The things "everybody knows" about YKTLISP

This is basically an attempt to put some of my most popular song-and-dance routines into labanotaion. It is a miscellany of hints, warnings and admonitions written "on the fly". It is guaranteed to be incomplete, but (as of today) correct.

YKTLISP / VM interaction.

XEDIT

Obey

PRY

YKTLISP peculiarities

Cyclic structures

Debugging compiled code versus interpreted code.

Values and functions.

Macros

Scope of variables.

Structured bound variable lists.

Implied PROGN

Scope of GO and EXIT

New built-in functions and special forms.

STATE and state descriptors

& -- the back-trace function.

CATCH and THROW.

SUPV and the break loop

Taking over the error handler.

Evaluation

External interrupt

PROG and SEQ

READ <--> PRINT

What are all those Qs and commas doing?

"T

Property lists

Sub-mini guide to LISPEDIT

Versions and systems

YKTLISP / VM interaction.

XEDIT

YKTLISP and XEDIT have two incompatibilities. First, YKTLISP is loaded into FREE/FRET storage, which is allocated at load time, and released only upon invocation of the DROPnnnn module. XEDIT normally runs from a shared segment, but if the user's virtual machine overlaps its shared segment area it loads itself into the equivalent storage, whether anything else is there or not. As a result, if you exit from LISP with the intention of resuming and use XEDIT, you may find that LISP has been partially overwritten. (Most users of YKTLISP at Yorktown use other editors than XEDIT for this reason, as well as for the second one, below.)

YKTLISP's reader completely ignores end-of-lines, treating a file as a single stream of characters. If you are using V format files for input, this may require a seemingly gratuitous leading blank on some lines. Some of the editors at Yorktown support so-called VB format files, in which all lines less than the maximum length for the file have a single trailing blank. XEDIT does not support such files. Thus a list of numbers or identifiers which spans two records may be misread if the second record does not start with a blank, e.g. the lines:

```
(a b c
d e f)
```

would be read as (a b cd e f). The LISP print functions may produce such files (as they too ignore end-of-line conditions), an some of the source files included in the YKTLISP distribution may also have this property. As long as these files are left untouched all is well, but if they are edited with XEDIT and refiled, things may break. One solution is to use F format files, however this is wasteful of disk space. Note that LISPEDIT does this, but it does not PRETTYPRINT its files, as they are not expected to be examined outside of LISPEDIT, thus the space lost is minimal.

Obey

YKTLISP includes a function, OBEY, which is supposed to pass a command string to the CMS command interpreter from inside LISP. It is written assuming the existence of a MODULE, called OBEY, which will do its work for it. This module is called by an SVC 202 with a new-form PLIST, with the command, OBEY, not contiguous with the argument list (the command string). As of now there does not exist such a module which will run outside of the Yorktown modified VM system.

The system itself does not depend on the OBEY function working, but if you need this facility you will have to construct such a module yourself. The LISPEDIT command CMS also assumes that OBEY works.

PRY

The function PRY forces an operation exception, to allow a system programmer to enter what ever debugging facility is available on the system. Execution can be resumed without harm after the exception is taken. The name derives from the particular debugging facility used at Yorktown.

YKTLISP peculiarities

Cyclic structures

YKTLISP (as did LISP370) has declared cyclic structures to be first class data objects. Most function which are defined on arbitrary arguments will accept such structures and return reasonable results. Included in this set are READ, PRINT, COPY and EQUAL. In fact the concept of EQUAL required rethinking in this context, see the LISP370 manual for a discussion of EQUAL and UEQUAL.

Many of the functions which operate on lists have been modified from their LISP370 definitions to be tolerant of cyclic lists. Thus MEMBER, ASSOC, and other list searching functions will (with a little extra overhead in the cyclic case) return () when asked to search a circular list. Other functions, such as LAST, APPEND, etc. which are expected to actually find the end of a list will take an error break. The code has been written to penalize the offenders, while adding almost no overhead to uses on non-cyclic lists.

Note that not all list-processing functions are so robust. For example INTERSECTION and UNION will both loop if given cyclic arguments.

Most LISPEDIT commands will also tolerate cyclic structures. The display will show as much of the equivalent infinite representation as fits on the screen. Some messages will show a labelled form.

Debugging compiled code versus interpreted code.

The interpreter of LISP370 and YKTLISP is not written in LISP. It is part of the "core" S/370 assembler code of the system. The interpreter is a direct implementation of an SECD machine definition of the basic semantics of YKTLISP. This has one important effect on the debugging process.

When you are in the break-loop you can examine the frames on the stack.

If a frame belongs to a compiled function you will be given the name of the function, together with the values of all its variables (by name).

If a frame belongs to an interpreted function two things must be noted. First, in cases where the compiled function has a single frame the interpreted function will have a separate frame for each level of LAMBDA binding, with at least one additional frame for the interpreter itself. Second, there is NO way to determine the "name" of the function, all frames being attributed to the interpreter, SECD or SECD1. In addition the SECD machine's S and C stacks will be displayed to you in their full inscrutability.

The only current and up to date definition of the SECD transition states consists of the comments in the ASSEMBLE file for the core of the system.

Of course if you are using LISPEDIT this is mostly moot, as the use of HEVAL allows you to run the interpreter in an interactive mode, with various controls on the stepping rate, and a full display of the code being interpreted.

Values and functions.

Probably the greatest difference between LISP370/YKTLISP and the most widely used LISP dialects (MACLISP and INTERLISP) is the lack of so-called function value cells. In YKTLISP operators are evaluated in exactly the same manner as any other variable.

In one sense assignment is equivalent to definition. (In practice it is not, in that DEFINE does a certain amount of processing on the expression given it. But in the end the resulting expression IS simply assigned as the value of the name.)

This means that a function which binds LIST as a variable may not behave as expected if it attempts to use LIST as an operator. (Of course it MAY behave as expected if you know what to expect, that is if you have a functional value for LIST and it is the function you wish to use when you say LIST.)

In addition, a name can be a function or a macro, but not both (we never look at the property list of an identifier during evaluation). This problem is surmounted by the use of different environments (embodied in state descriptors, SDs, see below) for compilation and interpretation.

Macros

In YKTLISP expressions beginning with MLAMBDA (as opposed to LAMBDA) define macros. (The compiler transforms these into MSUBRs (also known as MBPIs).) During interpretation or compilation if an operator evaluates to a macro the macro is applied to the entire form and the resulting value is treated as if it had occurred in the place of the form.

For technical reasons an MLAMBDA expression is treated as if it were APPLXed to its form. See the section on structured bound variable lists for examples of MLAMBDAs.

If an operator evaluates of anything but a macro or a special-form (e.g. SETQ, COND) the arguments are evaluated and then the operator value is again examined.

If it is applicable (a LAMBDA expression or a BPI for example) it is applied, otherwise it is re-evaluated until it becomes applicable or a constant. If, during the re-evaluation it becomes a macro an error is signaled, as the original form has been lost, and "argument" which the macro might have used unevaluated have be evaluated, with all concomitant side effects.

In compiled code all arguments are evaluated and then the operator is fetched. (The fiction is that (FOO X Y Z) is first transformed into (CALL X Y Z FOO), and only then compiled.) If an operator which was not a macro at compile time evaluates to a macro a run time the same error break is taken, viz. "Dynamic macros not allowed."

What about the values compile macro and contexts?

Scope of variables.

YKTLISP has dynamic lexical scope. In general a variable is only "visible" in the immediate context of its binding. Consider a LAMBDA expression which is the value of some variable, say FOO.

If X is bound by that LAMBDA, then any X contained in the expression, (even if it is in another, contained, LAMBDA expression) will "see" that binding. Note that an X which occurs in a macro expansion is treated as if it had been present in the original expression.

A use of X in a function called, at whatever depth, from this expression will NOT see this binding. If you wish to bind X in such a way that function which you call can see it you must bind it as a FLUID variable. Unlike many other LISPs this is done directly in the bound variable list. Thus the X in:

```
(LAMBDA (X) (BAR))
```

is invisible to BAR or any function it calls, while the X in:

```
(LAMBDA ( (FLUID X) ) (BAR) )
```

is visible to BAR and all functions called by it.

Structured bound variable lists.

When a LAMBDA expression (or its resulting BPI) is applied to a set of values the binding process can be conceptualize as follows. (The actual process differs somewhat, but the effect is the same.)

Given a LAMBDA expression (LAMBDA bv . body) being applied to values A1, A2, ..., form the list,

(A1 A2 ...)

Now examine this list and the bound variable list, bv, in parallel.
If bv is neither a pair or an identifier, discard the value list and terminate.
If bv is an identifier, bind the value list to that identifier.
If bv is a pair and the value list is not a pair, signal an error.
If both bv and the value list are pairs, repeat the process on their respective CARs and CDRs.
(Note that NIL is not an identifier.)

Let me present a few examples.

```
( (LAMBDA U U) 1 2 3) = (1 2 3)
( (LAMBDA (U) U) 1 2 3) = 1
( (LAMBDA (U . V) (LIST U V)) 1 2 3) = (1 (2 3))
( (LAMBDA (U V . W) (LIST U V W)) 1 2 3) = (1 2 (3))
( (LAMBDA (U V W) (LIST U V W)) 1 2 3) = (1 2 3)
( (LAMBDA (U V W . X) (LIST U V W X)) 1 2 3) = (1 2 3 ())
( (LAMBDA (U V W X) (LIST U V W X)) 1 2 3)
    = Error break, non-conformal arguments
( (LAMBDA ( (U . V) ) (LIST U V)) "(1 2)) = (1 (2))
( (LAMBDA ( (U V) ) (LIST U V)) "(1 2)) = (1 2)
```

Note that because of the "constant" rule, excess arguments are simply discarded, not considered an error, while missing arguments are treated as an error.

Macros (MLAMBDAs) are a special case, in that the form is treated as if it were the value list. In the next set of examples I will display the macro expansion, rather than the result of an evaluation. In each case we assume that FOO has as its value the indicated MLAMBDA expression.

```
(MLAMBDA U (LIST "PRINT (LIST "QUOTE U)))
  (FOO A B C) --> (PRINT "(FOO A B C))
(MLAMBDA (U V W) (LIST "PRINT (LIST U V W)))
  (FOO A B C) --> (PRINT (FOO A B))
(MLAMBDA (() . U) (CONS "PLUS U))
  (FOO A B C) --> (PLUS A B C)
(MLAMBDA (() U . V) (LIST "BAR U (CONS "FOO V)))
  (FOO A B C) --> (BAR A (FOO B C))
```

Again, note the use of the constant rule, in this case to discard the operator from the form.

Implied PROGN

One change in YKTLISP from LISP370 is the addition of the so-called implied PROGN. This effects LAMBDA, MLAMBDA and COND expressions. In LAMBDA and MLAMBDA expressions the "body" now consists of one OR MORE expressions. These are evaluated in order (as if they had a PROGN wrapped around them), and the value of the last is the value of the expression. The evaluation of an explicit RETURN anywhere in the body will immediately end execution and provide the value.

```
(LAMBDA (X) (PRINT X) (PLUS X X))
```

for example, will first print its argument, and then return twice its value.

In a COND expression the implied PROGN effects the expressions following a predicate.

```
(COND
  ((ATOM X) (PRINT "ATOM) (LIST X))
  ("T (PRINT "PAIR) X))
```

will print ATOM or PAIR, and then return a PAIR.

PROGN itself has been added as a special form to the system.

Scope of GO and EXIT

In YKTLISP the GO statement is strictly bounded by LAMBDA expressions. Control can only leave a LAMBDA expression by a RETURN or by "falling out the bottom", unlike INTERLISP where GO will search for labels in surrounding LAMBDA expressions. GO will leave an SEQ if it is embedded in another SEQ (with no intervening LAMBDAs, of course). In particular, SEQs "scope" labels in the same way as LAMBDAs "scope" variables. (See section on PROG and SEQ.)

YKTLISP allows GO and EXIT expressions in any context. Of course, both should be contained in an SEQ, and the GO should have a valid label available to it. In particular, a GO or EXIT in an argument to a function will abort the application of the function, and keep any later arguments from being evaluated.

New built-in functions and special forms.

YKTLISP has a number of built-in functions and special forms which do not exist in LISP370. (It has also eliminated one, LABEL, which has, to the best of my knowledge, never been debugged in any version of LISP370.) The new special forms are PROGN and CLOSEDFN (in addition provisions have been made for a CASE operator, but it is yet to be implemented), while the action of FUNARG has changed.

(PROGN exp1 ... expn) evaluates its expressions in turn, returning the value of the last as its value. It does not create either a binding (LAMBDA) contour, or an SEQ (scope delimiter for EXIT).

(CLOSEDFN LAMBDA-expression) acts like QUOTE when interpreted. The compiler compiles the LAMBDA-expression, producing a BPI. This allows compiled function-valued objects which are not funargs.

Funargs have become distinguished objects in YKTLISP, rather than interpretations of lists of the form (FUNARG exp sd). FUNARG as an operator creates a funarg object with the (unevaluated) components exp and sd. (Note that while PRINT will display a funarg correctly, PRETTYPRINT does not. Since the READ-EVAL-PRINT loop uses PRETTYPRINT, you may see ugly items of the form 93xxxxxx. These are the hexadecimal representations of the pointers to funargs.)

The new built-in functions are APPLY, CALLX, CLOSURE, FUNARGP, MDEF, MSUBRP, REALVECP, REFVECP, SUBRP and WORDVECP. In addition RETURN and EXIT, defined as special forms in LISP370 are now defined as built-in functions.

FUNARGP, MSUBRP, REALVECP, REFVECP, SUBRP and WORDVECP are simply the predicates for the associated data types. All of them had non-built-in functional definition in LISP370.

CALLX is a function (exactly equivalent to MACLISP's FUNCALL) which is used in the intermediate "pure" LISP code which is produced by pass I of the compiler. It is the "evaluate operator first" form of CALL.

APPLY is analogous to APPLX, with a third argument, which must be a state descriptor.

(APPLY fn valuelist sd)

will apply fn to the valuelist in the environment defined by sd.

MDEF is to MDEFX as APPLY is to APPLX. That is,

(MDEF macro form sd)

will apply the macro to the form in the environment defined by sd.

CLOSURE creates arbitrary funargs.

(CLOSURE exp1 exp2)

is equivalent to

(EVA1 (LIST "FUNARG exp1 exp2))

STATE and state descriptors

LISP370 and YKTLISP have a data type called a "state descriptor" (SD for short).

SDs are created by the built-in function STATE (and incidental by the creation of closures) and are used in three distinct but related ways. An SD captures and retains the state of the computation.

It must be emphasized that what is captured is NOT a set of name/value relationships, but a set of name/value-cell relationships. Repeated uses of a SD need not result in the same value, but will share the use of the same value-cells.

An SD may be; included as one component of a closure (funarg), given to EVAL, APPLY or MDEF as a final argument, applied to an argument.

The value of a LAMBDA expression, when it is not used as an operator, is a funarg.

```
(SETQ X (LAMBDA (Y) Y))
```

results in a value of X of

```
%.FUNARG.((LAMBDA (Y) Y) . %.SD.xxxxxxxx)
```

(Note that %. is used as a prefix in printing essentially unprintable objects, that is objects which can not be read back after being printed.) Note also that LISP370 fully supports so-called "upward funargs".

Let us suppose that the following expression has been evaluated.

```
(SETQ TEST ( (LAMBDA (X) (STATE)) 12))
```

The resulting value in TEST will be an SD. Then the following results would ensue.

```
(EVAL "X TEST) = 12
(APPLY "(LAMBDA (Y) (CONS X Y)) "(10) TEST) = (10 . 12)
```

Now, suppose we evaluate the following

```
(PROG (S)
  (SETQ S (STATE))
  (COND
    ( (STATEP S) (SETQ TEST S) (PRINT 'First time.') (RETURN 0) )
    ( "T (PRINT 'Again?') (RETURN S) )))
```

The result would be the printing of 'First time.' and a value of 0. If at any future time the expression

```
(TEST "BOO)
```

were evaluated, the result would be the printing of 'Again?' and a value of BOO.

When the function STATE is called the result is an SD, which has captured the state of computation. If that SD is applied to an argument, the computation is resumed at the return from STATE, however the value returned is the value of the argument of the SD. This resumption may be done as many times as desired.

This allows rather straight forward programming of back-tracking, coroutining and other hyphenated control structures. The use of SDs does exact a price in overhead, due to the retained stack frames which must be processed during garbage collection, and in the invalidation of shallow-binding cells when contexts are switched. Local experience has been that once an algorithm has been developed and debugged using SDs, it should be examined to see if it can be re-implemented without them. They make the initial development much simpler, but for much used processes they may prove expensive.

STATE will accept an optional argument, a global environment. This is an A-list like structure, of the form

(() (id . v)* ... <sd>)

The leading () is required. See the section on Evaluation for details on the use of this structure. If the A-list is omitted the resulting SD is given the current global A-list.

The value of STATE captures both the environment (for EVAL) and the control chain (for application). If it is to be used only for evaluation this is an expensive weight to carry around.

To allow "cheaper" SDs, where they are not to be used for resumption of execution, two functions are supplied, EXTSTATE and GLOEXTSTATE. They both take three arguments, a list of identifiers, a list of values, and a SD.

(EXTSTATE "(A B C) "(4 6 9) sd)

They return a new SD which has the same set of bindings as the old SD (the third argument) with the given identifiers bound to the values "ahead" of the old bindings. In addition the new SD does not capture the control at its point of creation, but acts as if it had been created in the same control environment as the given SD. Thus on returning from a function where one of these functions was called the frames can be discarded.

EXTSTATE adds the new bindings to the head of the environment stack of the old SD, while GLOEXTSTATE adds them to the head of the old SD's global A-list.

& -- the back-trace function.

The stack examining function, ?, described in the LISP370 manual has been replaced by a new function, &. (? is still present, just not used.) The most immediate difference is that & receives its arguments unevaluated. Thus

(& full 3)

replaces

(? "full 3)

& normally uses a special print routine which will only write one line per value, eliding large objects. & also accepts a number of new parameters.

(& UNWIND) will display a list of all stack frames which will "stop" an UNWIND, with descriptive titles for those it recognizes. One can then (UNWIND n) to return control to some particular frame, by counting (zero origin) back in the list.

(& ... p ...), where p = PP, PR or (PR fn), will cause the value displays to be done this, respectively PRETTYPRINT, PRINT or fn, rather than with the one-line printer.

(& (var1 ... varn) ... VAL ...) cause a search for the first frame binding any one of the variables vari, and the returning of the value of the first found variable as the value of the call to &. Usually a single variable in the list is most useful. (& (X) 5 10 VAL) will return the first value of X in frames 5 through 10 that is found.

(& (var1 ... varn) ... VALLIST ...) is similar to VAL, but a list of ALL relevant values is returned.

& normally searches for the nearest frame used by the function EVA1FUN and numbers the frames from 1 up, starting with the frame above the EVA1FUN frame.
(& ... (FL <n | id>) ...) will cause the frame numbering to begin, respectively, n frames above &'s frame, or, if the value of id is a number, as if that had been used, if the value of id is a BPI, with the first frame used by that BPI. Thus (& INDEX (FL FOO)) will display the stack index, treating the first frame used by the function bound to FOO as frame 1.

CATCH and THROW.

YKTLISP has a MACLISP-like CATCH and THROW facility. The standard form is

(CATCH name expression id)

Where name is an identifier, and id is a variable name. The expression is evaluated, and in the normal case, its value is the value of the CATCH expression, while the variable, id, will be set to (). If during the evaluation of the expression a THROW of the form

(THROW name-exp t-exp)

is executed, where the value of name-exp (in the THROW) is EQ to name (in the CATCH), then the value of the CATCH is the value of t-exp, while the variable, id, is set to name. Thus after the CATCH the program can unambiguously determine whether control returned normally or not. The final argument, id, is optional, so MACLISP style CATCHes will work. Note that the first argument, name, is NOT evaluated in the CATCH, but is evaluated in the THROW.

Another form provided is THROW-PROTECT.

(THROW-PROTECT expression1 expression2)

evaluates the two expression in order. If a THROW occurs during the first evaluation, which would return control to a CATCH above the THROW-PROTECT, the second expression is evaluated "on the way through". Of course a THROW during the evaluation of the second expression is NOT intercepted.

UNWIND is implemented using THROW, so UNWINDing from an error-break may cause THROW-PROTECT expressions to be activated.

Other forms using the CATCH/THROW logic are ERRSET, ERRCATCH and NAMEDERRSET.

(ERRSET expression) evaluates expression, establishing a CATCH point for UNWIND. If the expression evaluates normally its value, V, is made into a list, (V) and returned. If an UNWIND is done to this CATCH point, its value is passed on as the value of the ERRSET. This has the ambiguity of the original ERRORSET, in that UNWIND can provide a value which is a list, an thus not distinguishable from a normal value.

(ERRCATCH expression id) behaves like a (CATCH xx expression id), but instead of establishing a named CATCH point sets up an UNWIND CATCH point. Here, the ambiguity of ERRSET is avoided.

(NAMEDERRSET name exp) behaves like ERRSET, in that the value is made into a list for normal return, and not for UNWIND. Unlike ERRSET, NAMEDERRSET will also stop a THROW to name, treating it like an UNWIND.

All these facilities are implemented with the under-lying primitive S,CATCH.

(S,CATCH expression id message) binds the value of message to the variable CATCH,MESS and evaluates expression. If the evaluation completes, id is set to NIL and the value is returned. If a (THROW name t-exp) is executed, id is set to the value of name, and the value of t-exp is returned. A value of NIL for name is an error. All of the "user level" operators, above, do a S,CATCH and examine the resulting value of id, then either return the value, signaling normal/THROW, or THROW to the next CATCH point above them.

UNWIND throws a "name" which is a number, and each CATCH point which can stop an UNWIND checks it for zero. If it is not zero, the next THROW will use name-1. Thus the number is counted down until some one claims it, or until the root of the stack is reached. In that case (as with THROWs to unused names) the initial READ-EVAL-PRINT loop is restarted.

SUPV and the break loop

When you first start YKTLISP (not in LISPEDIT), you are confronted with an EVAL supervisor. This is a program, SUPV, which reads an expression from the console, displays the expression back to you, evaluates it, using the interpreter, and displays its value. It then goes back to the READ, to wait for a new input.

The first thing it will do is print "Value = n", for some value of n. This is simply the number of generations of FILELISP that this particular system has been through.

The echoing of input and the display of values can both be turned on or off, independently, by the functions SET-ECHO-PRINT and SET-VALUE-PRINT. Called with arguments of () they turn off the appropriate function, called with non-() arguments they turn it on.

The functions LAST-VALUE and LAST-EXP will return the last value produced, or the last expression read. Thus if you entered an expression and want to save the value you can type

(SETQ X (LAST-VALUE))

If an error is signaled by any process, or if you force an external interrupt, you will be put in the "break loop". This is similar to SUPV. It differs chiefly in not echoing input, and not prefacing the values with "Value =". LAST-EXP and LAST-VALUE do not work in the break loop. Furthermore, if you enter a single ? the break loop will repeat the original error message.

In either SUPV or the break loop entering a "null" line (hitting the ENTER key without any input) will result in a message indicating where you are. SUPV prints "LISP", the break loop prints "BREAK". Immediately upon entering, the break loop clears any stacked lines in the console input and holds them in a variable STACK. This can be examined using &, but there is no provision for re-stacking them.

To exit from the break loop you must either enter (FIN exp) or (UNWIND n). Many entries to the break loop are not re-startable, and FINing in those cases is equivalent to (UNWIND 1). If an error is recoverable, the error message MAY tell you what you should do. (Not always though, there are still a lot of loose ends around.) If you FIN the value (of exp) will be tested against a filter set by the function which reported the error. If it passes, all well and good. If not you will receive a message, such as, "... FIN with small integer, ... Repeat with correct type." "Still in break loop."

One characteristic of READ should be pointed out here. Since READ treats its input as a continuous stream of characters, ignoring end of lines, any input to either SUPV or the break loop must end with a delimiter. The closing parenthesis of a list, or > of a vector will do, but if you wish to enter a simple identifier or a number you MUST explicitly type a trailing

blank before you hit ENTER, to inform the reader that there are no more characters to follow. (See the example in the section on XEDIT.)

Taking over the error handler.

When an error occurs a function (in the file ERROR LISP370) CONDERR is called with four arguments. The first is an error "channel", a number, currently from 1 to 24. CONDERR calls a second function, S,PROGRAM-EVENTS, with this number as its argument. S,PROGRAM-EVENTS checks for a FLUID or global binding of the identifier PROGRAM-EVENTS in the control chain, and if it finds one searches its value with ASSQ for an entry of the form (n fn . string). If one is found its CDR is returned to CONDERR. Otherwise the number is used to access an internal table in S,PROGRAM-EVENTS and a pair, (fn . string) is returned.

CONDERR constructs an error message based on the string in the value of S,PROGRAM-EVENTS and its other arguments, and then applies fn, from the value of S,PROGRAM-EVENTS, to three arguments, the channel number, its third argument (?ARGS?) and the message it has constructed. (See the file ERRORS NOTES for a list of all error messages in the BASE system.)

To co-opt this machinery FLUIDly re-bind PROGRAM-EVENTS to a list of the form

((1 fn1 . stg1) (2 fn2 . stg2) ...)

with your choice of functions for the fn's.

Evaluation

Identifiers are normally evaluated by searching up the stack. The search commences in the current stack frame, looking at all variables. Once it passes a "closed" contour, that is a frame corresponding to a non-lexically present LAMBDA (see section on Scope of variables) only FLUIDly bound variables are looked at. If the variable is not found in the stack the current global A-list is searched. If it is not found and if the A-list terminates in an SD the global A-list associated with that SD is searched, and so on until the variable is found or a non-SD terminator is reached.

(Note that this search is often avoided by the use of shallow binding cells. In practice, once the search has left the immediate (lexical) scope of the variable a "look aside" is performed. If the variable has a shallow binding cell (not all do) and if that cell is current (that is it was last set in the current environment) then it is used to access the current binding. If it is not current the search is carried out and the shallow binding cell is "refreshed", to avoid further searches, until the environment is changed again.)

The frames searched during this process are normally the same as the "control chain", that is the chain of stack frames from called function to calling function. However, whenever a funarg is applied or EVAL, APPLY or MDEF are called a fork occurs, with the environment chain following the stack frames captured by the SD involved. In addition, in each of these cases the global A-list in the SD becomes the current global environment.

Since error handling is most usefully thought of as a function of the control, a special function, S,CEVAL-ID is supplied, which searches the "control chain" for a binding of its argument, and returns its value if it exists. Like normal evaluation, an "unbound" variable evaluates to itself. No global environment is searched by this function.

A second function, S,EVAL-GLOBAL-ID, will search only the current global environment for a value.

External interrupt

When YKTLISP is loaded a external interrupt handler is enabled. If at any time CP is entered and EXT typed, a request for interrupt is posted in LISP. This request is polled on every function entry, function exit and also in all loops in LISP function. When the request is detected the break loop is entered. An (UNWIND 1) or a (FIN x) will cause execution to resume at the point of interruption. Note that if LISP is waiting for input it will not see the interrupt request, it must be actively running.

PROG and SEQ

In YKTLISP the form (PROG bv . body) is not primitive. PROG is a macro which expands effectively as follows.

```
(PROG (X Y Z) . body)
   --> ( (LAMBDA (X Y Z) (SEQ . body)) () () ())
```

There is actually some further processing.

First, the YKTLISP definition of PROG states that if control leave by "falling out the bottom" rather than by a RETURN, the value of the PROG expression is (). To ensure this the macro scans the body, and if it can not be certain that all exits are by RETURNs it adds a (RETURN ()) to the end.

Then, if the body consists of a single expression the SEQ is omitted.

Finally, the local variable list may contain initial values. If one writes

```
(PROG ( (X exp1) Y (Z exp2) ) . body)
   --> ( (LAMBDA (X Y Z) (SEQ . body)) exp1 () exp2)
```

will result.

Note that the expressions are evaluate outside the LAMBDA expression. This allows the "pushing down" of variables by use of expressions such as

```
(PROG ( ( (FLUID FLAG) FLAG) ) . body)
```

While body is evaluated the FLUID variable FLAG will initially have its previous value. Any assignments to FLAG within body will be available to function called from there, but upon leaving the PROG, FLAG will revert to its previous value.

The rational behind these constructs was the desire to limit variable binding to a single form, LAMBDA (considering MLAMBDA as simply a variant), as well as a desire to separate the scope of variable from the scope of labels. In order to accomplish this a new form, SEQ, was invented. SEQ is to labels as LAMBDA is to variables.

YKTLISP (as noted elsewhere) does not allow GO to cross variable binding contours. Thus GO can not leave a PROG. SEQ on the other hand binds no variables, it simply delimits a set of labels. A GO will attempt to find its target label within the immediately enclosing SEQ. If it fails it will then search more remote enclosing SEQs until it either locates its target label or encounters a LAMBDA contour, in which case it fails. Just as a variable bound by a LAMBDA can not be seen from outside that LAMBDA, a label within an SEQ can not be reach by a GO in a enclosing SEQ. This provides a clean mechanism for isolating labels in mechanically generated pieces of code.

READ <--> PRINT

To as great an extent as possible READ and PRINT in YKTLISP are symmetric. That is, if an object is PRINTed to a file the result of READing from that file should be UEQUAL to the original object. Similarly, if one READs an object form a file and then PRINTs it the resulting representation should match the original.

Both of these ideal are approached, but not reached.

In the first case, there exist within YKTLISP certain data objects which have no READable representation. These include compiled programs (BPIs) and state descriptors (SDs). Both will PRINT, but only as %. forms, which can not be READ.

In the second case there are several areas of failure. The representation of numbers is "many to one", in that 127, for example, can be written as 127, 00127, or %X7F, but will only be PRINTed as 127. In a similar manner extra blanks will be lost, as will gratuitous "letterizer" characters, e.g. |A|B|C|D will print as ABCD. Finally, PRINT favors list notation over dot notation, so

```
(A . (B . (C . ())))
   --> (A B C)
```

(This is particularly annoying when one has written a FLUID indefinite trailing argument in a LAMBDA expression, so that

```
(LAMBDA (X . (FLUID Y)) ...)
   --> (LAMBDA (X FLUID Y) ...)
```

The interpreter and compiler do just fine, but reading the listing takes getting used to.)

What are all those Qs and commas doing?

LISP has always had a few conventions for naming functions with common attributes. Most "true/false" functions (predicates) have names ending in P, for example. YKTLISP has several conventions of its own (as with the ...P they are not 100% consistent).

You will quickly notice many functions with commas imbedded in their names. These all were originally thought of as "system" functions, which the casual user would not be interested in. As the comma has no special syntactic meaning in LISP370/YKTLISP, but acted as a list separator in the previous LISP available at Yorktown this was felt to be a protection from name clashes.

A few functions have names which start with a comma. These are usual "under cover" versions of normal functions. Thus ,PLUS is a two argument generic addition function, used by PLUS which is the multiargument generic addition function.

Many other functions have names prefixed by one or more letters and a comma. The prefix groups the functions, e.g. parts of the compiler start with C, parts of the LAP assembler with L, and general system functions with S,.

Many other functions will be seen which start with a Q. This stands for "QUICK", and can have one of two meanings. In all cases it implies the existence of a macro in the compile environment which produces in-line code. In many cases it also implies a lack of type checking.

So QCAR and QCDR do not test their arguments for PAIRness, they assume correct type and act accordingly. The compiler is, at present, not smart enough to elide the type checks in code such as

```
(COND ((PAIRP X) (CAR X)) ("T X)
```

so the careful programmer is allowed to circumvent the built in checks by writing

```
(COND ((PAIRP X) (QCAR X)) ("T X))
```

Other of the Q operators do type checking, but are quick in that they result in inline code. These include QMEMQ and QASSQ.

Another group of Q operators are the QS... operators. These (QSPLUS, QSADD1, etc.) assume that their arguments and values are small integers, (numbers between -(2*26) and 2*26-1). general they do arithmetic modulo 2*26, forcing correct small integer type codes on their results. They are also aware of each other, and will skip the forcing of type codes on some intermediate results when nested.

Note that there are a few functions starting with Q which are neither inline nor unchecked. These include QSORT and QUOTIENT.

Functions whose names end in Q are usually version of other functions which use EQ rather than EQUAL. Such pairs include MEMBER/MEMQ, ASSOC/ASSQ, UNION/UNIONQ.

"T

YKTLISP does not treat the identifier T specially. Thus in the "otherwise" clause of a COND you should quote the traditional T. (Any constant will suffice, some people favor 1). If you write

```
(COND ((PAIRP X) (PRINT (CAR X))) (T (PRINT X)))
```

You could be surprised if some function contained

```
(SETQ T ())
```

with T free.

Property lists

YKTLISP's property lists are ASSOC type lists, not flat list as in INTERLISP. If you evaluate

```
(MAKEPROP "X "A 1)
(MAKEPROP "X "B 2)
(MAKEPROP "X "C 3)
```

then

```
(PROPLIST "X) = ((A . 1) (B . 2) (C . 3))
```

Note that the value of PROPLIST is not the actual list from the identifier. The top-level pairs have been copied, as in APPEND. The property name/value pairs are "real", so updates on them will be reflected in the actual property list.

Since YKTLISP relies on multiple environments and uses assignment for function definition, the use of properties to annotate functions is of little or no value. There is still one (potentially incorrect) vestige of this in YKTLISP, with various identifiers having the property SAFE.

N.B. the description of DEFLIST in the LISP370 manual is incorrect. The second argument is a list of length two lists, with the CAR being the identifier receiving the property, the CADR being the value.

Sub-mini guide to LISPEDIT

If you enter LISPEDIT, by evaluating (EDIT exp) for example, you need to know at least six commands in order to do anything.

TELL item
Will present you with a block of text about "item". If you are on a remote terminal the text will be assigned to the FLUID variable =, if you are on a non-remote terminal you it will be put in a recursive call to EDIT, editing the text. In the first case you should immediately enter
EDIT = or E =

LISPEDIT views most objects as lists, and the change of focus is with respect to the list structure. You can "move" to the right or left within a list, up to a containing list, or down to some element of a list.

SON n or S n
If you are currently focused on a list (the current focus is bright unless it is the entire object being edited) S n will move the focus to the nth element of the list (with the CAR being counted as 1). S * will move the focus to the last element, while S -2 will move it to the next-to-last.

UP n or U n
If you are currently focused on an element of a list U 1 will move the focus to the list itself. U 2 is equivalent to entering U 1 twice in succession.

RIGHT n or R n
If you are currently focused on an element of a list R n will move the focus n elements to the right. R * will move to the rightmost element.

LEFT n or L n
If you are currently focused on an element of a list L n will move the focus n elements to the left. L * will move to the leftmost element.

TELL WHAT
Entering TELL WHAT will present you with text which describes all the commands available in LISPEDIT.

not adequate

Versions and systems

As is explained in the YKTLISP MEMO, a YKTLISP system is embodied in two files, a SEGMENT file and a SHLISPWS file. These files are given names of the form SSSSnnnn, where SSSS designates as particular system, and nnnn a particular version. For example, LEDT0050 is the current (version 0050) LISPEDIT system.

A system may have only a SHLISPWS, using a SEGMENT with another system name.

The most commonly used systems are
COLDnnnn -- An "empty" LISP, can only coldstart (see YKTLISP MEMO).
EVALnnnn -- A "stripped down" system. No compiler or assembler.
BASEnnnn -- A LISP system with a compiler and assembler, but no LISPEDIT.
LISPnnnn -- A full LISP system, with LISPEDIT.
LEDTnnnn -- A SHLISPWS with LISPEDIT activated.

If you have defined a LISP shared segment on your VM system you should save the LISPnnnn SEGMENT as the shared version. Otherwise, you may wish to run with the smallest system that will support your needs, as the SEGMENT will be loaded into your own virtual memory.