstraction construct CLASS. The description of classes below on this page is based on [Wand 80a]. The remaining pages of this section develop a sample Ski program that uses classes and objects.

Ski classes are similar to those of Simula and Smalltalk, and we will use the Smalltalk terminology and say that an "object" is an "instance" of a "class". A class expression has this syntax:

```
(CLASS  <basis>
        (<msg1> <lambda exp1>)
        (<msg2> <lambda exp2>)
        ...
        (<msgn> <lambda expn>))
```

A class expression is a prototype for an object. An object is a procedural value, a kind of closed function. A class expression evaluates to an object. (This is similar to the relationship between lambda expressions and closures: A lambda expression is a prototype for a closure. A closure is a procedural value, a kind of closed function. A lambda expression evaluates to a closure.)

An object differs from a closure in that it may take different numbers of arguments, depending on the value of its first argument, whereas a closure always takes a fixed number of arguments, and a continuation always takes one argument. An object consists of a set of closures, the values of the <lambda expi>, indexed by the first argument in a call to the object. Following Smalltalk terminology, we call this first argument the "message". An object always takes at least one argument: a message. A message must be a Lisp atom.

The value of a class expression is a newly created object in which the <msgi> are bound to their corresponding closures. These closures result from closing (evaluating) the <lambda expi> in an environment where the identifier SELF is bound to the newly created object. This provides self-referential capability. An object is printed by SKI as "**OBJECT**".

The new object can take the messages <msg1> ... <msgn>. If the object is sent a message that is not one of these <msgi>, then an error will result, unless <basis> is not NIL. <basis> must evaluate to NIL or to an object. If <basis> evaluates to an object, the message-closure pairs in the value of <basis> are INHERITED by the new object, i.e., if the new object is called with a message that is not one of its <msgi>, then that message is matched against the messages in the value of <basis>, (and then against the messages of ITS basis, etc.). If no match is found, then an error results. Otherwise, the closure found is applied to the remaining arguments in the call. There must remain (after the message) exactly the right number of arguments in the call for the formal parameters of the closure. The value of the call to the object is the value of the call to the closure.

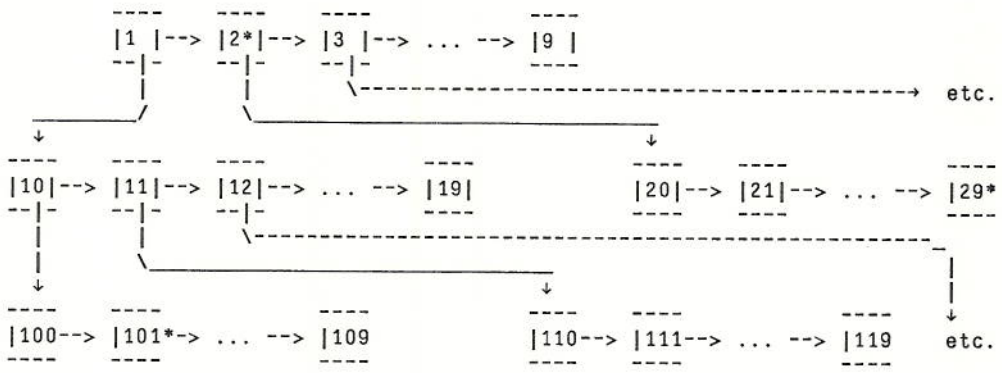
Let's state this explanation in another way: A class expression contains some function definitions named by atoms called "messages". An object contains -- has access to -- all of the functions from the class of which it is an instance. An object is called just like any other function; it is called in prefix functional notation:

```
(<object> <message> <arg1> ... <argn>)
```

<object> must evaluate to an object: it is typically a variable bound to an object. <message> must evaluate to an atomic message. There may be zero or more <argi>, arbitrary expressions that evaluate to the actual arguments. An object inherits all of the message-closure pairs contained in its basis, in its basis's basis, etc. So an object lies in an inheritance tree. If there is a duplication of messages in an object and its basis, the object's message takes precedence over the basis's message.

Classes and objects can be useful to Kim, our novice SKI programmer working on the 3n+1 problem. She realizes that much computation is duplicated when she computes, for example, (TNPO 5) and then (TNPO 10). The value of (TNPO 10) is (TNPO 5) + 1. If her program kept a record of the integers for which it had already computed the 3n+1 property, it could reuse that information when computing TNPO of other integers.

Kim needs a data structure in which to represent pairs of integers, (<key>, <answer>), where <key> is an integer and <answer> is the 3n+1 property of <key>. Because she wants to keep the data structure small, and because she wants to insert and retrieve the 3n+1 answers for arbitrary integer keys, she decides to use a Dewey-decimal tree structure. The root of the tree is 1 since (TNPO n) is defined only for n ≥ 1. Here is a picture of the inspiration for a data structure she is going to create. In this picture, the tree is infinitely deep. Level j contains all of the nodes for keys with j digits, and has 10 to the j nodes. The actual data structure that Kim will use will behave as if it were this infinite tree, but will consist only of one node for each (<key>, <answer>) pair that has been calculated by the program and inserted into the data structure.



In this infinite tree, you find the node for an integer <key> by finding the most significant digit of <key> in the top row, then if there are no more digits you're done, otherwise follow the arrow down a row from that spot and find its first two digits in that row, and so on. The node indexed by the integer 2 is starred in the top row; 29 is starred in the second row; 101 is starred in the third row. Notice that in this tree, the node for 2 is the root node for the subtree of all nodes whose keys have "2" as their most significant digit. We'll say that these keys are "prefixed" by 2. Similarly, the starred node in the third row is the root of the subtree for all nodes prefixed by 101, i.e., all nodes whose keys have "101" as their first three digits. The advantage of this tree is that a data structure representing it can be sparse. Only a small part of the tree needs to be represented in the data structure: one node for each key represented.

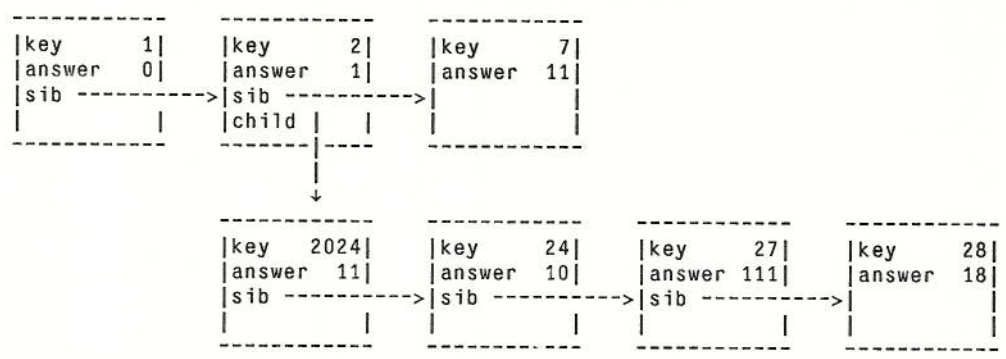
One of the reasons for data abstraction -- for classes and objects -- is to organize data into meaningful units that have named fields. Pascal records are an example of this kind of data structure. Kim decides that each node in the Dewey-decimal tree will need four fields:

key: the integer key in the domain of 3n+1 that this node represents;
answer: an integer in the range of 3n+1 -- the integer that is (3n+1 key);
sib: the node to the right, if any;
child: the node in the row below, if any.

Below is a picture of a tree containing the following information:

(TNPO 1) = 0
(TNPO 2) = 1
(TNPO 7) = 11
(TNPO 24) = 10
(TNPO 27) = 111
(TNPO 28) = 18
(TNPO 2024) = 11

page 32. Sample CLASS expression. Data protection. Calling objects.



Kim makes a first attempt at defining a class NODE with these four fields. Each box in the picture above will be a node, i.e., an instance of the class NODE.

```
>(DEFINE NODE (KEY ANSWER) (LET ((SIB NIL) (CHILD NIL)) (CLASS NIL)))
NODE
```

KEY and ANSWER are formal parameters. SIB and CHILD are private variables initialized to NIL. "(CLASS NIL)" defines an empty object with a null basis. Now Kim can make nodes -- objects that are instances of the class NODE -- but what good is an object with no message-function pairs in it? It can't DO anything! It has storage places for four data, bound to the variables KEY, ANSWER, SIB, and CHILD, but no way to access them, no way to store or retrieve or operate on them. KEY and ANSWER are bound automatically when NODE is called because they are the formal parameters to the def NODE. But there is no way to ask: "What is stored in the KEY field of this node?" and "What is stored in the ANSWER field of this node?". To do this, Kim adds messages to NODE that return the values of the fields. The convention she uses is that a named field is accessed by a message of the same name.

```
>(DEFINE NODE (KEY ANSWER)
  (LET ((SIB NIL) (CHILD NIL))
       (CLASS NIL
              (KEY      (LAMBDA NIL KEY))
              (ANSWER   (LAMBDA NIL ANSWER))
              (SIB      (LAMBDA NIL SIB))
              (CHILD    (LAMBDA NIL CHILD)))))
NODE
```

It may seem like busywork to have to write these field-accessing messages, but one of the uses of objects is to contain private variables -- variables that no outside procedure can access. This is called "data protection". It is easy to write a synmac to automatically provide these field-accessing messages, and the field-setting messages explained below, if you want them.

Notice that the variables KEY, ANSWER, SIB, and CHILD are not bound INSIDE the class expression. They don't need to be bound inside the class expression, and the syntax for CLASS does not provide a place for binding variables. Because the class expression is evaluated within the static scope of the variables KEY, ANSWER, SIB, and CHILD, the resulting object is closed in a private environment containing bindings for those variables.

Kim can now try out her NODE record structures:

```
>(ASETQ N (NODE 7 11))
**OBJECT**
>N
**OBJECT**
>(N (QUOTE KEY))
7
>(N (QUOTE ANSWER))
11
>(N (QUOTE SIB))
NIL
```

page 33. Sending messages to objects. SEND. Top-down vs. bottom-up.

The act of calling an object with a message and some arguments may also be stated as "sending a message to the object". We now define a synmac SEND, which lets us use what we feel is a more intuitive and readable syntax for calls to objects. We want to write:

```
(SEND N (KEY))
```

instead of:

```
(N (QUOTE KEY)).
```

```
>(DSYNM SEND Z (CONS (CADR Z) (CONS (LIST (QUOTE QUOTE) (CAADDR Z)) (CDADDR Z))))
SEND
```

This macro automatically quotes the message for us. We have found that we seldom write procedures that compute messages; we usually can write the literal message in the source-language call to an object. Kim and we will use this SEND syntax from here on, but remember that it is only a rearrangement of the more general syntax.

```
>(SEND N (KEY))
7
>(SEND N (ANSWER))
11
>(SEND N (CHILD))
NIL
```

The KEY and ANSWER fields of a node will never need to be reset. Once a node is linked into the tree in the proper place for its <key>, its KEY field shouldn't be changed. <answer> will be used to calculate the 3n+1 properties of other integers. A change to the ANSWER field would indicate that the old answer (or the new one) was a mistake, and that other 3n+1 calculations might be based on this mistake. So the KEY and ANSWER fields must be protected data; there should be no way to change them.

The SIB and CHILD fields may need to be reset many times as new nodes are inserted into the tree. Kim needs a way to make assignments to those fields. She adds two more messages to NODE for this purpose.

```
>(DEFINE NODE
         (KEY ANSWER)
         (LET ((SIB NIL) (CHILD NIL))
              (CLASS NIL
                     (KEY (LAMBDA NIL KEY))
                     (ANSWER (LAMBDA NIL ANSWER))
                     (SIB (LAMBDA NIL SIB))
                     (CHILD (LAMBDA NIL CHILD))
                     (SET-SIB (LAMBDA (X) (ASETQ SIB X)))
                     (SET-CHILD (LAMBDA (X) (ASETQ CHILD X))))))
NODE
```

Messages allow for varying degrees of data protection. In this definition of NODE, the KEY and ANSWER fields can never be changed because there are no messages that affect them and no way for an outside procedure to access the bindings. But the CHILD and SIB fields may be changed through the messages SET-CHILD and SET-SIB.

Kim still needs to write the procedures to organize nodes into a tree structure, to insert new key-answer pairs into it, and to retrieve answers for known keys. But it is not obvious how to do this. Should these procedures be messages in NODE? or external procedures? or part of the TNPO function? or what? So far, Kim has been taking the bottom-up approach to writing this data structure. She hasn't really thought about the operations to be performed on it -- inserting and retrieving key-answer pairs. She now leaves the definition of NODE for a while and takes the top-down approach to the problem: What does she want the data structure to do? What operations are to be performed on it? The top-down, data-abstraction approach to programming dictates that she should identify those operations as messages to be sent to the data structure.

page 34. Abstract data structure. TNPO-MAKER. Sample call to OR.

Kim defines TNPO-MAKER to return a procedure that has in its private environment a tree data structure in which will be stored the 3n+1 values that it calculates. The value of a call to TNPO-MAKER is a procedure that may be bound to a variable, may be called many times to compute the 3n+1 property of different integers, and will "remember" those integers and their 3n+1 properties so that it can use them to respond to subsequent calls without recalculating them.

```
>(DEFINE
  TNPO-MAKER
  NIL
  (LET ((PAIRS (PAIR-KEEPER)))
       (LABELS
        ((CALCULATE
          (LAMBDA(N A)
           (IF (EQ N 1)
               A
               (TEST
                (SEND PAIRS (LOOKUP N))
                (LAMBDA(X) (PLUS A X))
                (CALCULATE (IF (EVENP N) (QUOTIENT N 2) (ADD1 (TIMES N 3)))
                           (ADD1 A)))))))
         (LAMBDA(K)
          (OR (SEND PAIRS (LOOKUP K))
              (SEND PAIRS (INSERT K (CALCULATE K 0))))))))
TNPO-MAKER
```

The function TEST in this definition of TNPO-MAKER is a Scheme primitive (see section D, "SPECIFICATION OF THE SKI LANGUAGE"). DEFINE, LET, LABELS, LAMBDA, IF, EQ, PLUS, QUOTIENT, ADD1, TIMES, and OR are also primitives.

The variable PAIRS is bound to a data structure, returned by the function PAIR-KEEPER, that stores pairs of integers (<key>, <answer>). On each iteration, CALCULATE checks to see if its N parameter is represented in this pair-keeper. If so, it can return an answer immediately. Otherwise it keeps iterating.

Kim has not yet written the function PAIR-KEEPER or the definitions for the messages LOOKUP and INSERT. This approach is typical of top-down programming. As new procedures are needed to solve parts of the programming problem, the programmer ASSUMES that they exist, and writes them later. This definition of TNPO-MAKER does not concern itself with HOW the (<key>, <answer>) pairs are stored. It does not depend on their being stored in any particular way: in a Dewey-decimal tree, an array, a sorted list, or whatever. It ASSUMES the existence of an object, returned by the function PAIR-KEEPER, that takes two messages, LOOKUP and INSERT.

LOOKUP:

(SEND <pair-keeper> (LOOKUP <key>))

returns the 3n+1 property of <key> if it is represented in the data structure, and returns NIL if it is not represented.

INSERT:

(SEND <pair-keeper> (INSERT <key> <answer>))

inserts the pair (<key>, <answer>) into the data structure and returns <answer>.

The Ski primitive OR is used in TNPO-MAKER as a sequencing and conditional operator. OR evaluates its arguments sequentially and returns the first non-NIL value. Here it first evaluates "(SEND PAIRS (LOOKUP K))". If this returns a non-NIL value, then that value is immediately returned as the value of the OR form, and the second argument to OR is not evaluated. Otherwise the 3n+1 property of K is calculated by calling CALCULATE, and is inserted into the data structure by sending the message INSERT.

page 35. Sample program with classes and objects, continued.

Kim now has to write PAIR-KEEPER. PAIR-KEEPER must return an object that takes the two messages LOOKUP and INSERT, and must contain any procedures necessary to implement those messages. Kim has already decided to implement the pair-keeper as a Dewey-decimal tree. She continues using the top-down approach, solving the programming problem a little at a time. Here is her entire definition for PAIR-KEEPER. We motivate and explain it piece by piece below, but notice here its overall "shape". It binds the variable HEADER to a root node for the tree, i.e., to a node for the (<key>, <answer>) pair (1, 0). It contains the messages LOOKUP and INSERT. And it introduces the new, as yet undefined procedures and messages TREE-NODE, AT-OR-JUST-ABOVE, PREFIXP, and SPLIT.

```
>(DEFINE
  PAIR-KEEPER
  NIL
  (LET ((HEADER (TREE-NODE 1 0)))
       (CLASS
        NIL
        (LOOKUP
         (LAMBDA(K)
          (LET ((FOUND (SEND HEADER (AT-OR-JUST-ABOVE K))))
               (IF (EQ (SEND FOUND (KEY)) K) (SEND FOUND (ANSWER))))))
        (INSERT
         (LAMBDA(K V)
          (LET ((FOUND (SEND HEADER (AT-OR-JUST-ABOVE K))) (N (TREE-NODE K V)))
               (COND
                ((EQ K (SEND FOUND (KEY))) (ERR (QUOTE CAN'T-INSERT-KEY-TWICE)))
                ((PREFIXP (SEND FOUND (KEY)) K)
                 (LET ((S (SPLIT (SEND FOUND (CHILD)) K)))
                      (BLOCK (SEND N (SET-CHILD (CAR S)))
                             (SEND N (SET-SIB (CDR S)))
                             (SEND FOUND (SET-CHILD N))
                             V)))
                (T
                 (LET ((S (SPLIT (SEND FOUND (SIB)) K)))
                      (BLOCK (SEND N (SET-CHILD (CAR S)))
                             (SEND N (SET-SIB (CDR S)))
                             (SEND FOUND (SET-SIB N))
                             V)))))))))
PAIR-KEEPER
```

Here we insert commentary after sections of the definition of PAIR-KEEPER:

```
 (DEFINE
  PAIR-KEEPER
  NIL
  (LET ((HEADER (TREE-NODE 1 0)))
       (CLASS
        NIL
```

PAIR-KEEPER binds the variable HEADER to the node that will be the root of the Dewey-decimal tree. Nodes in this tree will be similar to instances of the class NODE which Kim defined earlier. Each node will contain the fields KEY, ANSWER, SIB, and CHILD, but will also contain procedures for traversing the tree structure. Kim hasn't written those procedures yet.

```
        (LOOKUP
         (LAMBDA(K)
          (LET ((FOUND (SEND HEADER (AT-OR-JUST-ABOVE K))))
               (IF (EQ (SEND FOUND (KEY)) K) (SEND FOUND (ANSWER))))))
```

Execution of the message LOOKUP first sends the message "(AT-OR-JUST-ABOVE K)" to HEADER. K is an integer key. The value of the message AT-OR-JUST-ABOVE should be the node in the tree that is "at or just above" the node for K. We'll define "at or just above" soon. The point here is that AT-OR-JUST-ABOVE always returns a node as its value. This node is bound to the variable FOUND. In the definition of LOOKUP, if the KEY field of the node FOUND is equal to K, then the ANSWER field of FOUND is returned as the 3n+1 property of K. Otherwise NIL is returned.

page 36. Sample program with classes and objects, continued.

```
        (INSERT
         (LAMBDA(K V)
          (LET ((FOUND (SEND HEADER (AT-OR-JUST-ABOVE K))) (N (TREE-NODE K V)))
               (COND
                ((EQ K (SEND FOUND (KEY))) (ERR (QUOTE CAN'T-INSERT-KEY-TWICE)))
                ((PREFIXP (SEND FOUND (KEY)) K)
                 (LET ((S (SPLIT (SEND FOUND (CHILD)) K)))
                      (BLOCK (SEND N (SET-CHILD (CAR S)))
                             (SEND N (SET-SIB (CDR S)))
                             (SEND FOUND (SET-CHILD N))
                             V)))
                (T (LET ((S (SPLIT (SEND FOUND (SIB)) K)))
                        (BLOCK (SEND N (SET-CHILD (CAR S)))
                               (SEND N (SET-SIB (CDR S)))
                               (SEND FOUND (SET-SIB N))
                               V)))))))))))
```

Execution of the message INSERT also sends the message "(AT-OR-JUST-ABOVE K)" to HEADER and binds the result to the variable FOUND. This guarantees that FOUND is the node that is "at or just above" K. A node is "at or just above" K when its KEY field is the lexically greatest KEY field in the tree that is lexically less than or equal to K. ("Lexically greater" and "lexically less" have nothing to do with "lexical" (static) binding.) Once this node is found, its KEY field may be:

a) equal to K; or
b) not equal to K, but a prefix of K; or
c) neither equal to nor a prefix of K, but lexically less than K.

These three cases are checked in order in the three COND clauses in INSERT.

a) When the KEY field of FOUND is equal to K, then FOUND is the node for K, and its ANSWER field contains the 3n+1 property of K.

b) A number is a prefix of another number when, viewing both as strings of digits, the first is a string of digits at the front of the other. For example, 10 is a prefix of 101 because the string of digits "10" is a string of digits at the front of "101". Similarly, "2" is a prefix of "2024".

c) A string of characters is "lexically less than" another string of characters if it would appear before the other in the dictionary. For example, "an" appears before "another", and "cat" appears before "dog". When we extend this to strings of digits, "2024" appears before "24", and "24" appears before "5".

Since FOUND is bound to the node "at or just above" K, this guarantees either that FOUND is the node for K, or that a new node for K should be inserted in the tree just to the right of FOUND or just below FOUND. This new node for the (<key>, <answer>) pair (K, V) is created and bound to the variable N.

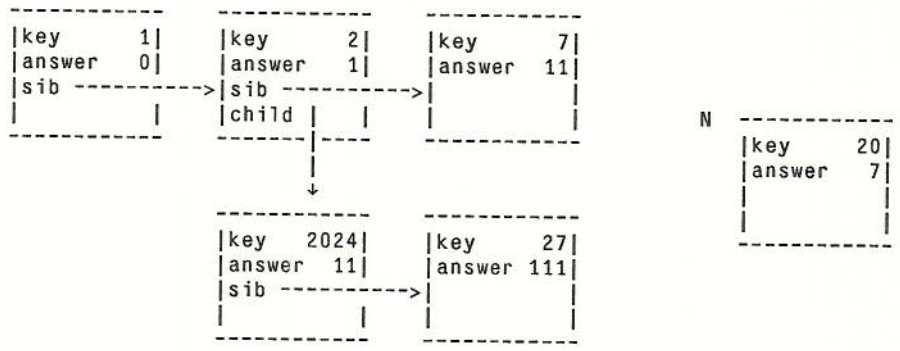
a)
```
((EQ K (SEND FOUND (KEY))) (ERR (QUOTE CAN'T-INSERT-KEY-TWICE)))
```

In this case, INSERT is being called to insert a (<key>, <answer>) pair that is already in the tree. This is an error condition.

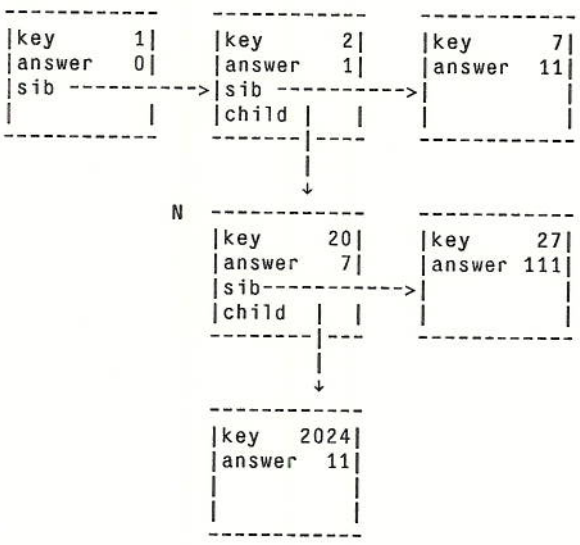
b)
```
((PREFIXP (SEND FOUND (KEY)) K)
 (LET ((S (SPLIT (SEND FOUND (CHILD)) K)))
      (BLOCK (SEND N (SET-CHILD (CAR S)))
             (SEND N (SET-SIB (CDR S)))
             (SEND FOUND (SET-CHILD N))
             V)))
```

In this case, (the value of the KEY field of) FOUND is a prefix of (the value of) K. So, the node for K must be inserted just below FOUND. Here is an example of such a situation. In the initial configuation of this tree, nodes for 1, 2, 7, 2024, and 27 are represented. A new node N has been created for the pair (20, 7), but has not yet been linked into the tree.

page 37. Sample program with classes and objects, continued.



We want to insert node N into the structure. There is no node for 20 in the tree (case a); but "2" is a prefix of "20" (case b), so FOUND is bound to the node for 2. The new node N must be inserted just below FOUND. Part of the subtree that is already below FOUND should be stored in the SIB field of N, and part of it should go in the CHILD field of N. That part of the subtree in which "20" is a prefix should go below the new node. The rest should go to the right of the new node.



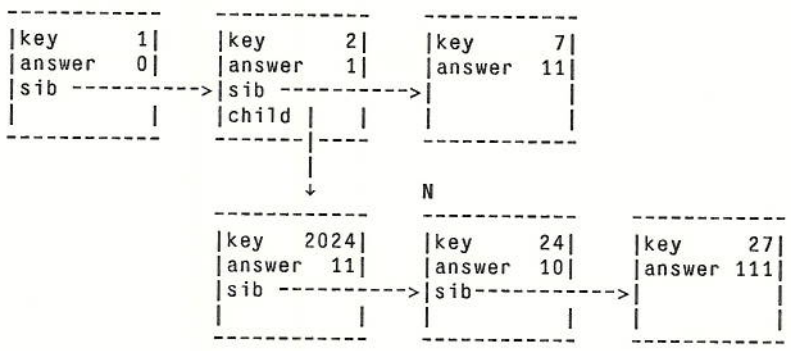
The subtree below FOUND is broken into two pieces by the procedure SPLIT. SPLIT takes two arguments: a subtree and an integer. SPLIT returns a dotted pair consisting of the part of the subtree in which the integer is a prefix of the KEY fields, and the part in which it is not. The subtree always breaks neatly in two, because at most one break is needed, and that break will always be found along the top row of the subtree. In this example, "20" is a prefix of "2024", so the subtree with 2024 at its root should be stored in the CHILD field of N. "20" is not a prefix of "27", so the subtree with 27 at its root will be stored in the SIB field of N. The break occurrs between the nodes for 2024 and 27.

Finally, the new node N is linked into the tree below FOUND. Then 7, the answer which is bound to V, is returned as the value of the message INSERT.

c)

```
(T
 (LET ((S (SPLIT (SEND FOUND (SIB)) K)))
      (BLOCK (SEND N (SET-CHILD (CAR S)))
             (SEND N (SET-SIB (CDR S)))
             (SEND FOUND (SET-SIB N))
             V))))))))))
```

In this case, the new node N should be inserted to the right of FOUND. To demonstrate this, let's start with the same initial configuration as in the example in case (b). This time, we want to insert a node N for the pair (24, 10). The message (AT-OR-JUST-ABOVE 24) returns the node for 2024, because 2024 is the lexically greatest key that is lexically less than 24. But because "24" is not a prefix of "2024", the new node should be inserted to the right of 2024. Part of the subtree to the right of FOUND should be stored in the CHILD field of N, and part of it into the SIB field. The procedure SPLIT is called to break the subtree into two pieces. Again, that part of the subtree in which 24 is a prefix is stored in the CHILD field of N, and the rest is stored in the SIB field. In this example, that part of the subtree in which 24 is a prefix is empty, so NIL is stored in the CHILD field of N. Either or both of these parts may be empty subtrees (may be NIL) in some configurations.



Kim now takes stock of her progress. She still has three procedures to write: PREFIXP, SPLIT, and TREE-NODE. The first two are straightforward recursive Scheme functions. The last will be a modification of her definition of NODE.

(PREFIXP <m> <n>) returns T if <m> is a prefix of <n>;
(SPLIT <tree> <key>) splits <tree> into two pieces; and
TREE-NODE returns nodes that take the message AT-OR-JUST-ABOVE.

```
>(DEFINE PREFIXP
         (M N)
         (LABELS ((F (LAMBDA(A B)
                     (COND ((NULL A) T)
                           ((NULL B) NIL)
                           ((EQ (CAR A) (CAR B)) (F (CDR A) (CDR B)))
                           (T NIL)))))
                 (F (EXPLODE M) (EXPLODE N))))
PREFIXP
>(PREFIXP 20 2024)
T
>(PREFIXP 24 2024)
NIL
```

The Ilisp primitive EXPLODE, used in PREFIXP, takes an s-expression as its argument and returns the list of the characters of the printed representation of that s-expression. It "explodes" an atom into a list. For example, (EXPLODE (QUOTE RAT)) = (R A T), and (EXPLODE 2024) = (2 0 2 4).

```
>(DEFINE SPLIT (TREE KEY)
         (LABELS
          ((F
            (LAMBDA(P Q K)
             (IF (AND Q (PREFIXP K (SEND Q (KEY))))
                 (F Q (SEND Q (SIB)) K)
                 (IF P
                     (BLOCK (SEND P (SET-SIB NIL)) (CONS TREE Q))
                     (CONS NIL Q))))))
          (F NIL TREE KEY)))
SPLIT
```

page 39. Sample program with classes and objects, continued.

SPLIT returns a dotted pair of two subtrees, either or both of which may be NIL.

Kim's last remaining job is to write the procedure TREE-NODE, which, like the procedure NODE that she has already defined, must return nodes with KEY, ANSWER, SIB, and CHILD fields. But a tree-node must also take the message AT-OR-JUST-ABOVE as explained above. Kim decides that an instance of the class TREE-NODE should have as its <basis> an instance of the class NODE. This means that an instance of TREE-NODE will inherit the messages KEY, ANSWER, SIB, CHILD, SET-SIB, and SET-CHILD from the class NODE. The help function (LEXLEQP <m> <n>) in TREE-NODE returns T if <m> is lexically less than or equal to <n>.

```
>(DEFINE
  TREE-NODE
  (KEY ANSWER)
   (CLASS
    (NODE KEY ANSWER)
    (AT-OR-JUST-ABOVE
     (LAMBDA(K)
      (COND
       ((AND (SEND SELF (SIB)) (LEXLEQP (SEND (SEND SELF (SIB)) (KEY)) K))
        (SEND (SEND SELF (SIB)) (AT-OR-JUST-ABOVE K)))
       ((AND (SEND SELF (CHILD)) (PREFIXP (SEND (SEND SELF (CHILD)) (KEY)) K))
        (SEND (SEND SELF (CHILD)) (AT-OR-JUST-ABOVE K)))
       (T SELF)))))))
TREE-NODE
```

Each tree-node has as its basis a private node -- an instance of NODE that is not the basis of any other object -- created by the call "(NODE KEY ANSWER)".

The message AT-OR-JUST-ABOVE tail-recursively traverses the tree. All instances of TREE-NODE share access to the message AT-OR-JUST-ABOVE, so one tree-node can send the message to another. When the search reaches a leaf of the tree or the node for the lexically greatest KEY in the tree that is lexically less than or equal to K, that node is returned.

The body of AT-OR-JUST-ABOVE contains many nested SEND commands. The nested message calls have been removed from the following version which uses the implicit binding of non-NULL predicate values by the Scheme primitive TEST. If the contents of a SIB or CHILD field is not NIL, it must be a node. The consequents in the calls to TEST in the body of AT-OR-JUST-ABOVE are functions that send the messages KEY and AT-OR-JUST-ABOVE to the implicitly bound nodes to retrieve their KEY fields and to continue the tail-recursive search of the tree.

```
(DEFINE
  TREE-NODE
  (KEY ANSWER)
   (CLASS
    (NODE KEY ANSWER)
    (AT-OR-JUST-ABOVE
     (LAMBDA(K)
      (TEST
       (OR (TEST (SEND SELF (SIB))
                 (LAMBDA (S) (IF (LEXLEQP (SEND S (KEY)) K) S))
                 NIL)
           (TEST (SEND SELF (CHILD))
                 (LAMBDA (C) (IF (PREFIXP (SEND C (KEY)) K) C))
                 NIL))
       (LAMBDA (N) (SEND N (AT-OR-JUST-ABOVE K)))
       SELF))))))
```

Kim next writes the predicate LEXLEQP and inserts it into the definition of TREE-NODE, moves SPLIT into PAIR-KEEPER, and pretty-prints all of her definitions:

page 40. Sample program with classes and objects, continued.

```
>(PP TNPO-MAKER NODE TREE-NODE PREFIXP PAIR-KEEPER)

(DEFINE
 TNPO-MAKER
 NIL
 (LET ((PAIRS (PAIR-KEEPER)))
      (LABELS
       ((CALCULATE
         (LAMBDA(N A)
          (IF (EQ N 1)
              A
              (TEST
               (SEND PAIRS (LOOKUP N))
               (LAMBDA(X) (PLUS A X))
               (CALCULATE (IF (EVENP N) (QUOTIENT N 2) (ADD1 (TIMES N 3)))
                          (ADD1 A)))))))
        (LAMBDA(K)
         (OR (SEND PAIRS (LOOKUP K))
             (SEND PAIRS (INSERT K (CALCULATE K 0))))))))

(DEFINE NODE
        (KEY ANSWER)
        (LET ((SIB NIL) (CHILD NIL))
             (CLASS NIL
                    (KEY (LAMBDA NIL KEY))
                    (ANSWER (LAMBDA NIL ANSWER))
                    (SIB (LAMBDA NIL SIB))
                    (CHILD (LAMBDA NIL CHILD))
                    (SET-SIB (LAMBDA (X) (ASETQ SIB X)))
                    (SET-CHILD (LAMBDA (X) (ASETQ CHILD X))))))

(DEFINE
 TREE-NODE
 (KEY ANSWER)
 (LET ((LEXLEQP
        (LAMBDA(X Y)
         (LABELS
          ((F (LAMBDA(A B)
               (COND ((NULL A) T)
                     ((NULL B) NIL)
                     ((LESSP (CAR A) (CAR B)) T)
                     ((EQ (CAR A) (CAR B)) (F (CDR A) (CDR B)))))))
          (F (EXPLODE X) (EXPLODE Y))))))
      (CLASS
       (NODE KEY ANSWER)
       (AT-OR-JUST-ABOVE
        (LAMBDA(K)
         (TEST
          (OR (TEST (SEND SELF (SIB))
                    (LAMBDA (S) (IF (LEXLEQP (SEND S (KEY)) K) S))
                    NIL)
              (TEST (SEND SELF (CHILD))
                    (LAMBDA (C) (IF (PREFIXP (SEND C (KEY)) K) C))
                    NIL))
          (LAMBDA (N) (SEND N (AT-OR-JUST-ABOVE K)))
          SELF))))))

(DEFINE PREFIXP
        (M N)
        (LABELS ((F
                  (LAMBDA(A B)
                   (COND ((NULL A) T)
                         ((NULL B) NIL)
                         ((EQ (CAR A) (CAR B)) (F (CDR A) (CDR B)))))))
                 (F (EXPLODE M) (EXPLODE N))))
```

page 41. Sample program with classes and objects, continued.

```
(DEFINE
 PAIR-KEEPER
 NIL
 (LET ((HEADER (TREE-NODE 1 0))
       (SPLIT
        (LAMBDA(TREE KEY)
         (LABELS
          ((F
            (LAMBDA(P Q K)
             (IF (AND Q (PREFIXP K (SEND Q (KEY))))
                 (F Q (SEND Q (SIB)) K)
                 (IF P
                     (BLOCK (SEND P (SET-SIB NIL)) (CONS TREE Q))
                     (CONS NIL Q))))))
           (F NIL TREE KEY)))))
      (CLASS
       NIL
       (LOOKUP
        (LAMBDA(K)
         (LET ((FOUND (SEND HEADER (AT-OR-JUST-ABOVE K))))
              (IF (EQ (SEND FOUND (KEY)) K) (SEND FOUND (ANSWER))))))
       (INSERT
        (LAMBDA(K V)
         (LET ((FOUND (SEND HEADER (AT-OR-JUST-ABOVE K))) (N (TREE-NODE K V)))
              (COND
               ((EQ K (SEND FOUND (KEY))) (ERR (QUOTE CAN'T-INSERT-KEY-TWICE)))
               ((PREFIXP (SEND FOUND (KEY)) K)
                (LET ((S (SPLIT (SEND FOUND (CHILD)) K)))
                     (BLOCK (SEND N (SET-CHILD (CAR S)))
                            (SEND N (SET-SIB (CDR S)))
                            (SEND FOUND (SET-CHILD N))
                            V)))
               (T
                (LET ((S (SPLIT (SEND FOUND (SIB)) K)))
                     (BLOCK (SEND N (SET-CHILD (CAR S)))
                            (SEND N (SET-SIB (CDR S)))
                            (SEND FOUND (SET-SIB N))
                            V)))))))))

NIL
>
```

Kim is now ready to try out her new definitions. The inner workings of the data structure that holds the computed pairs (<key>, <answer>) are hidden from view. Kim has only to call TNPO-MAKER to create a procedure that will compute and store pairs of 3n+1 integers. Then she can call that procedure again and again to compute 3n+1 for different integers. The data structure is completely secure. There is no way to access the data structure except through that procedure.

```
>(ASETQ ORACLE (TNPO-MAKER))
**OBJECT**
>(ORACLE 3)
7
>(ORACLE 27)
111
>
```

It works, but Kim wants to prove to herself that previously computed results are really being stored in the Dewey-decimal tree and reused to compute new values. She puts a print statement into the definition of CALCULATE to demonstrate whether or not the tree data-structure is really being used. Here is TNPO-MAKER after she edits it. Notice the print statement in the TEST form.

page 42. Sample program with classes and objects, continued.

```
>(PP TNPO-MAKER)

(DEFINE
 TNPO-MAKER
 NIL
 (LET ((PAIRS (PAIR-KEEPER)))
      (LABELS
       ((CALCULATE
         (LAMBDA(N A)
          (IF (EQ N 1)
              A
              (TEST
               (SEND PAIRS (LOOKUP N))
               (LAMBDA(X)
                (BLOCK (PRINT (LIST (QUOTE EUREKA) X)) (PLUS A X)))
               (CALCULATE (IF (EVENP N) (QUOTIENT N 2) (ADD1 (TIMES N 3)))
                          (ADD1 A)))))))
       (LAMBDA(K)
        (OR (SEND PAIRS (LOOKUP K))
            (SEND PAIRS (INSERT K (CALCULATE K 0))))))))

NIL
>
```

Now she has to start over with a new procedure created by TNPO-MAKER that does not yet contain any (<key>, <answer>) pairs, (except for the pair (1, 0) in its header node).

```
>(ASETQ ORACLE (TNPO-MAKER))
**OBJECT**
>(ORACLE 3)
7
>(ORACLE 3)
(EUREKA 7)
7
>(ORACLE 12)
(EUREKA 7)
9
>
```

The first call above, (ORACLE 3), iterated all the way down to 1 because there were no answers stored in the tree. After the 3n+1 property of 3 was computed, it was stored in the tree. The second call, (ORACLE 3), returned this stored value 7 immediately. The last call, (ORACLE 12), iterated only until it found a known value, the 3n+1 property of 3, and returned without iterating further.

J. EDITING A FILE

The SKI file editor creates and modifies files that contain definitions and other s-expressions. It is not a general-purpose file-editing program, but it performs many operations on files that LISP or SCHEME programmers will probably find helpful. It is meant to create and modify files that will be read by SKI or ILISP. A file is represented in SKI by a FILE DEFINITION. A file definition is stored on the property list of the file name under the property name FILE. Ski file names differ in syntax from Ilisp file names. An Ilisp file name either is an atom, or is a dotted pair of two atoms -- a file name <file> and its extension <ext>, CONSed together to form: (<file> . <ext>). SKI file names are always atoms -- even file names with extensions or directory specifications. The parts of a file name are separated by a dash. The parts of a directory specification are also separated by dashes. The file name proper and the directory specification are separated by the character "@". A SKI file name is one of the following:

a) <file>

b) <file>-<ext>

c) <file>-<project>@<programmer>

d) <file>-<ext>@<project>-<programmer>

(At SAIL, substitute <directory> for <project>.) Examples of each syntax:

a) FOLLY

b) FILHAN-IL

c) HELP@50304-123

d) SKI-MAN@1-GAC

The file editor uses the device "DSK:" in all file specifications. Once you refer to a file in one of these formats, you should stick to that format because information about the file is stored on the property list of that identifier. All of the file-editing functions use file names in this single-identifier, dashed format. (However, Ilisp primitive functions for file handling, e.g., DSKIN, LOOKUP, etc., require file names with extensions in the dotted format, and directory and device information as explained in the Ilisp documentation.)

A file definition consists of the atom FILE, the name of the file, and some page lists. Each page list is a list beginning with the atom PAGE and then any number of s-expressions that appear on that page of the file. A definition that appears on a page is abbreviated to a list of two things: the definer for that definition and the name of the definition, e.g., (DEFINE TNPO-HELP), (DE EVENP). Other s-expressions on the page are not abbreviated. This page list:

```
(PAGE   (DE EVENP)
        (ASETQ NUMS (QUOTE (3 5 20 40)))
        (AMAPCAR TNPO NUMS))
```

indicates a page which contains a definition of the expr EVENP, and two lists, "(ASETQ NUMS (QUOTE (3 5 20 40)))" and "(AMAPCAR TNPO NUMS)". The following file definition defines a file with two pages.

```
>(FILE TNPO-KIM
       (PAGE (DEFINE TNPO) (DEFINE TNPO-HELP))
       (PAGE (DE EVENP)
             (ASETQ NUMS (QUOTE (3 5 20 40)))
             (AMAPCAR TNPO NUMS)))
TNPO-KIM
```

When this file, TNPO-KIM, is read into SKI by GETFILE (described below) it will define the functions TNPO, TNPO-HELP, and EVENP, set NUMS to a list of integers, and then call TNPO on those numbers.

page 44. What files can be edited. Reading files. GETFILE, GETFILEQ, GETDEFS.

Definitions on files may be defs, synmacs, exprs, macros, fexprs, or values. A user may add new definition types as explained in section N, "CUSTOMIZED DEFINITION TYPES"; then these new types may be written onto files, too. The file editor does not write file definitions onto files. File definitions must be created anew for each SKI session. Vals are not written onto files because they often contain circular list structures which cannot be printed.

Not all files can be modified with the file editor. Only a file that is LISP-readable can be edited. Arbitrary LISP-readable expressions (e.g., data, Lap code, etc.) may occur in files to be edited.

File-editing functions can be divided into two groups, those that read files or parts of files, and those that write files. There are functions for reading files in various ways: as text, as unevaluated expressions, and as expressions to evaluate. File writing is done by editing a file definition. Once you exit the editor, the file is rewritten automatically.

reading files

The operations that read files are:

GETFILE	allows a file to be read by a read loop.
GETFILEQ	returns a list of all expressions on a file, unevaluated.
GETDEFS	finds definitions by name on a file; reads and evaluates them.
GETDEFSQ	finds definitions by name on a file and returns them unevaluated.
TY	types specified pages of a file to the terminal screen.
INVENTORY	reads the file and creates a new file definition for it.
CONTENTS	returns the current file definition of a file.
LOOKUPFILE	tells whether there exists a file with given name

GETFILE (GETFILE <file1> <file2> ... <filen>)
When called from the SKI or ILISP top level, GETFILE causes the <filei> to be read. It is the best function to use to read files from the SKI top level, or from any read loop. GETFILE directs ILISP's input channel to each of the <filei> in turn. GETFILE returns the previously open input channel. If any of the <filei> does not exist, an error is generated. GETFILE does not evaluate its arguments.

GETFILELX
same as GETFILE but takes only one argument, a list of files to read. GETFILELX evaluates its argument.

GETFILEQ (GETFILEQ <file>)
GETFILEQ returns a list of all s-expressions on <file> -- that is, the entire contents of the file -- unevaluated. GETFILEQ does not evaluate its argument. If <file> does not exist, an error is generated. GETFILEQ is useful for manipulating the contents of a file without evaluating it. For example, there may be a file of errorful definitions that you want to edit, but that you don't want to evaluate BECAUSE they are faulty; you can set some variable to the list of the contents of the file, and edit the structures there. GETFILEQ lets you read a file without evaluating its contents.

GETFILEQX
same as GETFILEQ but does evaluate its argument.

GETDEFS (GETDEFS <file> <f1> <f2> ... <fn>)
where <fi> is either an identifier: <id>,
or a list or a definer and an identifier: (<definer> <id>).

sample calls:

(GETDEFS TNPO-KIM (DEFINE TNPO) (DE EVENP))
(GETDEFS NGTF GRNL)

GETDEFS retrieves the specified definitions from <file> and calls ILISP'S EVAL to evaluate them. If <fi> is an identifier, like GRNL in the second sample call above, then the first definition of <fi> found on <file> is read and evaluated. This is useful when there is only one definition with name <fi> on <file> (probably the most common case). If <fi> is a list of a definer and an identifier, only a definition of the particular type indicated by <definer> will be retrieved. GETDEFS does not evaluate its arguments. It returns a list of the values of the defining expressions found on <file> -- usually a list of the names of the definitions. In the examples above, GETDEFS might return "(TNPO TNPO (EVENP REDEFINED))" and "(GRNL)" respectively. If the definition of any <fi> is not found on <file>, then that definition is not retrieved and its value does not appear in the value of GETDEFS, but no error is generated.

GETDEFSLX
same as GETDEFS but does evaluate its two arguments: a file name and a list of the <fi> to be retrieved from that file.

GETDEFSQ (GETDEFSQ <file> <f1> <f2> ... <fn>)
where <fi> is either an identifier: <id>,
or a list or a definer and an identifier: (<definer> <id>).

sample calls:

```
(GETDEFSQ TNPO-KIM (DEFINE TNPO) (DE EVENP))
(GETDEFSQ NGTF GRNL)
```

GETDEFSQ finds the definitions of the <fi> on <file> as explained in GETDEFS, but returns the list of the unevaluated definitions -- the actual s-expressions found on the file. If some of the <fi> are not found on <file>, their defining expressions are missing from the value of GETDEFSQ. GETDEFSQ does not evaluate its arguments.

GETDEFSQLX
same as GETDEFSQ, but takes two arguments -- a file name and a list of the <fi> to be retrieved from that file -- and evaluates them.

TY (TY <file and pages> <file and pages> ... <file and pages>)
where <file and pages> is either a file name: <file>;
or a list of a file name and some page numbers: (<file> <p1> <p2> ... <pn>).

TY types specified pages of specified files to the terminal screen, pausing after each screenful to let the user cancel file viewing. It reads a file one character at a time, so it can type any ASCII file to the screen; a file need not contain only legal Lisp s-expressions to be TYed. TY looks for page marks and prints an extra message line when it reads a page mark (ascii 12 decimal). The message tells which page of what file is about to be typed out. TY uses the global Ilisp variable SCREENSIZE, whose value should be 2 or 3 smaller than the number of lines that fit on the terminal screen. TY pauses after every SCREENSIZE lines and asks the user if it should continue typing or should stop and return.

TY allows one to specify certain pages of a file to be typed. TY takes any number of arguments, each of which specifies a file and the pages of that file to be typed. Each specification is either the name of a file or is a list of the name of a file and some integer page numbers. The page numbers need not be in order. Typically, few pages of a file will be requested, because any serious file browsing should be done with a screen-oriented editor. If no page numbers are present, the entire file is typed. Sample calls to TY:

```
(TY NGTF)
(TY (SKI-MAN@1-GAC 4 11) (TNPO-KIM 2))
```

TY does not evaluate its arguments. TY returns NIL. If one of the files specified does not exist, an error occurs, but references to nonexistent pages are ignored.

TYLX
same as TY, but takes 1 argument: a list of file and page specifications. TYLX evaluates its argument.

INVENTORY (INVENTORY <file>)
<file> must be the name of a file. INVENTORY creates a file definition for <file> by reading <file> and classifying each s-expression it contains as a definition or as an arbitrary s-expression to be copied literally. If <file> does not exist, an error is generated. INVENTORY does not evaluate its argument. INVENTORY updates the FILE property of <file> to the new file definition and returns the new file definition as its value.

INVENTORYX
same as INVENTORY but evaluates its argument.

CONTENTS (CONTENTS <file>)
<file> must be a file name. CONTENTS returns the file definition of <file>. If <file> already has a FILE property on its property list, that property is returned by CONTENTS. Otherwise, INVENTORY is called to create a file definition for the file. CONTENTS does not evaluate its argument. CONTENTS should usually be used instead of INVENTORY to get a file definition because it does not reread <file> if <file> already has a FILE property on its property list.

CONTENTSX
same as CONTENTS but does evaluate its argument.

LOOKUPFILE (LOOKUPFILE <file>)
<file> must be the name of a file. LOOKUPFILE returns a non-NIL value if <file> exists, and returns NIL if <file> does not exist. LOOKUPFILE evaluates its argument.

writing files

The operations that write files are:

FILE	creates or modifies a file and its definition.
EDIT	edits a file definition and modifies the file.
RENAMEFILE	renames file to <name> if file <name> doesn't already exist.
DRENAMEFILE	deletes any old file with <name>, then renames file to <name>.
DELETEFILE	deletes a file.

FILE (FILE <f> <page1> <page2> ... <pagen>)
where <f> is an identifier,
<pagei> is (PAGE <exp1> <exp2> ... <expn>),
and <expi> is an s-expression.

The following formats for an <expi> have special meanings:

a list of a definer and an identifier: (<definer> <identifier>),
for example, "(DEFINE TNPO)";

a list of the atom ">" and an identifier: (> <identifier>),
for example, "(> TNPO)";

or a list of ">", a definer, and an identifer: (> <definer> <identifier>),
for example, "(> DEFINE TNPO)";

FILE is the definer for files. Evaluating a FILE definition causes a file to be written. FILE returns <f>. FILE does not evaluate its arguments.

creating new files

The way to create a new file is to evaluate a file definition. For every page list in a file definition, a pagemark is inserted into the file, and for every s-expression in the page list, an s-expression is written onto that page. For example, if you type:

```
>(FILE FOLLY (PAGE (> DSYNM MAC1) (> DE F1))
             (PAGE (> DEFINE F2) (> DEFINE F3)))
FOLLY
```

then you create a file with two pages. The first page contains definitions of the synmac MAC1 and the expr F1. The second page contains the two defs F2 and F3. Each expression designating a definition is of the form:

(> <definer> <identifier>)
or just
(> <identifier>)

The right-arrow (greater-than sign: ">") means "insert this into the file". The right-arrow causes the file editor to write onto the file a new definition of <identifier>. The type of definition written -- expr, def, synmac, fexpr, etc. -- is indicated by <definer>. If <definer> is absent, then the function PPX is called to pretty-print all definitions of <identifier> onto the file. Any definition marked by a ">" MUST be on the property list of <identifier> when the file is written or an error will result. Obviously, when a FILE definition is used to create a file, every definition must be marked by a ">".

modifying existing files

FILE may also be used to modify an existing file. One usually wants to copy some definitions from the old version of <f> onto the new version of <f>. A definition to be copied is designated in the format: (<definer> <identifier>). This is roughly the same format as above but WITHOUT the right-arrow, ">". For example, evaluating this file definition:

```
>(FILE HOLLY (PAGE (DSYNM MAC1) (> DE F1))
             (PAGE (DEFINE F2) (DEFINE F3)))
HOLLY
```

causes the file HOLLY to be modified to include a new definition of the expr F1. F1 may or may not have been on the old version of the file named HOLLY. The SYNMAC definition of MAC1 and the DEF definitions of F2 and F3 are all copied from the old version of HOLLY. If an old version of HOLLY does not exist, or if any of MAC1, F2, or F3 is missing from the old version of HOLLY, an error will result. It doesn't matter in what order MAC1, F2, and F3 appear on the old file, but they must be there somewhere.

Other expressions in page lists are copied literally onto the file. Strings or lists whose first elements are not definers are written onto files unevaluated.

After you have finished editing a file definition and exited from the editor with the OK command, the file editor will ask you to confirm that you want to rewrite the file according to the new file definition. If you say yes by typing "T", then the file editor will try to rewrite the file. If any error occurs during the rewriting -- e.g., if a definition marked with a ">" can't be found on the property list of the indicated identifier, or if a definition NOT marked with a ">" can't be found on the old version of the file -- then FILE generates a Ilisp error and no files are changed. If the rewriting is successful, then the old version of the file is saved and is given the name <f>-LBK, i.e., the name of the file with the new extension "LBK".

File definitions for the files in the user's directory are NOT automatically created when SKI is initialized, and a file definition cannot be edited until it is created. Call (CONTENTS <f>) to get a first file definition for <f>. The (EDIT <f> FILE) will allow you to edit the definition.

page 48. Restore file definition for file. Sample of file editing.

If you mangle a file definition in the editor, then you must restore it in one of two ways: If you have not yet exited from the editor, you may use the "!UNDO" command to undo all changes made to the file definition since the beginning of that call to EDIT. If you have already exited from the editor, then you must call INVENTORY (see INVENTORY above). Both (INVENTORY <f>) and (CONTENTS <f>) return file definitions for <f>, but CONTENTS will return the file definition already on the property list of <f>, whereas INVENTORY always rereads file <f> to create a new, accurate file definition.

Some files may not be appropriate for manipulation by the SKI file editor. Files that contain many s-expressions that are not definitions are not suitable for editing by the file editor. S-expressions that are not definitions (i.e., are not lists whose first elements are definers) are copied in full into the definition of the file. This can make a file definition very bulky. A file definition for a file containing no definitions at all is a full copy of the file in s-expression form!

Suppose that Kim wants to create a file with her definitions of TNPO, TNPO-HELP, and EVENP on it. Before she can edit the definition of a file, she must create a file.

```
>(FILE TNPO-KIM)
OK to write file TNPO-KIM?
Type 'T' for yes, 'NIL' for no, then carriage return>>T
TNPO-KIM
>
```

Now she edits it.

```
>(EDIT TNPO-KIM FILE)
EDIT
#PP
(FILE TNPO-KIM)
>
```

She changes the file definition to include the functions she wants, but she forgets to put a right-arrow ">" into "(DE EVENP)".

```
#(N (PAGE (> DEFINE TNPO) (> DEFINE TNPO-HELP))
    (PAGE (DE EVENP) (AMAPCAR TNPO (QUOTE (3 5 20 40)))))
#PP
(FILE TNPO-KIM
      (PAGE (> DEFINE TNPO) (> DEFINE TNPO-HELP))
      (PAGE (DE EVENP) (AMAPCAR TNPO (QUOTE (3 5 20 40)))))
#OK
OK to write file TNPO-KIM?
Type 'T' for yes, 'NIL' for no, then carriage return>>T
Can't find EVENP on file TNPO-KIM.
SKI-ERROR
>
```

At this point, file TNPO-KIM has not been changed. It is still an empty file. No changes are made when an error occurs. Kim can either get a fresh file definition of TNPO-KIM by calling (INVENTORY TNPO-KIM), or she can try to patch up the present version. She opts for the latter:

```
>(EDIT TNPO-KIM FILE)
EDIT
#PP
(FILE TNPO-KIM
      (PAGE (> DEFINE TNPO) (> DEFINE TNPO-HELP))
      (PAGE (DE EVENP) (AMAPCAR TNPO (QUOTE (3 5 20 40)))))
#F DE  PP
(DE EVENP)
#(-1 >)  PP
(> DE EVENP)
OK
OK to write file TNPO-KIM?
Type 'T' for yes, 'NIL' for no, then carriage return>>T
TNPO-KIM
>(PP TNPO-KIM)
(FILE TNPO-KIM
      (PAGE (DEFINE TNPO) (DEFINE TNPO-HELP))
      (PAGE (DE EVENP) (AMAPCAR TNPO (QUOTE (3 5 20 40)))))
```

Her file has been successfully rewritten and the file definition updated. Notice that all of the right-arrows are now gone.

EDIT (EDIT <id> FILE)

EDIT (same as EDIT in section F, "EDITING A DEFINITION") lets you edit the file definition of <id>. The second argument to EDIT, the atom "FILE", specifies that the FILE property of <id> is to be edited. After the OK command, the file editor restructures the file to reflect its new definition, (see discussion in FILE above). EDIT does not evaluate its arguments.

EDITX

same as EDIT but does evaluate its arguments.

RENAMEFILE (RENAMEFILE <old> <new>)

RENAMEFILE tries to rename the file named <old> to have the name <new>. If file <old> does not exist or if file <new> already exists, renaming fails, RENAMEFILE returns NIL, and no files are changed. RENAMEFILE returns T if the renaming is successful. RENAMEFILE also moves the FILE property of <old> to <new>. The arguments are not evaluated.

RENAMEFILEX

same as RENAMEFILE but does evaluate its arguments.

DRENAMEFILE (DRENAMEFILE <old> <new>)

DRENAMEFILE deletes file <new> if it exists, removes the FILE property of <new>, and then calls RENAMEFILEX to rename file <old> to have the name <new>. DRENAMEFILE does not evaluate its arguments. DENAMEFILE returns T if the renaming is successful.

DRENAMEFILEX

same as DRENAMEFILE but does evaluate its arguments.

DELETEFILE (DELETEFILE <f>)

DELETEFILE deletes file <f> and removes its FILE property. If <f> does not exist or cannot be deleted, DELETEFILE returns NIL. DELETEFILE returns T if it successfully deletes <f>. DELETEFILE does not evaluate its argument.

DELETEFILEX

same as DELETEFILE but does evaluate its argument.

K. SIMULATING PARALLEL PROCESSING

SKI provides a scheduler for simulating parallel processing. The scheduler is written in Ski and is derived from [Wand 80b]. Other scheduling constructs are presented in [Wand 80b] and could be useful for many programs. All of the functions for simulating parallelism are written in Ski and can be modified and extended by a SKI user. This construct is called RACE and we feel that it is general and useful for simulating parallelism. In the discussion of RACE, we'll often say "parallelism" where we mean "simulated parallelism".

RACE (RACE)
is a function of no arguments that returns an object, an instance of the class RACE. We'll call such an object a "race". A race holds a set of processes (Scheme procedures) and simulates the parallel execution of those processes. Processes that have been added to this set are called "contestants" and are said to be "in the race". In a newly created race, the set is empty; there are no contestants in the race. Sample call:

```
>(ASETQ R (RACE))
**OBJECT**
```

Now R is an empty race. It contains no contestants yet, and it is not evaluating anything yet. A race takes four messages: ENTER, EMPTYP, RUN, and DROPOUT. ENTER puts a contestant (a process written as a Scheme procedure of no arguments) into the race. EMPTYP returns T if there are no processes in the race. RUN commands a race object to evaluate in parallel all of the contestants currently in the race and to return the value of the first one that converges. When a contestant converges, it is excluded from the race, leaving the other (partially evaluated) contestants in the race. A race may be loaded with several contestants, and then RUN again and again to return values one by one until all of the processes have converged. Meanwhile, more contestants may be added by sending more ENTER messages. Contestants may themselves be races.

The message DROPOUT is useful only when sent from a contestant to the race that it is in. When this happens, that contestant drops out of the race without returning a value -- it simply disappears.

ENTER (SEND <race> (ENTER <contestant>))
for example: (SEND R (ENTER (LAMBDA NIL (TNPO 27))))
places <contestant> into <race>. <contestant> must evaluate to a closure of no arguments. ENTER returns NIL. The message ENTER does NOT cause <race> to evaluate anything; it just adds another process to the race.

Notice that "(SEND R (ENTER (TNPO 27)))" won't work. This call would enter the value of (TNPO 27) into R, i.e., it would enter the integer 111 into the race as a contestant. But 111 is not a process; it is an integer, and would eventually cause an error. Wrapping a "(LAMBDA NIL <exp>)" around any Scheme expression <exp> will transform <exp> into a process of no arguments, because the lambda expression evaluates to a closure that will do <exp> when eventually called.

EMPTYP (SEND <race> (EMPTYP))
for example: (SEND R (EMPTYP))
causes <race> to return T if and only if there are no contestants in the race. If EMPTYP returns NIL then a RUN message may result in the eventual return of another value from the race. But even when EMPTYP returns NIL, there is no guarantee that the remaining contestants will not contain infinite loops, and no guarantee that the race will not become empty while it is running because all of the contestants DROPOUT; it is up to the user to ensure that these do not happen.

RUN (SEND <race> (RUN))
for example: (SEND R (RUN))
causes <race> to evaluate in parallel all of the contestants currently in the race and to return the value of the first contestant that returns. That

converged process is excluded from the race so that the next time a RUN message is sent to <race>, it will start again to compute a next value. If <race> is sent the message RUN when it is empty, or if it becomes empty while it is running, it generates the error (ERR (QUOTE EMPTY)). It is often useful to wrap an ERRSET around a call to a race so that this condition can be detected.

If a contestant generates an error, parallel evaluation within the race stops and control bounces out of the race either to a break, to the nearest ERRSET, or to the top level. Control does not return to the race until another RUN message is sent. The error causes the erring contestant to be excluded from the race; at the moment of the error, it disappears from the race without returning a value.

DROPOUT (DROPOUT)

DROPOUT causes the contestant to voluntarily kill itself off -- to disappear from the race without returning any value. DROPOUT directs <race> to exclude from the race the currently running contestant. DROPOUT does not cause the race to return; the race just keeps running with one process fewer. If the last contestant in a race drops out, then the race becomes empty and an (ERR (QUOTE EMPTY)) is generated by the race. DROPOUT must not be executed by a process that is NOT in a race; the result of such a call is undefined and may be an error.

ENTER (ENTER <fn>)
for example: (ENTER (LAMBDA NIL (TNPO (ADD1 N))))

ENTER is discussed above. It adds the value of <fn> to the race. Only a process IN a race can call ENTER in this syntax because no race is explicitly mentioned in the call. The race into which <fn> is entered is the race that the calling process is in. ENTER must not be executed in this syntax by a process that is NOT in a race; the result of such a call is undefined and may be an error. This syntax for ENTER permits contestant processes to spawn new contestant processes from within the race.

When using races, the fluid variable THISRACE is reserved for use by the races. We used a little trick to let contestants send messages (the DROPOUT and ENTER messages with implicit destinitation races) to the races that they are in. As a process is entered into <race>, the variable THISRACE is bound to <race> in the dynamic environment of the process.

#TIME, time slices, and timing interrupts:

SKI uses timing interrupts to simulate parallel processing. When the Ilisp variable #ENABLED is non-NIL, timing interrupts are generated by the stack language interpreter. #ENABLED is initialized to T, and should be non-NIL when using races.

The Ilisp variable #TIME has as its value the length of time in milliseconds between interrupts. This length of time is called a "time slice". The clock used is ILISP's clock accessed through the ILISP subr TIME. The values return by successive calls to TIME increase monotonically, counting the amount of time in milliseconds used by the ILISP interpreter. Interrupts are generated only during the execution of the stack-language instructions PUSH#, PUSHV#, and PUSH-FLUID#, which push values from the environments onto the value stack. The ILISP variable #TIME should have as its value the desired minimum time between interrupts. Before each push instruction, #TIME is checked and compared to an internal variable, #ALARMCLOCK, to see if #TIME milliseconds have elapsed since the last interrupt. If so, then an interrupt is generated as explained in [Wand 80b]. Control is automatically passed to the closure bound to the variable PREEMPT in the current fluid environment. PREEMPT is initialized in the global environment to do nothing; it just returns control to the interrupted procedure. But each race binds PREEMPT in its own fluid environment, so each race has a private PREEMPT function that schedules the contestants in the race. It lets the contestants run one by one, one time slice each turn, until one of them converges.

page 52. Custom parallelism. Time slices. Sample race. TNPO.

A SKI user may write her own Ski functions to simulate parallel evaluation. PREEMPT may be locally redefined to handle interrupts any way the user likes. Typically, PREEMPT is used to save a continuation, capturing and saving the current state of the executing process.

The variable #TIME is checked each time a push instruction is executed, so changes to #TIME made dynamically during the execution of a Scheme program will change the length of time between interrupts. #TIME is initialized to 50 (decimal). Higher settings for #TIME give bigger time slices to each process and result in more coarsely simulated parallelism with less time spent doing interrupt handling. If one is running the interpreted, Ilisp source-language version of SKI, #TIME must be set much higher, about 800 decimal, because interpreted ILISP runs much more slowly.

With RACE, Kim now has the tools to run in parallel several tests of her Ski function TNPO which computes the 3n+1 property of positive integers. She uses the version of TNPO developed in section E, "DEFINITIONS AND PRETTY-PRINTING THEM". Here is that same algorithm in a syntax more typical of Scheme than her earlier version:

```
>(DEFINE
  TNPO
  (X)
  (LABELS
   ((EVENP (LAMBDA (Y) (ZEROP (REMAINDER Y 2))))
    (TNPO-H
     (LAMBDA(N A)
      (IF (EQ N 1)
          A
          (TNPO-H (IF (EVENP N) (QUOTIENT N 2) (ADD1 (TIMES N 3)))
                  (ADD1 A))))))
   (TNPO-H X 0)))
TNPO
>
```

She does NOT use the version that stores previously computed answers for future use, developed in section I, "CLASSES AND OBJECTS". The expression below creates a race, adds three contestants to it, and asks for three values. The contestants are calls to TNPO. Since each TNPO calculation will iterate all the way down to 1, she will be able to see whether contestants that require fewer steps to reach 1 will return values sooner than contestants that require more steps:

```
>(LET ((R (RACE)))
      (BLOCK
        (SEND R (ENTER (LAMBDA NIL (LIST 28 (TNPO 28)))))
        (SEND R (ENTER (LAMBDA NIL (LIST 27 (TNPO 27)))))
        (SEND R (ENTER (LAMBDA NIL (LIST 26 (TNPO 26)))))
        (PRINT (SEND R (RUN)))
        (PRINT (SEND R (RUN)))
        (PRINT (SEND R (RUN)))
        (QUOTE DONE)))
(26 10)
(28 18)
(27 111)
DONE
>
```

Next, Kim writes a function, (LOAD-RACE <l>), to load a race with calls to TNPO, one call for each integer in the list <l>. She also writes DRIVE-RACE which asks for values from the race and prints them.

page 53. Sample race, continued. Stream of values.

```
>(DEFINE LOAD-RACE
         (R L)
         (IF (NULL L)
             R
             (BLOCK (SEND R
                          (ENTER (LAMBDA NIL (LIST (CAR L) (TNPO (CAR L))))))
                    (LOAD-RACE R (CDR L)))))
LOAD-RACE
>(DEFINE DRIVE-RACE
         (R)
         (LET ((A (ERRSET (SEND R (RUN)) NIL)))
              (IF (EQ A (QUOTE EMPTY))
                  A
                  (BLOCK (PRINT (CAR A)) (DRIVE-RACE R)))))
DRIVE-RACE
>(DRIVE-RACE (LOAD-RACE (RACE) (QUOTE (20 21 22 23 24 25 26 27 28 29))))
(21 7)
(24 10)
(26 10)
(20 7)
(23 15)
(28 18)
(29 18)
(22 15)
(25 23)
(27 111)
EMPTY
>
```

Notice that the values are returned only approximately in order of the length of time it takes to compute them. Values of contestants that require similar lengths of time to converge will be close to each other in the stream of values returned from a race, but their order is subject to seemingly unpredictable local variation, because interrupts control the race.

L. THE SKI COMPILER

The SKI compiler translates Scheme source-language expressions into stack language. All identifiers bound by LAMBDA or LABELS in source-language expressions are replaced by pairs of integers (the "indirect address plus offset" pairs used in many implementations of Algol-like languages) in stack-language instructions. Any static identifier that is not in the static scope of a Scheme expression is assumed to refer to a global value. It is not possible to use an identifier created at run time with Ilisp's atom-creating functions (e.g. READLIST) to refer to values in Ski static environments. This guarantees that identifiers refer to the "right" (as prescribed by the specification of Scheme) values.

The SKI compiler translates each expression typed to the read-eval-print loop into stack language by a recursive descent compilation. During compilation, each subexpression whose CAR is an atom is classified as one of the following four types of expressions:

I) a Scheme procedure
  a) an identifier that has been bound by LABELS, LET, or LAMBDA; or
  b) an identifier that has a DEF property (has been defined with DEFINE).

II) a Scheme magic word
  a) a kernel magic word, i.e., LAMBDA, QUOTE, IF, LABELS, ASETQ, CATCH, FLUID, FLUIDSETQ, FLUIDBIND, or CLASS; or
  b) a synmac defined with DSYNM).

III) an Ilisp macro
  defined with DM; analogous to the SCHMAC of [Steele and Sussman 78].

IV) a primop
  a) an expr, subr, lexpr, or lsubr.
  b) a fexpr or fsubr.

After compilation, only calls of types I and IV will remain, since all magic words and macros will have been expanded.

The order of precedence given by the compiler to these different function types is important because it is possible for a single identifier to have multiple definitions as, say, a Scheme procedure (def), a syntactic macro (synmac), and an Ilisp expr. The compiler uses two pieces of information to classify each call:

1) the definitions on the property list of the function name, and
2) the static environment in which the call takes place.

Expression types I through IV are explained below in slightly more detail and in descending order of their precedences to the compiler.

Ia) a Scheme procedure bound in the static environment.
A local binding of an identifier, i.e., a LAMBDA or LABELS binding, has highest priority and is used in preference to any def, magic word, or Ilisp function with the same name. (Actually, all lexically bound identifiers, whether they have procedures or data as values, are treated in exactly the same way. But a lexically bound identifier in the CAR position of a function call must have a procedural value -- something of which PROCP is true -- or an error will result.)

Ib) a globally defined Scheme procedure, defined with DEFINE.
Defs are treated as if they were bound in the outermost block of the static environment. A DEF property for an identifier is used in preference to any other definition or functional property EXCEPT for local binding. A globally defined def MUST be defined (though not necessarily compiled) before any calls to it are compiled. As explained below in "the forward declaration non-problem", this is usually not a problem.

IIa) a Scheme kernel magic word.
The identifiers LAMBDA, QUOTE, IF, LABELS, ASETQ, CATCH, FLUID, FLUIDSETQ, FLUIDBIND, and CLASS have next priority. Each of these identifiers has a COMPILE property on its property list. The COMPILE property signals that there are

compiler routines to generate stack language for expressions with these identifiers in their CAR positions.

IIb) a synmac, defined with DSYNM.

Some SKI-provided magic words and all user-defined magic words have this property. These macros are expanded at compile time. The expansion resulting from a call to a synmac is then compiled. Synmacs may be defined in terms of themselves and other synmacs, but this recursive expansion must eventually stop and return an expression that is composed entirely of calls to Ski primitives, SKI-supplied magic words, and Ski and Ilisp functions.

III) a macro, defined with DM.

A MACRO definition comes next. Ilisp macros are expanded at compile time, and must be written as syntactic macros, i.e., they should avoid the use of EVAL.

IVa) a Lisp function that takes evaluated arguments.

The EXPR, SUBR, LEXPR, and LSUBR properties are checked next. The compiler generates stack language to evaluate all of the arguments in the call, and then to call the Ilisp interpreter to apply the expr, subr, lexpr, or lsubr to its evaluated arguments. Only lexprs and lsubrs that take definite numbers of arguments can be called directly from SKI. Each subr and lsubr must have the NUMBER-OF-ARGS property on its property list, telling how many arguments it takes.

IVb) a Lisp function that does not (necessarily) evaluate its arguments.

A call to a fexpr or fsubr has the lowest priority. ILISP's EVAL is called at run time to evaluate calls to fexprs and fsubrs. No Scheme variables or procedures, either locally or globally bound, may be referenced in a call to a fexpr or fsubr. The call must be written in Ilisp and is evaluated by ILISP.

Calls to defs, exprs, subrs, lexprs, and lsubrs are checked at compile time to ensure that the correct numbers of arguments are supplied to them. When a call to an expr or def is compiled, the EXPR or DEF definition on the property list of the function name is used to check whether the correct number of arguments is supplied to the call. If not, the error "Mismatch of parameters and arguments in <call>" occurs.

The number of arguments that a subr or lsubr requires is stored on its property list under the property name NUMBER-OF-ARGS. This integer is used to check the numbers of arguments in calls to that subr or lsubr. If a subr or lsubr has no NUMBER-OF-ARGS property, the error "Declare the number of arguments required by <subr or lsubr>" occurs, and the user is given directions to make the appropriate declaration to continue the computation.

page 56. No forward declaration problem. Definition time. Compile time.

the forward declaration non-problem

SKI determines the type of every function call at compile time. Because of this, SKI is susceptible to a "forward declaration problem". In a one-pass compiler for, say, Algol or Pascal, a call to an as-yet-undefined procedure must be preceded by a forward declaration of that procedure. A forward declaration typically tells the name of the procedure, the type of value it will return, and the number and types of its arguments. Without a forward declaration, a one-pass compiler cannot know if a called procedure will be defined later, or if the call is just a typographical error. Mutually recursive function definitions cannot be handled by a one-pass compiler without forward declarations. In order to compile a function call, the SKI compiler needs to know to what type of function -- Scheme procedure or Ilisp function -- the call refers.

SKI does not require forward declarations. It uses the definition of a Scheme procedure -- its DEF property -- instead of a forward declaration. SKI distinguishes the DEFINITION TIME of a Scheme procedure from the COMPILE TIME of that procedure. The execution of a DEFINE statement does not cause a def to be compiled -- only defined. In this way, definitions may be typed in, read from files, whatever, before any compilation is done. The first time that a defined Scheme function is CALLED at run time, it is compiled and immediately executed. Certainly at execution time, any other functions called from it should also have been defined, though not necessarily compiled. Any of these functions that has not yet been compiled is also compiled, and so on.

One can write and edit Ski functions with the freedom of an interpreted language like Lisp because no function is compiled until it is called. Postponing the compile time of functions requires a compiler resident at run time, but then so do run-time calls to the Scheme primitive function ENCLOSE, so the compiler is available at no extra cost. In fact, all calls to the compiler occur through calls to ENCLOSE. A consequence of this is that all compiler errors are run-time errors and can be handled by Ski ERRSET (see section H, "INTERACTIVE HELP AND ERROR MESSAGES").

M. THE STACK LANGUAGE INTERPRETER

There are several flags that the SKI user may set to affect the SKI program. They are all global Ilisp variables.

#ENABLED

If non-NIL, timing interrupts are generated by the interpreter. #ENABLED is initially T. If you set #ENABLED to NIL, you can't use races because the interpreter does not generate the timing interrupts necessary to time-share the contestants.

#TIME

It must have an integer value: the length of time (in milliseconds counted by ILISP) between timing interrupts. #TIME is initialized to 50 (decimal).

#STATS

If non-NIL, some SKI performance statistics are printed out after the evaluation of each expression typed to the top-level read-eval-print loop. The number of steps the program counter takes, the time in milliseconds, the average time per step in milliseconds, the number of CONSes performed, and the amount of garbage collection time in milliseconds are all printed.

N. CUSTOMIZED DEFINITION TYPES

A SKI user may create a new definition type by writing a defining function -- definer -- for that definition type. Some users might find it useful to add a COMMENT definition type. Then the editor could be used to create and modify comments, and to write them onto files. SKI is initialized to use the definers:

DEFINE, DSYNM, DE, DM, SETQQ, DF, and FILE.

A list of all definers and the property names to which they correspond is the value of the global Ilisp variable DEFINERS. DEFINERS is initialized to:

```
((DEFINE DEF)   (DSYNM SYNMAC)  (DE EXPR)       (DM MACRO)
 (SETQQ VALUE)  (DF FEXPR)      (FILE FILE))
```

A list of the corresponding property names is the value of the global Ilisp variable GRINPROPS:

(DEF SYNMAC EXPR MACRO VALUE FEXPR FILE)

A definer <d> is an Ilisp macro, fexpr, or fsubr. Its first argument must be an identifier <id>. A defining expression is of the form

(<d> <id> <s-exp1> <s-exp2> ... <s-expn>)

for example

```
(DEFINE RAC (IF (CDR L) (RAC (CDR L)) (CAR L)))
```

When a defining expression is evaluated, the function <d> constructs a certain s-expression <s> from the <s-expi> and stores it on the property list of <id> under a property name <prop>. The definer must not evaluate any of its arguments or any parts of its arguments. It simply rearranges them into some s-expression <s> and uses PUTPROP or DEFPROP to put <s> onto the property list of <id>. The name of the definer, <d>, paired with its corresponding property name <prop>, must occur in the list DEFINERS, and the property name, <prop>, must be GRINPROPS. Definitions of a new customized type will be edited and pretty-printed in Ilisp's standard DEFPROP format.

Three additional functions may be defined to format customized definitions for editing and pretty-printing. These functions must be exprs or subrs and must follow the naming conventions explained below. If three such functions are defined for a new definition type, definitions of that type will be edited and pretty-printed in the format they dictate, instead of in ILISP's standard DEFPROP format. The names of the three functions are composed from the property name <prop> from GRINPROPS, and the suffixes "-PPRINT", "-FORMAT", and "-UNFORMAT":

<prop>-PPRINT, <prop>-FORMAT, and <prop>-UNFORMAT.

For example, the property name <prop> for Scheme procedures is DEF, so the functions DEF-PPRINT, DEF-FORMAT, and DEF-UNFORMAT control the formatting and printing of Scheme procedures.

<prop>-PPRINT (<prop>-PPRINT <id> <s>)

takes two arguments: an identifier <id>, and an s-expression <s> that is the <prop> property of <id>. <prop>-PPRINT does the actual pretty-printing of a definition. It is called by PPX and by the file editor to print definitions of type <prop>. It usually calls <prop>-FORMAT to format <s> into a defining expression. <prop>-PPRINT's value is not used. Sample:

```
(DE DEF-PPRINT (ID S) (TERPRI) (SPRINT (DEF-FORMAT ID S) 1.) (TERPRI))
```

<prop>-FORMAT (<prop>-FORMAT <id> <s>)
takes the propert-list version of the definition and its name, and returns the defining expression. It takes the two arguments, <id> and <s>, as described for <prop>-PPRINT above. <prop>-FORMAT must not print anything, but must arrange <id> and <s> into the DEFINING EXPRESSION for this definition, i.e., the list that would be evaluated in order to define <s> as the <prop> property of <id> -- the list whose CAR is a definer, whose CADR is <id>, and whose CDDR is (<s-exp1> ... <s-expn>). The value of <prop>-FORMAT must be a defining expression. Sample:

```
(DE DEF-FORMAT (ID S) (APPEND (LIST (QUOTE DEFINE) ID (CADR S)) (CDDR S)))
```

sample call:

```
>(DEF-FORMAT (QUOTE RAC) (QUOTE (LAMBDA (L) (IF (CDR L) (RAC (CDR L)) (CAR L)))))
(DEFINE RAC (L) (IF (CDR L) (RAC (CDR L)) (CAR L)))
>
```

<prop>-UNFORMAT (<prop>-UNFORMAT <defexp>)
takes the defining expression for a definition and returns a list of the name of the definition, its definition type, and its property-list version. It takes one argument, a defining expression <defexp> (as explained above). <prop>-UNFORMAT returns a list of three things:

(<id> <prop> <s>)

where <id>, <prop>, and <s> all have the meanings explained in the paragraphs above. <id> is the CADR of <defexp>. <prop> is the property name associated with this definition type. <s> is the s-expression that would be stored on the property list of <id> if <defexp> were evaluated. Sample:

```
(DE DEF-UNFORMAT
    (D)
    (LIST (CADR D) (QUOTE DEF) (CONS (QUOTE LAMBDA) (CONS (CADDR D) (CDDDR D)))))
```

sample call:

```
>(DEF-UNFORMAT (QUOTE (DEFINE RAC (L) (IF (CDR L) (RAC (CDR L)) (CAR L)))))
(RAC DEF (LAMBDA (L) (IF (CDR L) (RAC (CDR L)) (CAR L))))
>
```

0. THE SKI SOURCE FILES

This section is only for someone who wants to change the SKI source files or wants to recompile them from scratch.

The Ilisp functions that implement SKI must be compiled and used as subrs. This prevents any possible conflicts (duplications) between Ilisp variable names in users' function definitions and variable names in the source language for SKI. The source files for SKI are SKI.IL and FILHAN.IL.

When the ILISP compiler compiles functions from an input file, it doesn't rewrite macros onto the output file. It assumes that macros are for compilation purposes only, and its default action is to throw them away after compilation. Many important SKI functions are macros and must be explicitly saved when the source files are recompiled. These macros must be read into ILISP, along with the Lap code for SKI, when the SKI system is rebuilt from the source files.

The ILISP compiler at SAIL did not compile the Ilisp fsubr AND correctly. We tricked it by rewriting AND as an Ilisp macro for compilation purposes only:

```
(DM AND (M) (ANDX (CDR M)))

(DE ANDX
    (L)
    (COND ((NULL L) T)
          ((NULL (CDR L)) (CAR L))
          (T (LIST (QUOTE COND) (LIST (CAR L) (ANDX (CDR L)))))))
```

The default binary program space allocation of Ilisp must be overriden and raised provide room to load the compiled files FILHAN.LAP and SKI.LAP into ILISP.

The SKI compiler overflows the ILISP regular and special pushdown stacks (both at IU and SAIL) if the default allocation of 1000 octal for each is used. Increasing both by 1000 octal has been sufficient for all of our work.

naming conventions in the source code for SKI

The names of all of the functions in the SKI program, except for those that are specified in the Revised Scheme Report [Steele and Sussman 78] and the top-level function SKI, have a hash sign, or pound sign, "#", suffixed to them (e.g., COMPILE1#, APPLY-EXOP#). This is to set them off from other Ilisp functions and to help the SKI user avoid accidentally redefining them by defining Ilisp functions with the same names. The global Ilisp variables that the SKI compiler and stack language interpreter use as internal flags and as registers for the virtual stack machine are prefixed with a hash sign, "#", (e.g., #VALSTACK, #CSTACK, #ENABLED, #COMPILETRACE), for the same reason. The compiler routines that generate stack language for the kernel set of Scheme magic words (LAMBDA, QUOTE, IF, LABELS, ASETQ, CATCH, FLUID, FLUIDSETQ, FLUIDBIND, and CLASS) are implemented as Ilisp functions whose names are composed by concatenating "COMP-", the magic word in question, and a hash sign (e.g., COMP-ASETQ#, COMP-CLASS#). The interpreter routines that execute stack language instructions are named by concatenating "EX-", the opcode name of the instruction, and a hash sign (e.g., EX-CATCH#, EX-PUSH-ENV#).

P. GLOSSARY

break
: is a call to the break package. When an error occurs and the break package is called, the computation is said to be "in a break".

break package
: is part of ILISP. It stops a computation when an error occurs and preserves the context -- the environment of variable bindings and the stack of function calls -- in which the error occurred. The break package has its own read-eval-print loop and commands that are documented in [Bobrow et al] or [Meehan 79].

classes and objects
: are programming-language constructs originally from SIMULA [Dahl et al 70] and used in SMALLTALK [Shoch] and other languages. An object is a procedure that may take different numbers of arguments depending on its first argument, called a MESSAGE. A message serves as a flag to determine what the object will do with the other arguments. A class is a prototype definition from which objects are generated. In SKI, an object is created when a CLASS expression -- a form whose first element is the atom CLASS -- is evaluated. Classes may be related to each other in a tree-like hierarchy. Some classes are then subclasses of others. An object generated from a subclass, C, can process any messages defined in C or in any of the superclasses of C. There are discussions of classes and objects as programming-language constructs in [Birtwistle et al 73] and [Shoch]. See section I, "CLASSES AND OBJECTS".

closure
: is, in Scheme, the executable form of a procedure. A closure consists of a compiled Scheme expression and an environment in which all references to variables and functions in that expression (except references to the formal parameters of the procedure) can be satisfied.

DEF
: is the atom used as the Ilisp property name for Ski source-language definitions. When a Scheme procedure or datum is defined with a DEFINE statement, that definition is placed on the property list of the name of the procedure or datum under the property name DEF.

definer
: is a macro or fexpr that puts a DEFINITION on the property list of an identifier. "Definer" can also refer to the atom that names such a function, e.g., "DEFINE", "DSYNM", "DE", "DM", etc. A definer must be a macro or fexpr (as opposed to an expr, def, subr, etc.) because a definer may not evaluate its arguments. A definer should be like a syntactic macro: it may rearrange its arguments but should not call EVAL.

definition
: is any s-expression that resides on the property list of an ILISP identifier and was stored there by a function called a "definer". Such properties are usually definitions of Ilisp functions and values, Scheme procedures, values, and macros, file definitions, etc. Definitions are pretty-printed and stored on files as described in section E, "DEFINITIONS AND PRETTY-PRINTING THEM".

dynamic
: fluid.

EXPR, FEXPR, SUBR, LSUBR, FSUBR, MACRO
: are atoms used as the property names under which definitions of ILISP functions appear on property lists. These function types are described in [Bobrow et al] or [Meehan 79]. These will often be written in lower case to refer to definitions that appear under these properties, (e.g., "Exprs, fexprs, and macros may be edited with the function EDIT.")

file definition
: is the description of a file in s-expression form. It is used for the purposes of pretty-printing and editing. It consists of the atom FILE, the name of the file, and zero or more PAGE LISTS. A page list shows what definitions and other s-expressions are on that page of the file. File definitions are described in section J, "EDITING A FILE".

fluid
: describes identifiers that have dynamic scope, as opposed to statically scoped (static) identifiers. A fluid variable has meaning (is bound) throughout the execution (not just throughout the written text) of the block where it is bound. An example that contrasts static and fluid binding may be found in section C, "SHORT DESCRIPTION OF THE SCHEME LANGUAGE AND PARALLEL PROCESSES".

ILISP
: refers to the UCI-LISP program. It supports the Ilisp dialect of Lisp, an s-expression-oriented editor, the break package, a compiler and LAP, and some debugging aids. It is documented in [Bobrow et al] and [Meehan 79]

ILISP editor
: is part of the ILISP program. It allows one to edit list structures. The editor is very convenient for defining and editing Scheme and Ilisp functions, modifying values, and for correcting faulty definitions in a break when an error has occured. Extensive documentation of the ILISP editor is available in [Meehan 79].

IUSCHEME
: is the program documented in [Wand 80a]. It is the program that was modified to become SKI. The core of SKI -- its compiler, assembly language, and interpreter -- is based on IUSCHEME.

LAP
: refers to the Lisp Assembly Program. It is a combined assembler and loader. The assemblers used by LISP1.6 and ILISP are both called "LAP".

Lap
: refers to assembly language generated by LAP.

LISP1.6
: refers to the Stanford LISP1.6 program, a predecessor of ILISP.

magic word
: is a Scheme reserved word, a key word. A core set of magic words (IF, ASETQ, LABELS, LAMBDA, etc.) is specified in the definition of Scheme and is supported by compiler routines. Some other magic words provided by SKI, and all user-defined magic words, are synmacs.

read-eval-print loop
: is a common control structure for interactive programs. A read-eval-print loop typically prints a prompt to the terminal, waits for the user to type a line or expression at the terminal, processes that input, prints a value or response, then prints the prompt again to ask for more, and so on.

Scheme
: describes procedures written in the Scheme programming language as defined in the Revised Report on SCHEME [Steele and Sussman 78]. "Scheme" may sometimes describe procedures written in this language augmented by CLASSES and OBJECTS, i.e., the language supported by the SKI program. "Scheme function" and "Scheme procedure" are synonyms.

SKI
: is the program described in this report.

Ski
: is the programming language supported by SKI. It is an extension of Steele and Sussman's Scheme, extended by the data-abstraction construct CLASS and simulated parallelism.

SKI compiler
: is the Ilisp program, described in this report, that accepts Ski expressions as input, and generates stack language output.

SKI interpreter
: stack language interpreter.

SKI stack-machine assembly language
: is the output of the SKI compiler. Often the term will be shortened to "stack language". This stack language is a list structured assembly language for the S-machine [Wand 80a]. A hardware S-machine does not exist at this time. The SKI interpreter interprets stack language.

stack language interpreter
: is the Ilisp program that interprets SKI stack language. The interpreter is the SKI virtual machine.

static
: describes identifiers that have lexical scopes, as opposed to dynamically scoped (fluid) identifiers. A static variable has meaning only in expressions written within the block where it is bound. An example that contrasts static and fluid binding may be found in section C, "SHORT DESCRIPTION OF THE SCHEME LANGUAGE AND PARALLEL PROCESSES".

SYNMAC
: stands for "syntactic macro" and is the atom used as the Ilisp property name for SKI syntactic-macro definitions. A syntactic macro defined by evaluating a DSYNM expression. A synmac is stored on the property list of the name of the macro under the property name SYNMAC.

syntactic macro
: is a macro that does not evaluate any of its arguments, or any parts of its arguments, or anything in the global state; it simply examines and rearranges its arguments SYNTACTICALLY. A syntactic macro is safe to expand at compile time because its result does not depend on the values of its arguments or on the global state of the computation, but only on the actual s-expression that is the call to that macro. This means that the value of a given call to a syntactic macro will always be the same, regardless of the environment in which it is evaluated. Therefore, syntactic macros can safely be called from other functions that will be compiled by the ILISP compiler or by the SKI compiler. Both compilers expand all macro calls at compile time. Here is a sample syntactic macro, a definition of LIST using CONS.

```
(DSYNM LIST
       Z
       (COND ((CDR Z) (LIST (QUOTE CONS)
                            (CADR Z)
                            (CONS (QUOTE LIST) (CDDR Z))))))
```

top level
: is usually a read-eval-print loop.

UCI-LISP
: ILISP
: ILISP

VAL
: is the atom used as the Ilisp property name for global values in the SCHEME environment. A global value set by evaluating a DEFINE or ASETQ expression is kept on the property list of the identifier under the property name VAL.

references:

[Birtwistle et al 73]
Birtwistle, G. M. et al "Simula Begin", Auerbach Publishers Inc, Philadelphia, Pennsylvania, 1973.

[Bobrow et al] UCI-LISP Manual

[Dahl et al 70]
Dahl, O. J. et al "The SIMULA 67 common base language", Pub. S-22, Norwegian Computing Center, Oslo, 1970.

[Hofstadter et al 79]
Hofstadter, D. R. et al, "SEEK-WHENCE, A Project in Pattern Understanding", NSF proposal, Indiana University, Bloomington, Indiana, 1979.

[Meehan 79]
Meehan, J. R., "The New UCI Lisp Manual", Lawrence Erlbaum Assoc, Hillsdale, New Jersey, 1979.

[Reddy et al 76]
Reddy, R. et al "The HEARSAY II System", Working Papers in Speech Recognition, February, 1976, Computer Science Speech Group, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

[Shoch] An Overview of the Programming Language Smalltalk-72

[Steele and Sussman 78]
Steele, G. and Sussman, G. "The Revised Report on SCHEME", AI Memo no. 452, January, 1978, MIT AI Lab, Cambridge, Massachusetts.

[Steele and Sussman 80]
Steele, G. and Sussman, G., "Design of a LISP-based Microprocessor", Communications of the ACM, vol. 23, number 11, November, 1980.

[Wand 80a]
Wand, M. "SCHEME 3.1 Reference Manual", Technical Report no. 93, June, 1980, Computer Science Dept., Indiana University, Bloomington, Indiana.

[Wand 80b]
Wand, M. "Continuation-Based Multiprogramming", Proceedings of 1980 LISP Conference, August, 1980.