

# New Features in MINT-3

M. D. Godfrey

**Compiler Level: 3.0**

**VM version: 1.2**

November 12, 2000

Updated: December 26, 2002

May 27, 2003

January 2004

## Contents

<b>1.0</b>	<b>Introduction</b> .....	1
<b>2.0</b>	<b>Dictionaryess</b> .....	2
2.1	Introduction.....	2
2.2	Dictionary Class and Operators.....	2
2.3	The Dictionary List .....	6
2.4	Details of Dictionary Manipulation Procedures.....	8
2.5	Compiler Dictionary Manipulation Procedures .....	10
2.6	Definition of Dictionary Records.....	10
2.7	Dictionary and Identifier Displays.....	12
2.8	Use of NOW and PDUMP.....	12
2.9	Auto-compilation Facilities .....	13
<b>3.0</b>	<b>B-Tree Data Access</b> .....	14
3.1	B-tree Functions.....	14
3.2	FORBTVAL .....	16
3.3	Data Structures .....	16
<b>4.0</b>	<b>Other New Features</b> .....	17
4.1	Precedence Mechanism (PRIORITY) .....	17
4.2	Deleting an Item List .....	17
4.3	Operation on Characters of a String (FORCHS).....	17
4.4	Input and Output Enhancements .....	18
4.5	New Features in the Portable 32-bit Virtual Machine.....	18



# 1. Introduction

This note describes the main new features of the MINT system since MINT 2.2 as described in the Revised Second Edition of the book. The most significant changes are the introduction of dictionaries, inclusion of priority (shunt factor) in the dictionary record (rather than the CLASS record), and the extension of the system to 32-bit words and use of a large (typically  $2^{24}$  words) virtual address space. Strings are now stored 4 characters per word.

The new dictionary mechanism is used to improve auto-compilation of the system. Auto-compilation is now much simpler, more flexible, and more clearly realized within the standard MINT facilities. One specific improvement is that the layout and contents of dictionary records may be changed during auto-compilation in a relatively convenient manner. This will facilitate further planned developments.

Since the precedence is not declared by the CLASS directive the precedence field has been removed. Thus, CLASS declarations have three instead of four arguments. All MINT code using CLASS must be changed. It may also be necessary to include PRIORITY statements to set the precedence that was declared in the CLASS directive.

These changes and additions have resulted in a reduction of the listing length of the compiler, including the auto-compilation procedures and the B-tree functions, from 58 pages to 55 pages.

The new C-coded reference VM makes use of the environment variable MINT\_HOME. When the VM starts execution it normally loads a PDM-format MINT system from the file MCOMP.PDM. If MINT\_HOME is defined, the VM uses the path defined by MINT\_HOME to locate the file MCOMP.PDM. Otherwise it uses the current directory. The command option -f can be used to specify another file to be loaded instead.

PDM format is sensitive to the “byte order” of the CPU being used. However, if the byte-order appears to be incorrect for the CPU architecture which is in use, the order is reversed during PDUMP reads. The -v option will indicate what the VM thinks is required during PDUMP load.

## 2. Dictionaries

### 2.1 Introduction

The MINT dictionaries provide the information which causes operation of the entire system. In order to provide control over the currently known sets of identifiers the identifier records are composed into distinct dictionaries. These dictionaries are composed into a list. The list of dictionaries may be manipulated by referencing dictionary names and by procedures which are described below. New dictionaries are created by the DICT construct. The operators \ and % may be used to directly reference a named dictionary. The dictionary list may also be referenced directly by use of the MINT list operators. Pointers into the list structure control the way in which the various dictionary manipulations are carried out. Thus, it is possible to select which dictionaries are searched and which dictionary is used for introduction of new identifiers.

### 2.2 Dictionary Class and Operators

#### 2.2.1 The CLASS DICT

Class DICT is used to introduce a new dictionary. Its form is:

DICT <dictionary name>

After introduction the dictionary is set by:

<dictionary name>: HDICT

where HDICT is a macro that builds the data structure required for a dictionary. Typically, a dictionary will be introduced and set by, for example:

DICT DC1: HDICT

After a dictionary name has been introduced and set it may be referenced, much like a DIR. Referencing a dictionary name causes that dictionary to

become the current *active* dictionary. If the dictionary had not previously been referenced, it is initialized and pushed onto the dictionary list (see below). The compiler's dictionaries, MAINDIC and INTDIC, may be referenced in this way. However, it is recommended that the user introduce and use his own dictionaries. Thus,

```

    DICT DC1: HDICT
    DICT DC2: HDICT
    .
    . text section 1
    .
    DC1
    .
    . text section 2
    .
    DC2
    .
    . text section 3
    .
    DC1

```

will have the following effects:

1. The two new dictionaries DC1 and DC2 are created. Within text section 1 new introductions are made into the dictionary that was previously active.
2. Within text section 2 new introductions go into dictionary DC1.
3. Within text section 3 new introductions go into dictionary DC2.
4. After the last line (DC1) operation continues using DC1.

### 2.2.2 The \ and % Operators

Two operators permit *transient* use of dictionaries, i.e. the dictionary name which follows the operator is used once and then the dictionary structure is returned to its previous state. The two operators are \ which causes identifier lookup in the specified dictionary, and % which introduces an identifier into the specified dictionary. Thus,

```

DICT DC2: HDICT
DICT DC4: HDICT
      DC2
VAR DCVAR:25
      DC4
VAR DCVAR:50
FN FF: ENTRY
      \DC2 DCVAR ->@TEMP
EXIT

```

is a function which would generate text to obtain the the value of DCVAR in dictionary DC2 even though dictionary DC4 would normally have been searched first. The % operator directs introduction to the named dictionary, as, for example:

```
%DC4 VAR DCVARX:75
```

would introduce the variable DCVARX into dictionary DC4 regardless of what dictionary was currently active.

### 2.2.3 BLOCK and ENDBLOCK

The BLOCK and ENDBLOCK directives function as they did in MINT-2. ENDBLOCK will always remove the dictionary created by its matching BLOCK regardless of other dictionaries which may have been referenced after the BLOCK directive.

### 2.2.4 SAVBLOCK and SETBLOCK

These directives operate almost exactly as they did in MINT-2. However, in MINT-2 a SAVBLOCK of a directory using a directory address of a directory which already contained entries would result in a merge of the old entries with the new entries. In MINT-3 SAVBLOCK simply saves the named directory.

### 2.2.5 The SETDIC Function

This function provides a means of setting a dictionary's access control. Its general form is:

```
SETDIC(<value>, <dictionary address>)
```

The function sets <value> as the access control for the dictionary whose

address is given. At present the following access controls are available:

Value	Meaning
0	dictionary is not used for lookup (locked)
1	dictionary may be read or written
2	dictionary is read-only (lookup only)

Thus, for example:

```
SETDIC(0, @INTDIC)
```

will exclude INTDIC from searches, and

```
SETDIC(1, @INTDIC)
```

will return it to read/write state so that it will be searched. For some purposes it is more convenient to lock a dictionary rather than to remove it from the dictionary chain. SETDIC may be applied to any dictionary regardless of whether it is in the dictionary list. New dictionaries are always initialized as read/write.

### 2.2.6 The LOCK, RD-ONLY, and UNLOCK Directives

These three directives reference NEXTELT, then SETDIC to set the requested dictionary state. Each requires a dictionary name as its argument. UNLOCK INTDIC is the replacement for ICL\$ and LOCK INTDIC is the replacement for RCL\$.

### 2.2.7 The LASTDIC Function

This function provides a means of setting the point at which the search of the dictionary list should be terminated. LASTDIC expects the top item on the stack to be the address of a dictionary in the dictionary list. This pointer is saved so that subsequent dictionary list searches stop after the dictionary whose address was provided.

### 2.2.8 The ICL\$ and RCL\$ Directives

These directives act in the same way as in MINT-2, but by new means. The RCL\$ directive performs a SETDIC(0, @INTDIC), and ICL\$ performs an

SETDIC(1, @INTDIC).

### 2.2.9 Notes on Dictionary Manipulation

There are two important points to keep in mind when managing multiple dictionaries. First, each dictionary record contains a pointer to the dictionary record for its CLASS. Therefore, items of class CLASS should not be introduced into dictionaries which are then subsequently removed if identifiers of that class are also introduced into other dictionaries which are retained. There is no check that a CLASS pointer still points to the intended dictionary item. Second, if multiple introductions of the same identifier are made it is important to ensure that the point at which the identifier is inserted and the dictionary search order are such that the intended identifier is found. A prominent situation in which this could be a problem is the introduction of identifiers which are the same as ones in INTDIC after an ICL\$ directive. The new identifier will be inserted in MAINDIC, which is searched after INTDIC.

When the compiler is initialized it creates and references a dictionary named USERDIC. Unless there is some special reason to do so, it is better not to introduce new identifiers into the compiler dictionaries, MAINDIC and INTDIC.

## 2.3 The Dictionary List

The dictionaries that are in current use are members of the list whose list pointer is ENVLIST. Dictionary addresses may be pushed and popped from this list either directly or by using the directives described below. In addition there is a pointer to a particular item in the dictionary list which determines the dictionary into which new definitions are added. Initially, this pointer points to the top item in the list.

### 2.3.1 The Dictionary List Structure

The form of the dictionary structure after compiler initialization is:



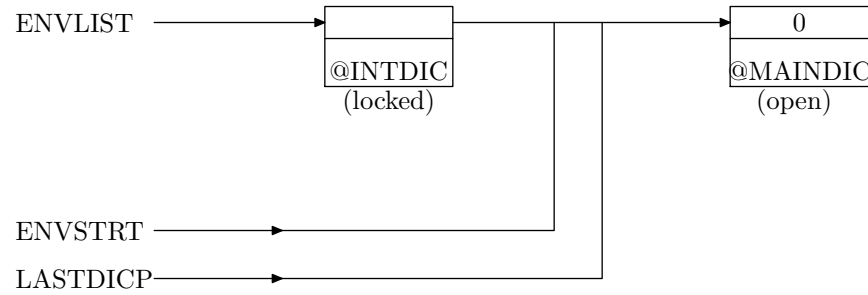


Figure 1. Initial Dictionary List

The items in Figure 1 are defined as follows:

ENVLIST Pointer to start of dictionary list. All dictionary searches start at the dictionary pointed to by ENVLIST.

ENVSTRT Pointer to current *active* dictionary. All new definitions are inserted into this dictionary.

LASTDICP Pointer to the *last* dictionary to be searched. Setting LASTDICP to a dictionary before the last in the list permits searches of “windows” of dictionaries.

The example below shows the form of the list for the case where the compiler has been initialized and then an ICL\$ directive has been obeyed.

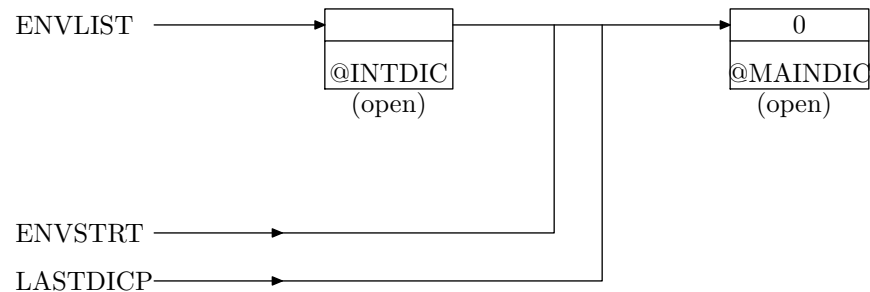


Figure 2. Dictionary List after ICL\$

The ICL\$ directive simply did a SETDIC(0,@INTDIC) to open INTDIC. This has the effect that INTDIC and MAINDIC are searched when an identifier is matched, but new definitions are inserted in MAINDIC. Note

that the requirement to match the longest string means that the entire dictionary list must be searched on all matches.

### 2.3.2 The Compiler Dictionaries

The dictionary MAINDIC is the base dictionary for the system. It contains the standard set of identifiers. Without this dictionary none of the normal MINT identifiers can be matched. While a POPUP(@ENVLIST) will pop this item if it is the only dictionary in the list, this is not a good idea in most cases. When the compiler is initialized the dictionary list pointers are set as shown in Figure 1. An RCL\$ is performed which locks the dictionary INTDIC. The ICL\$ directive unlocks INTDIC.

## 2.4 Details of Dictionary Manipulation Procedures

### 2.4.1 BLOCK and ENDBLOCK

The BLOCK directive performs the following operations:

1. A new dictionary is allocated, initialized to be empty, and pushed onto the dictionary list.
2. The current active dictionary pointer is saved on a stack and the active dictionary is set to the top of the dictionary stack.
3. The address of the new dictionary is saved in an internal list.

The ENDBLOCK directive performs the following operations:

1. The top address is obtained from the list used in BLOCK. This dictionary is removed from the dictionary list. Note that the last dictionary created by a BLOCK directory is the one removed regardless of any dictionaries that may have been pushed onto the dictionary list by other means.
2. The top item on the active pointer stack is obtained and the active dictionary pointer is reset as it was before the previous BLOCK directive.
3. The record space used by the dictionary records and dictionary index table is released.

These actions have the effect that a BLOCK/ENDBLOCK sequence is transparent in the sense that the dictionary list and pointers are put back

as they were before the BLOCK directive, but any non-BLOCK changes to the dictionary list are preserved. Thus, specifically, a dictionary may be added by PUSHNDIC or PUSHODIC after a BLOCK directive and it will remain in the dictionary list after the following ENDBLOCK. For example, if the dictionary list was as shown in Figure 2 and then a BLOCK directive was obeyed, the list would be:

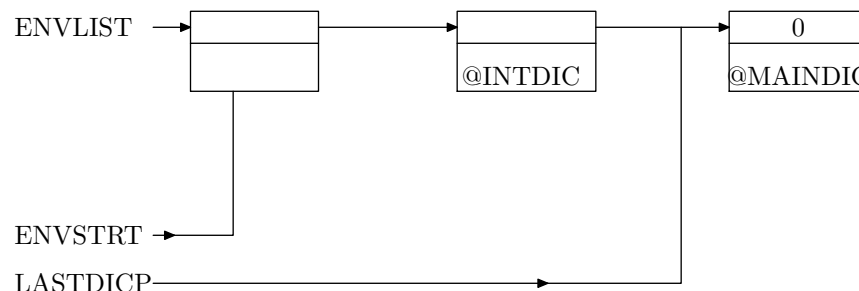


Figure 3. Dictionary List after BLOCK

#### 2.4.2 SAVBLOCK and SETBLOCK

The SAVBLOCK function acts exactly like ENDBLOCK except that the dictionary address is stored at the address provided on the stack and the dictionary records are not released. The SETBLOCK function acts exactly like BLOCK except that the dictionary whose address is pointed to by the address on the stack is pushed, rather than pushing a newly initialized dictionary. Thus,

```

VAR SAVDIC:0
FN SAVRES:ENTRY, SAVBLOCK(@SAVDIC), SETBLOCK(@SAVDIC),
    EXIT
  
```

would perform an ENDBLOCK, but then save the dictionary address in SAVDIC, and perform a BLOCK, but restoring the dictionary as the new top item. Note that SAVBLOCK does not “accumulate” dictionary records as was the case in previous versions of MINT.

### 2.5 Compiler Dictionary Manipulation Procedures

### 2.5.1 PUSHNDIC

This function allocates a new dictionary, initializes it to an empty state, pushes its address onto the dictionary list, and returns the address on the stack. Note that PUSHNDIC does not modify ENVSTRT.

### 2.5.2 PUSHODIC

This function expects a dictionary address on the stack. It pushes this address onto the dictionary list. Note that PUSHODIC does not modify ENVSTRT.

### 2.5.3 POPDIC

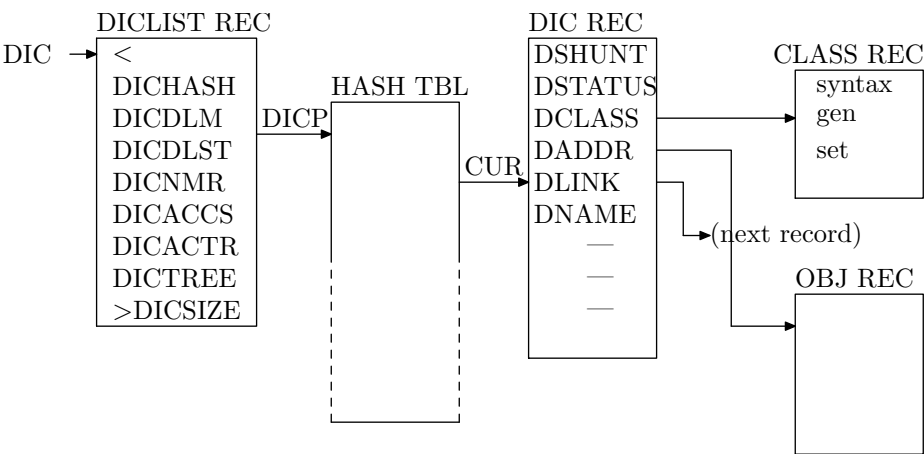
This function pops the top dictionary address from the dictionary list and releases the dictionary and record space. Note that POPDIC does not modify ENVSTRT.

### 2.5.4 ACTDIC

This function sets the active dictionary pointer, ENVSTRT, to the dictionary whose address is supplied on the stack. Identifiers are always introduced into the active dictionary.

## 2.6 Definition of Dictionary Records

Figure 4 shows the definition and contents of all dictionary-related records.



Example Records:

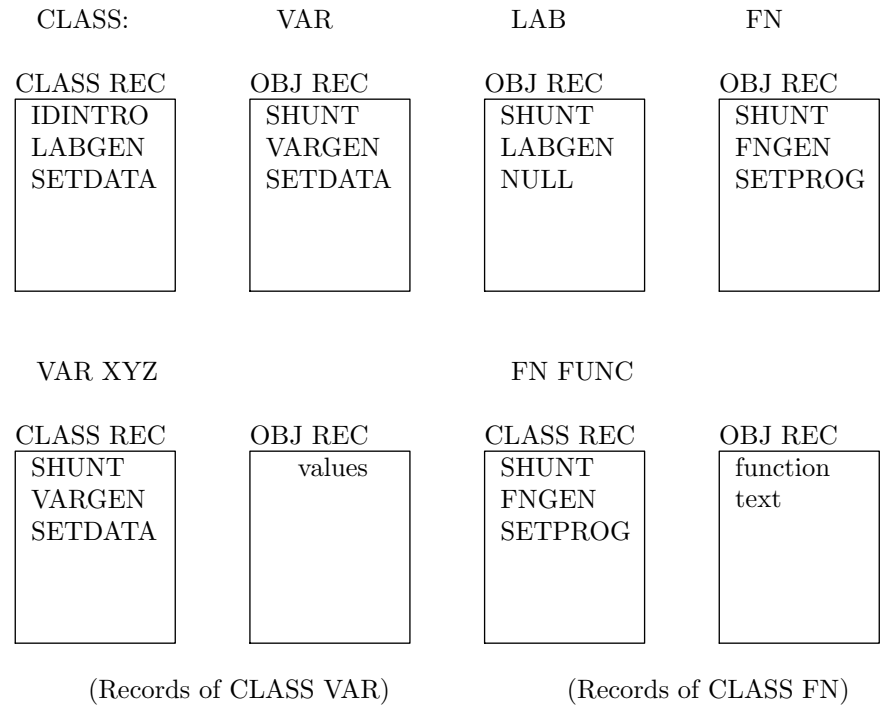


Figure 4. Dictionary Record Structure.

## 2.7 Dictionary and Identifier Displays

The LV\$ directive has been modified so that it displays the identifiers in each of the dictionaries in the current list from ENVLIST through LASTDICP. It also lists the dictionary name and count of items for each dictionary. The new directive, OBJ, provides a convenient means to display the identifiers in a given dictionary. (see below.)

### 2.7.1 Display of Objects

The directive LISTDICS lists the name and item count for each dictionary in the current list from ENVLIST through LASTDICP. The directive OBJ <object name> lists the properties and body of any object whose CLASS is known within the compiler. Specifically, if the object is a FN or DIR the object text will be displayed, and if the object is a DICT, the identifiers in that dictionary will be displayed.

The directive CURDIC will display the identifiers in the currently active dictionary.

The directive PREVDIC will display the identifiers in the “previous” dictionary. On the first use of PREVDIC, after a use of CURDIC, it will display the contents of the dictionary previous to the current active dictionary. If PREVDIC is referenced again, it will display the contents of the next previous dictionary. Further references provide displays of further dictionaries until the end of the dictionary list, at which point the cycle will repeat.

## 2.8 Use of NOW and PDUMP

Note that since NOW references BLOCK and since BLOCK pushes a new dictionary onto the dictionary list, constructs like:

```
NOW PDUMP('SAVE'), GO 32768 !
```

are not a very good idea since an extra dictionary will have been left on the list due to the fact that the ENDBLOCK in ! is not executed. For this reason a directive, CDUMP, is available. CDUMP expects two arguments which are the filename for the PDUMP and the address to which to transfer on reload. If this address is 0 a normal EXIT is taken on reload. Thus, a standard means of creating a PDUMP of the compiler is:

```
CDUMP 'COMP' 32768
```

## 2.9 Auto-compilation Facilities

### 2.9.1 The Dictionary List during Auto-compilation

The text in MINTAUTO introduces and sets two dictionaries, MAUTO and IAUTO. When the AUTO directive is referenced the active dictionary is set to MAUTO. IAUTO is referenced in the text when it is required to introduce identifiers in the compiler internal dictionary. Thus, for the auto-compilation process MAUTO and IAUTO correspond to MAINDIC and INTDIC respectively. The GENSYS directive copies and relocates these dictionaries into data space and converts them into MAINDIC and INTDIC.

### 2.9.2 OLDIC

During auto-compilation a variable, BASESTRT, is used to control the point in the dictionary list at which identifier searches are started. BASESTRT is set to point to INTDIC. OLDIC does a lookup starting at BASESTRT. This causes previously defined identifiers (introduced in INTDIC or MAINDIC) to be found through OLDIC lookups, but new identifiers to be inserted and matched in MAUTO or IAUTO.

## 3. B-Tree Data Access

The purpose of these routines is to provide a B-tree access structure for storing and retrieving data items or records within MINT VSTORE. The routines provide the following functions: initialization, storage of data for a given key, deletion of data, and a function that permits operation on each data item stored under a given key. Nodes in the tree are filled as new keys are stored, and split when they become full. At present, there is no provision for compacting the tree structure or deleting keys. If all the data items for a given key are deleted the key sequence will remain, with an empty item list at the end of it. Leaf nodes are split without increasing the tree depth and all non-terminal nodes are split by introducing a new node that increases the depth of its branch by one.

The node search function is supplied as an argument when a tree is initialized. Nodes contain single-word entries for keys. However, the search function may interpret these entries as addresses and thus may perform the comparison on any user-defined structure.

### 3.1 B-tree Functions

#### 3.1.1 BTINIT

This function creates the initial tree structure. It requires four arguments: the address of a pointer to the tree (for future reference), the address of the key compare function, and lower and upper limits for the expected key values. Thus, a reference is:

BTINIT(<lower limit>, <upper limit>, [key compare function], <tree pointer>)

The *key compare function* is a function which locates the correct entry in a node, given the key. The user is required to provide this function in order to permit the B-tree mechanism to be used with key structures of the user's choice. If the key is a one-word item it may be stored directly in the node. Otherwise, the entries in the nodes should be addresses of key objects (such



as variable length strings). If the key entry in the node is an address then the key values supplied for all other B-tree functions must be addresses of compatible objects. The key compare function must be written to expect three arguments and to return one argument. The three input arguments are:

1. *key*. This is either a key value or the address of a key structure. It must have the form used in all the other function references.
2. *@first entry*. This is the address of the first entry in the table to be searched.
3. *number of entries*. Number of entries to search.

Each entry in the table pointed to by the second argument is two words long. The key address or value is in the first word and a pointer is in the second word. Therefore, the search routine should increment by two as it compares table entries.

The returned argument is the address of the second word (pointer) of the entry before the entry containing the key greater than or equal to the argument key. If no key is found that is greater than or equal then the address of the second word of the last entry is returned.

The address of the tree pointer is required as an argument in all of the functions so that multiple trees may be in use at any time.

### 3.1.2 BTDEL

This function deletes the entire tree and returns all node and data item records to the record pool. Its reference is:

BTDEL(<tree pointer>)

### 3.1.3 BTINSRT

This function inserts a data item under the key provided. Its reference is:

BTINSRT(<data>, <key>, <tree pointer>)

### 3.1.4 BTREM

This function removes all data items stored under the given key. Its reference is:

BTREM(<data>, <key>, <tree pointer>)

3.2 FORBTVAL.

This function applies the provided function to each data item stored under the given key. Its reference is:

FORBTVAL(<key>, [<function reference>], <tree pointer>)

3.3 Data Structures

There are two data stuctures: non-terminal nodes and leaf nodes.

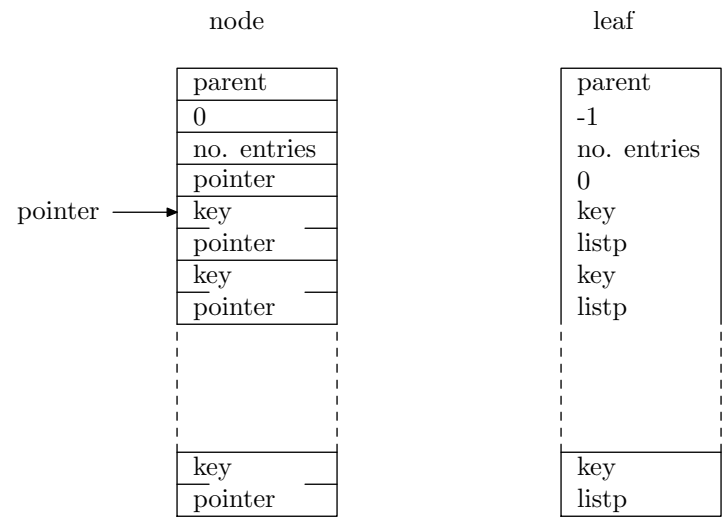


Figure 5. Node and Leaf Data Structures

## 4. Other New Features

### 4.1 Precedence Mechanism (PRIORITY)

The precedence or shunt factor of each identifier has been moved from the CLASS record to the identifier's dictionary record. Due to this change it is possible to set an operator's precedence independent of its CLASS. The directive PRIORITY provides the means of setting precedence. In principle, precedence could be changed dynamically during processing. The use of PRIORITY is as follows:

PRIORITY <n> <identifier introduction>

This statement causes precedence <n> to be applied to the following introduction. After the introduction the default priority is reset to 0. Thus, an example use is:

PRIORITY 4 PRIMOP MASK 15

This is the text used in the compiler to set priority 4 for the operator MASK, and set its value (operation code) to 15.

### 4.2 Deleting an Item List

The function CLEARLST(<@listp>) is available to perform a POPUP on each item in the list.

### 4.3 Operation on Characters of a String (FORCHS)

The FORCHS function provides a means of applying a procedure to each character in a string. It is referenced as follows:

FORCHS(<string address>, [function])

where [function] is a function that expects a character on the stack and returns its character result on the stack. FORCHS obtains a character from the string, references the function, and pushes the top item on the stack back into the character position of the obtained character. This

operation is performed on each character in the string in order.

#### 4.4 Input and Output Enhancements.

The OPINT and OPINTD functions have been enhanced by use of zero value of the width parameter to mean variable width. In addition, the variable CWIDTH is set to the width actually used on each OPINT or OPINTD reference. The width used when WIDTH has been set to zero is the width required to display the digits. Thus,

SETBSE(#0, 0, 10), OPINTD(123)

will display 123 without any leading or trailing blanks. After this operation the value of CWIDTH will be 3.

All input and output functions have been adjusted so that they handle 32 bit numbers.

A function INHEX is included for reading numbers in standard HEX format. It operates just like ININT.

The directive \$ reads the immediately following constant using HEX format. Thus, VAR XX:\$FF would assign the value 255 to the variable XX.

#### 4.5 New Features in the Portable 32-bit Virtual Machine

##### 4.5.1 C-coded Large VSTORE System

A new Virtual Machine has been written in ANSI C. This implementation is highly portable. It runs on any Linux platform and on other systems which have an ANSI C compiler, and support virtual memory. Systems can easily be configured to run in small real memory systems if this is required. The current default VSTORE size is 16M words, but this is easily reconfigured.

##### 4.5.2 Use of environment variable MINT\_HOME

The Virtual machine loader uses the environment variable MINT\_HOME to locate the MINT system files. This variable should contain the path to the “mint3” directory in the distribution.

### 4.5.3 New Debugging Features

A new primitive, VMDEBUG, is available to control Virtual machine debug mode. VMDEBUG pops the stack and stores the value in an internal register which determines VM debug mode. If the register is 0 debugging is turned off, and the VM runs at full-speed. If the register is non-0 VM debugging is turned on and, currently, the VM speed is about 1/2. With debugging on the following actions are taken:

1. Address values are checked for the range  $80 < \text{address} < \text{MAXVSTORE}$ .
2. Instruction values and the current top of stack are pushed into a circular list. In the event of an abnormal condition, the list is displayed and is written to the file: mint-tr.trc.

The option of attempting restart is available after an abnormal stop. On restart, normal entry to the compiler is taken. Of course, this may not always work.

### 4.5.4 Use of the readline Library

For systems, such as Linux and Mac OSX 10, the GNU *readline* library has been used in the VM INCHAR routine. This enables input line editing and input line history. This change does not require additional documentation since *readline* itself is documented in the system that provides it.

If your system does not provide *readline*, the feature may be turned off by removing the define of READLINE in mdefs.h and editing the makefile.

### 4.5.5 VM Update (January 2004)

A change was made to the VM stack handling and, as a result, a change was made to the header information in the PDUMP format.

The change to the VM is that the operand stack now expands dynamically if it reaches its size limit. Both the operand and the link stack are now obtained by use of malloc(). Operand stack expansion uses realloc().

In order to make PDUMP format work correctly with variable stack sizes, the stack sizes are included in the header and used to load the correct size. For this reason the PDUMP version, as written in the header, was incremented. Older format PDUMP files will be read using the assumed default stack size (100 words).