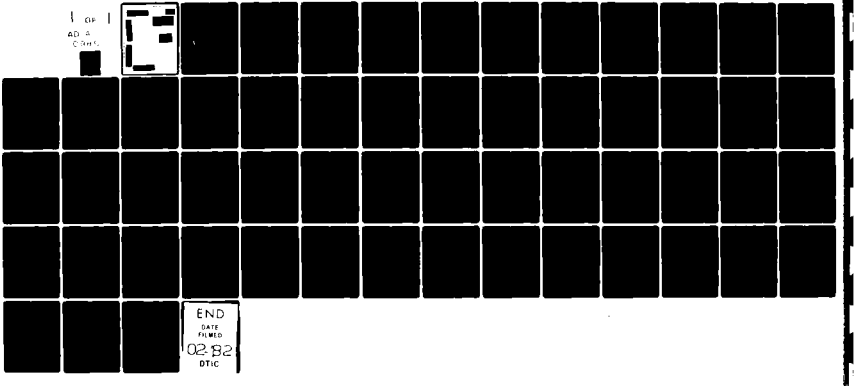
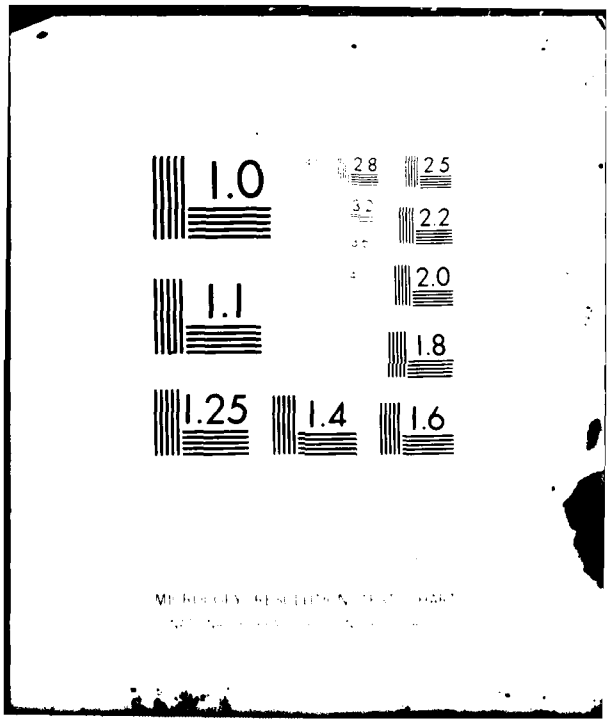
AD-A109 851 GENERAL SYSTEMS GROUP INC SALEM NH F/G 9/2
THE STUDY OF THE TRANSFER OF INTERRUPTED COMPUTATIONS BETWEEN C--ETC(U)
OCT 81 J TRIMBLE N00014-80-C-0228
UNCLASSIFIED NL



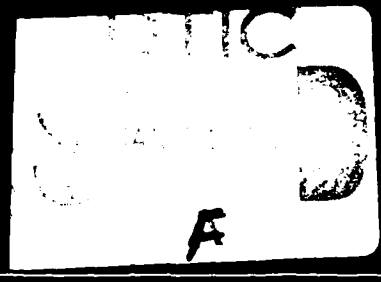
END
DATE FILMED
02-82
DTIC


1.0
2.8
2.5
3.2
2.2
2.0
1.1
1.8
1.25
1.4
1.6

General Systems Group, Inc.
51 Main Street
Salem, New Hampshire 03079

LEVEL

12



The Study of the Transfer of Interrupted Computations Between Computers

and

SAEC-0 User's Manual

4 --

General Systems Group Inc.

The Study of the Transfer of Interrupted Computations Between Computers

Final Report

October 30, 1981

Prepared for

Mr. Joel Trimble
Office of Naval Research
Department of the Navy
800 No. Quincy Street
Arlington, Virginia 22217

By

General Systems Group, Inc.

51 Main Street
Salem, NH 03079
Tel. (603 893-1000

1700 No. Moore Street
Suite 800
Rosslyn, VA 22209
Tel. (703) 524-7242

Contract no. N00014-80-C-0228

Table of Contents

1. Overview

We have reached the end of the contract; therefore it is time to review the goals of the project and summarize our accomplishments. This document provides such a review and summary. It goes into some detail on the accomplishments of the third phase; we provided similar details about the first and second phases in the final reports for those phases. We summarize the accomplishments of all three phases.

1.1. Project goals

The project began as an effort to study the transfer of interrupted computations between machines. Transferring an interrupted computation means moving a program and its execution environment from one machine to another, while the program is running and without having to restart the computation. There are both economic reasons and technical reasons for pursuing this goal. Suppose we have -- as we do -- implementations of a programming system on machines with different resources and different costs for their use. We want to use the cheapest machine with adequate resources for as long as possible. Only when we need a function or resource that is only available on another machine, do we want to migrate to the other machine and then perhaps only for the duration of our use of that function or resource. To take the system of interest, the programming system ECL, and the machines presently available, the PDP10 and PDP11/70, we can, for instance, be computing on the PDP11 and, at some point, require more space than it provides. Or we may want to use a package that is available only on the PDP10. We would then like to move the computation -- or some part of it -- to the PDP10, continue the computation there and possibly return to the PDP11, with the migration transparent to the user. Such a facility permits us to share resources in an economical fashion.

1.2. Project requirements

1.2.1. Dynamic transfer

The dynamic transfer goal imposes several requirements on a system's implementations. First, the translators for the programming language on the various machines must be similar. Second, we must be able to trace, from the point we want to transfer, the control state of a computation. We have to know all procedures, blocks, and statements in the executing program that we have entered and not exited and know the return points for each, and we must know where in the translator on the target machine to continue the computation. We

also must know, at the time we want to transfer, the location of the data that must be transferred and the data type of each piece of data. We need to know the location of the data objects to make sure we transfer the necessary data. We need to know the data types to map the objects. We have to know that some collection of bits is, say, an integer to map the bit pattern correctly. In our case, an integer on the PDP10 is a 36-bit object and on the PDP11 it is a two-word (each containing 16 bits) object. We can properly map the bits only if we know the data type of the stored objects.

The existing ECL implementations did not meet these requirements. The interpreter on the PDP10 is written in MACRO10, and the interpreter on the PDP11, in BLISS. In order to use the PDP10 implementation, we would have to write an interpreter for the 11/70 in machine code that parallels the 10 implementation, and we would have to construct a table that encodes corresponding points in the two interpreters. Though this approach is possible in theory, the resulting implementations would be difficult to maintain. We also could not use the implementation on the PDP11 as a starting point because its control state is embedded in the BLISS control stack. We doubt that many language implementations -- if any -- would meet the transfer requirements. Implementations of FORTRAN, for instance, do not include a record of the data type of an object, making it impossible to know whether some collection of bits is an integer or a real number. Most language translators tuck pieces of their control states into registers and other special places in memory. Any non-uniform treatment of control state information complicates the process of tracing and mapping the control state. We conclude that the transfer goal presents new requirements on a language implementation.

1.2.2. Portability

Since the ECL implementations did not meet the dynamic transfer requirements, we had to produce implementations that did. We also wanted eventually to have implementations on several different machines. We thus wanted to make it easy to produce new implementations, and we had to make sure the various implementations were consistent, each implementing the same language, within the differences caused by different word sizes on the various machines.

The ECL system consists of an interpreter -- which was hand-coded in the two implementations mentioned above -- a pretty printer, a backtrace facility and stepping function for debugging, a metering package, a compatible compiler for the PDP10, a

general-purpose hash package, several structure editors and a string editor, and a transformation facility for the creation and application of rewriting rules. Parts of the system – the pretty printer, the compatible compiler, and the transformation facility – are written in EL1, the programming language of the ECL system. We have or will reimplement the other packages in EL1. Coding the whole system in a higher level language makes it more portable. To produce an implementation of it on a new machine requires two things -- that we recode the machine-dependent parts of the system for the new machine and that we produce a translator for the implementation language that generates code for the new machine. The whole system can then be moved without further ado. We say more about isolating the machine-dependent parts of the system in section 2. We briefly discuss the implementation language and its translator in the paragraphs that follow. More is said about them in sections 3 and 4.

Recognizing that we would have to produce a translator for the implementation language for each target machine, we wanted to be able to produce them relatively easily and quickly. We defined a small but adequate implementation language – a subset of EL1 – and developed a retargetable translator -- a compiler -- for it. The subset of EL1 is called SPECL, an acronym for systems programming ECL. The criteria we used to define SPECL are (1) that the language not assume the availability of any runtime facilities, and (2) that it not assume the availability of a strong analyzer. The first criterion ruled out the use of pointers (since their use assumes the availability of a runtime facility, the garbage collector). Because the current SPECL compiler includes a simple analyzer, all free variables have to be top level variables, and we could not use the generic types ANY or ONEOF(...). The SPECL compiler generates stand-alone code and, because it does not attempt to be compatible with the interpreter, the code it generates is more efficient than the code that the compatible compiler generates.

We have designed the compiler to be retargetable. The machine-dependent part of the compiler is called the code generator; it generates machine instructions, in the form of text strings, that carry out the meaning of each expression in the compiled program. The compiler-writer encodes the strings that the code generator emits in what we call *formats*. We invented a simple language for the compiler-writer to use to define his formats. This language plays somewhat the same role that BNF plays for a language designer defining his language. To carry the analogy one step further, the code generator is like a table-driver parser, with the formats corresponding to the parse tables. We designed the format language

to cover the requirements of conventional machine architectures, using the PDP10 and VAX as guides for the initial design. After we looked at the M68000, the Apollo processor, we had to modify the language and processor somewhat; the changes were extensive but trivial, and we say more about them in section 3. As we move the SPECL compiler to new machines, we may have to make other modifications in the format package, but the scheme seems, on the whole, to be general and easy to use.

Coding the ECL system in a higher level language and developing a means for easily retargeting the compiler of that language facilitate moving ECL to diverse machines. We have not yet described how we assure the consistency of the implementations. This is achieved by using a *model* from which we derive the implementations. A model is a program that encodes, in this case, the operational semantics of EL1. We derive the production implementations by transformational refinement of the model and little else. The highest level of the model reveals the parts of each EL1 construct that are evaluated. The next level introduces the data stacks, which contain the name, type, and value of each variable that is visible in a scope. (We make sure that all manipulations on the data state meet the requirements for dynamic transfer.) Subsequent levels specialize the interpreter -- to a particular target machine and to a particular type of interpreter, value versus symbolic. Each level is produced by applying a collection of transformations to the previous level, starting with the model. The transformations may encode choices of data representation, provide definitions to procedure names, or be rewriting rules that describe the meanings of abstract notation. At each level of refinement, we have a program, and each program is derived from the model to ensure we maintain a degree of commonality. The various implementations comprise a program family, each of whose members eventually becomes an EL1 interpreter. This aspect of our approach -- the derivation of production software tailored to different machines by refinement of a model -- seems new.

1.3. Use of a prototype

We want to make one other comment about our approach, and that is that we found it useful to implement a prototype. The prototype differs from a production version in three ways: it does not model all of EL1; it uses simple representations for its data structures and simple algorithms, without regard for their efficiency; and it relies upon the runtime facilities of an interpreter-based programming system. Taking the first point, the prototype, for instance, does not model pointers. A pointer is a machine address, which we represent

by a bit vector in the production versions. The contents of the bit vector addresses a place in memory, which we represent by a vector of words. A word is also a bit vector. (Its length and the length of an address are machine-dependent.) Since it is difficult to debug code whose principal data structure is a collection of bit vectors, we wanted to exclude from the prototype those language features that are modelled at that level. The second point, about efficiency, is probably true of most prototypes. In some cases, it is difficult to know what is costly until you get something working that you can monitor. Also, as a general rule, we derive a prototype by deferring considerations of efficiency. The third point is a comment about the benefits of using an interpreter-based language with good debugging facilities for prototyping.

We did not abandon the prototype nor turn to a different programming language to derive a production version. Rather, we derive the production version by a gradual evolution of the prototype, casting out non-SPECL constructs and gradually expanding the model. (We have not begun to meter or tune the implementation.) The prototype allowed us to debug many components of the interpreter, to decide which concepts should be explicit at the highest level and which deferred, to ensure the implementation would meet the transfer requirements, and to ensure the model would support implementations on diverse machines.

2. The model

2.1. Choosing an implementation language

We started the project assuming we would write the model in EL1 and write a translator from EL1 to BLISS. We already had one implementation of EL1 in BLISS (on the PDP11/70), and BLISS was available on the other machine of interest at the time, the PDP10. After discussing the use of BLISS with the implementors of EL1 on the 11 and thinking about the design of a translator from EL1 to BLISS, we concluded that BLISS was not an adequate implementation language. The differences between EL1 and BLISS make writing a translator difficult. We would also be unable to provide implementations of EL1 for machines other than those for which BLISS generates code. Third, it was unclear whether we could, going that route, meet the dynamic transfer requirements.

We thus chose to use a suitably-constrained version of EL1. Since the interpreter had to be reasonably efficient, we had to choose a subset that did not require runtime facilities.

We thus excluded pointers. Because of this and because stacks and stack operations are not part of the subset, the interpreter creates a representation of the several stacks it needs and does its own retrieval of reusable space. Because the subset does not let us address memory directly, the interpreter constructs a representation of memory and keeps track of the available space in it. The interpreter represents memory by a large vector of words and divides the vector into segments in which it places the stacks and the heap, the storage area for non-stacked objects.

We also wanted to be able to produce compilers for the subset relatively easily. This ruled out any dependence upon a strong analysis component. The compiler has to determine, by the time it is ready to generate code and without much analysis, the data type of each expression in the program it is compiling. This ruled out the use of free variables and the generic types ANY and ONEOF(...).

2.2. Choosing target machines

Another change that occurred during the course of the contract was in the choice of target machines. We started with the PDP10 and PDP11/70 in mind. When the VAX was announced, we assumed it would replace the PDP11/70 as a target. Then large personal computers like the Apollo came on the market. They are good hosts for programming systems like ECL (and programming environments like the PDS, which is built on ECL). We thus have also kept the M68000, the Apollo processor, in mind as a target machine.

2.3. Isolating machine-dependencies

We have attempted to isolate the machine-dependent parts of the new implementations. The routines that decide how to pack the components of a compound object are sensitive to word and byte boundaries and the target machine's instructions for manipulating sub-word objects. We are likely to have to change them as we move to new machines. On the PDP10, for instance, we place the components of an array of characters, each of which is seven bits long, one right next to the other until we run out of space in a word. But because the VAX machine code allows us to address objects at byte boundaries, where a byte is 8 bits long, we would place each character at a byte boundary. On the other hand, we always place an address at a half-word or full-word boundary on the 10. Such differences affect the data packing algorithm, which is encoded in a few routines of the model. Other machine-dependencies, such as the size of a word or byte and the size of an integer or

pointer, are encoded in global variables. Routines uniformly use the global variables rather than the numbers.

2.4. Status

The diagram in Figure 1 illustrates the status of the components of the interpreter. Asterisks indicate the refinement is being tested, and vertical dashes indicate the refinement has been tested.

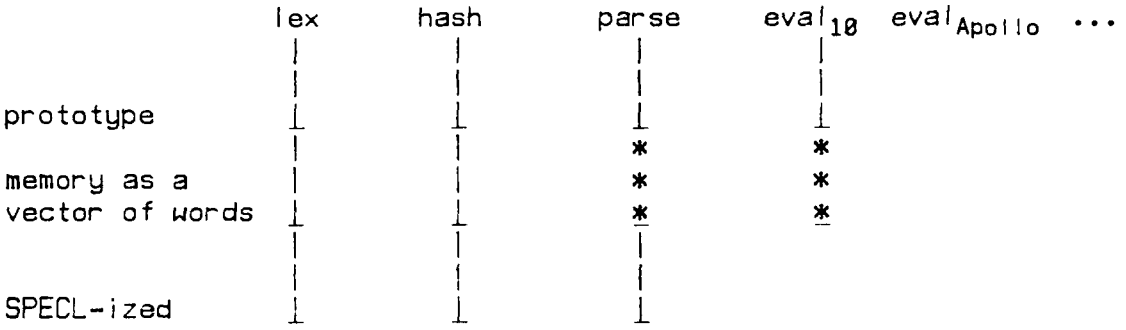


Figure 1. Status of the interpreter family

During the third year, we worked on the parts of the system that operate directly on the representation of memory as a vector of words. We modelled physical generation, which is the component that lays out objects from the user's program in memory; parts of memory management; the type generators; other type-related routines, including MD, VAL, and LENGTH; the data packing algorithms; the value stack, which is part of the data state; and initialization. The initialization phase lays out in memory all of the builtin objects, including the builtin data types, the builtin SYMBOLs and ATOMs, and the builtin numbers. We used physical generation as a model for initialization. We thus have two refinements of the model of physical generation -- one that the interpreter asks to lay out objects from the program being interpreted and one that we use to lay out the builtin objects during initialization. We thus assure that objects are laid out uniformly. We have debugged initialization and used it to debug the lexical analyzer. We will debug the other refinement

after we get the parser working and get it handing FORMs to the evaluator.

The source programs comprising the interpreter model and its refinements are available as standard DEC System-10 files on the PDP10 at Harvard University.

3. The SPECL compiler

Over the second and third phases of this effort, we developed a compiler for what is called "Systems Programming" ECL. This is an ECL subset that does not require runtime facilities, such as a garbage collecter or resident mode descriptions (necessary to handle indeterminate modes). The subset is the target for the final refinement(s) of the EL1 interpreter; having a retargetable code generator for the SPECL compiler means that we will have a portable ECL.

3.1. Retargeting to Apollo

An early version of SPECL (which produced code for the DEC PDP10) was functional at the end of the second phase. During the third phase, there were several extensions to the compiler. The most important of these is the addition of a code generator for a second machine, the Apollo. We discuss this in some detail, because retargeting is crucial to porting the interpreter and because the relative ease with which this was done demonstrates the feasability of the approach.

3.1.1. FORMATs

When the SPECL compiler was first designed, Apollo did not even exist. However, the architecture of the SPECL compiler was set up so that those aspects that "know" about EL1 are embodied in the program of the code generator, while almost all aspects that "know" about a particular machine are embodied in FORMATs. FORMATs have two roles. First, they supply the *text* that appears in the output file of the code generator (the output is machine language acceptable to the assembler of the target machine). The only text appearing in the output file not coming from a FORMAT is either the digits of an integer or the characters in a variable name. Second, FORMATs provide a mechanism for various types of case analyses, since the order in which cases are considered often varies from machine to machine. In spite of the ability to do case analysis, FORMATs are in no way procedural. They do not allow looping or recursion. The rationale is that anything this complicated should be in the code generator -- the rules for constructing FORMATs must be

very simple, easy to understand and learn.

To better understand how the SPECL compiler can produce code for both the PDP10 and for the Apollo, we consider a simple example. Suppose that the code generator wants to produce code to move the contents of one variable into another variable. Suppose that the data structures for the object program variables are pointed to by the code generator's variables v1 and v2, e.g. v1 points to a data structure for the variable that will be known as A in the object program, while v2 points to a data structure for B. In order to produce the desired code, the following statement might appear in the code generator.

```
GenCodeFormat(FormatCopyWord, GenCodeVariable(v1),
                              GenCodeVariable(v2));
```

The variable FormatCopyWord contains the FORMAT; it has a different value depending upon the target machine. In both cases, the FORMAT will be a pointer to a string. The strings for the two cases are:

```
PDP10:   'MOVE  $T,*
          MOVEM $T,**
         '

Apollo:  '      move.w  *,**
         '
```

We now consider the action of GenCodeFormat when it is given one of these strings as an argument. It begins by copying the characters of the FORMAT into the output, up to an *. When it encounters an *, it counts the number of *'s up to a non-*. This is the *hole-number*. In both the above cases, the first hole encountered is hole number 1, but, in general, holes may be encountered in any order – even in different orders on different machines -- and may occur any number of times (including none). When hole number n is encountered, GenCodeFormat evaluates its $n+1^{st}$ argument (arguments except the first are taken as unevaluated). It is the responsibility of the $n+1^{st}$ argument to "fill" the n^{th} hole; this argument is called the n^{th} filler. In this example, what must happen is that GenCodeVariable(v1) must place A in the output file. After the evaluation of GenCodeFormat's argument, copying the FORMAT string into the output file resumes after the hole, i.e. with a carriage-return line-feed for the DEC-10, or a comma on the Apollo. This continues up to the next hole, which is hole number 2 in both examples. Then GenCodeVariable(v2) places B in the output after which FORMAT processing finishes up with a carriage-return line-feed.

The above example illustrates how FORMATs supply text for the output file, but it does

not exemplify any case analysis. To see how this occurs, we look at an extension of the example. Consider that in reality, the variables might be imported. Assembly language syntax for the PDP10 uses the notation A## to mean that the symbol A is imported. With this in mind, a possible implementation of GenCodeVariable is

```
GenCodeSwitch(SwitchImportedVariable, <IsImported(vs1)>,
                                      GenCodeName(v1.name));
```

A *switch* is a FORMAT that can do case analysis on one or more flags. We use angle bracelets to enclose a list of mask-format pairs:

```
< 'T'   '*##'
  'F'   '*'>
```

In other words, if the flag is true, use '*##' as the FORMAT string; if false, use '*' -- nothing but the hole itself. The n^{th} filler for GenCodeSwitch is its $n+2^{nd}$ argument.

As the code generator stood at the end of phase two, this was how imported symbols were handled. All the machine languages with which we were then familiar had the property that instruction sequences did not depend upon whether a symbol was imported, and that a variable could be flagged as imported by some syntax local to the name itself.

The Apollo changed all that. It is a peculiarity of its operating system that a *different instruction sequence* is necessary when a symbol is imported, and variables cannot be flagged as imported using local syntax. Thus, the code generator had to be changed for the Apollo, as well as the FORMATs. *However*, because we adjusted the old FORMATs for the PDP10 to conform with the modified code generator, we still have one code generator for the two different target machines. We make no particular apology for not having anticipated Apollo's peculiarity. Machine and operating systems are sufficiently idiosyncratic that it is probably impossible, and certainly not cost-effective, to anticipate all variation. What we demonstrated is that even when the completely unexpected arises, the framework is robust enought to accomodate it. The changes to the code generator and PDP10 FORMATs were many, but they were trivial -- there was no reworking of the design, and no deep thought went into it.

Because the issue of imported symbols on the two machines illustrates some undiscussed FORMAT constructs, and because it shows how the modularity of one machine can be exploited even in the absence of it on another, we pursue the "fix" that is necessary to handle imported symbols. Since importedness can cause changes in the instruction sequence, the flags must be fed in at the highest level. Thus the code generator statement to copy a

word must read:

```
GenCodeSwitch(SwitchCopyWord,<IsImported(v1),IsImported(v2)>
                GenCodeVariable(v1), GenCodeVariable(v2));
```

The definition of GenCodeVariable is changed so that it outputs just the name, regardless of whether the name is imported. Note that SwitchCopyWord is fed two flags, not just one.

On the PDP10, what would we like SwitchCopyWord to do? Basically, we would like to use the simple notion of FormatCopyWord, but we want the holes filled differently. In place of hole 1, we do not want filler 1 directly, but rather the result of using filler 1 in SwitchImportedVariable, where its flag is the *first* flag of SwitchCopyWord. Likewise, in place of hole 2, we want filler 2 and the *second* flag of SwitchCopyWord to be fed to SwitchImportedVariable. Before explaining how to pass along fillers and flags, we give the definition of SwitchCopyWord.

```
(FormatCopyWord SwitchImportedVariable
                (<<SwitchImportedVariable '2'>> '**'))
```

To pass along fillers, a parenthesis notation is used; the construct is called a *substitution*. The format whose holes are to be filled -- in this case FormatCopyWord is the first element in parentheses, followed by the fillers, in order. The rule for a substitution is that the n^{th} hole of the first format is filled by the $n+1^{st}$ format, where *its* holes are filled by the original fillers for the substitution. Flags are just passed along. Thus, the first hole of SwitchCopyWord is filled by SwitchImportedVariable, where its hole is filled by GenCodeVariable(v1), and its flag is given by IsImported(v1). The effect is the same as the original use of FormatCopyWord.

Now consider the filler for the second hole of SwitchCopyWord. Begin at the innermost level; the << >>'s are used to change flags around. This construct is called a *toggle*, because it may change the settings of flags for a switch. It consists of a FORMAT and a *setting*, where the setting is just a string, usually of digits. The rule is that if the digit i occurs at the n^{th} place in the setting, then the n^{th} flag on the inside corresponds to the i^{th} flag on the outside. In our case, the first flag of SwitchImportedVariable (it has only one) corresponds to the second flag at SwitchCoyWord, i.e. to IsImported(v2). Thus, <<SwitchImportedVariable '2'>> is equivalent to:

```
<  'xT'  '*##'
   'xF'   '*'>
```

(The "x" means ignore the flag in this position.) Thus, a toggle may be viewed as a

substitution for flags (rather than holes).

Recall that the toggle we just discussed is the first item in a substitution. This entire construct may be viewed as equivalent to the following:

```
<  'xT'   '**##'
   'xF'   '**'>
```

When this is used as the second filler for FormatCopyWord, the effect is again the same as the original use of FormatCopyWord and the original call on GenCodeVariable(v2).

In summary, then, even when the PDP10 FORMATs are made compatible with those of the Apollo, it is still possible to modularize the notion of copying (in FormatCopyWord) with the notion of being imported (SwitchImportedVariable). The FORMAT SwitchCopyWord combines the two concepts for the desired effect. On the Apollo, SwitchCopyWord must begin in an entirely different way. It first figures out which instruction sequence it wants, for example by:

```
<  'FF'   FormatCopyWord
   'FT'   FormatCopyWordToImported
   'TF'   FormatCopyWordFromImported
   'FF'   FormatCopyWordFromImportedToImported>
```

We will not explore this in further detail. The point is that it goes about the case analysis in an entirely different way from the corresponding FORMAT for the PDP10, yet both are used by exactly the same call in the code generator itself.

3.1.2. Resources used

When the SPECL compiler was designed, it was hoped the use of FORMATs would make retargeting not only possible, but also relatively inexpensive. Thus a report on the effort required to do the FORMATs for the Apollo is in order. The project was brought to its present state (written and partially debugged) by an inexperienced undergraduate in two and one-half real time months. This person was unfamiliar with the Apollo system and with the underlying hardware (the Motorola 68000), and of course knew nothing of FORMATs. His knowledge of EL1 was based on a single-semester course. In the debugging phase on the Apollo, he had to contend with a somewhat shaky operating system and a file transfer protocol involving floppy disks on the subway. In view of the two and one-half months spent on this project, and all the impediments, it seems reasonable to expect that three to four months will suffice to produce a debugged code generator for a new target machine.

In addition to the time spent on the FORMATs themselves, there was also some

investment of effort by the designer and implementer of the original code generator. Two to three man-weeks (spent over a two month period) were spent in familiarizing the person doing the FORMATs with the requirements that they had to meet. Another two to three man-weeks were required to modify the code generator and the PDP10 FORMATs because of the aforementioned peculiarity with imported symbols in Apollo assembly language. One to two man weeks were required for the few target-dependent routines of the code generator (described in the next paragraph). In future efforts, the two to three week training effort by a person familiar with the code generator is likely to remain a constant, for each new *person* who retargets the code generator (but not for each retargeting). The time spent because of the imported symbol peculiarity of course will not recur, and we believe that modifications of this scope are more unlikely than ever to arise. But as we said earlier, the possibility cannot be ruled out entirely, and it is not until we understand a new target in detail that we know exactly what the situation is. From this experience, however, we have a rough estimate of how bad things can be. Finally, the one to two weeks required for the target-dependent routines will likely be reduced to a matter of days on future retargetings. In summary, once a person has retargeted the compiler for a new machine, the effort in addition to doing just the FORMATs is likely to be negligible -- in the worst case a few man-weeks. Further, the training of a new person also requires a few man-weeks of time by someone who has done the same thing before. Thus, the few man-months required to do the FORMATs for the new machine is the bulk of the total effort.

This optimistic report must be balanced by several observations. First, the only I/O that SPECL supports is characters to and from the user console. Once more I/O is developed, it will take us somewhat longer to retarget, to cover the additional capability. Second, if the compiler becomes capable of doing more optimization, the retargeting time will probably increase because many optimizations are machine-dependent. We add to these points the note that the generated code is *text*, which must be transferred to the target machine and assembled. If the turn-around time for this process is long, the shake-down period is unavoidably extended. Finally, none of these estimates includes the effort required to modify machine-dependent aspects of the ECL model.

3.1.3. Target-dependent routines

We now discuss those aspects of retargeting that cannot be captured with FORMATs. They are all concerned with how data objects are laid out – how many bits in a word, how STRUCTs are packed, how sub-word arrays are done, etc. In the original PDP10 implementation, the routines that address these issues were implemented by peering into the data description block for the mode. For example, to find out how many words are in a given mode on the PDP10, it is possible to use the size field in that mode's data description block. When we are cross-compiling, such a technique is impossible, and instead it is necessary to

> Specify (e.g. in a table) how much space is used by BASIC modes.
>
> Compute the size of a mode built up out of other modes, given the sizes of the constituent modes.

The implementation of this and a few related routines is what took the one to two man-weeks for the "target-dependent" routines of the code generator. The reason that the time will be reduced in the future is that now the routines for doing these tasks do not "cheat" by looking at data already set up, but compute it from scratch. They are parameterized by word size (an easily changed constant) and will not vary much from machine to machine.

3.2. The constant compiler

The role of the constant compiler is to set up the initial data used by a program. For example, any time a string such as 'ABC' appears in a source program, the bits for that string must occur somewhere in the object code of the program. This representation must be consistent with that of other data objects, and so is a target-dependent problem.

The constant compiler delivered at the end of phase two was specific to the PDP10 and was not set up to adapt readily to other machines, such as the Apollo. Rather than writing a new constant compiler for the Apollo, and facing the same problem anew for the next machine, the old constant compiler was discarded and replaced by one that is target-independent, in the sense that the dependence lies in the use of FORMATs (very few and very simple) and in the use of target-dependent routines described above. A particularly nice consequence of this approach is that there is guaranteed compatibility between the constant compiler and the code generator -- if we decide to change the layout strategy, only the target-dependent routines are affected. The output of the constant compiler and of the

code generator may change, but the changes will be consistent.

3.3. Compiler enhancements

The compiler delivered at the end of phase two did not allow mode-behavior functions nor length-unresolved objects; both features have been added. We only had to modify the graph-maker, the front end of the compiler, to incorporate mode-behavior functions. We modified both the graph-maker and the code generator to handle length-unresolved objects and produced additional FORMATs for the PDP10. This was completed before the Apollo effort began, so that effort included length-unresolved FORMATs as a matter of course. The length-unresolved machinery was tested thoroughly on the PDP10, by a complicated example having sequences of sequences and STRUCTs with multiple length-unresolved components. The Apollo implementation has not yet been fully tested.

4. Weak interpreters for ECL

4.1. Overview

4.1.1. Comparison with other analysis tools

A weak interpreter for ECL is an analysis tool. Tools of this general class are characterized by the fact that they gather information *uniformly* over the whole program and that the information they gather is typically *shallow* -- i.e. does not involve any intellectually difficult proofs -- and useful principally in compilation and optimization. This type of analysis is different from *symbolic evaluation*, which looks very hard at certain parts of the program and is capable of obtaining "deeper" information than a weak interpreter. The results of symbolic evaluation are typically used in understanding program behavior, e.g. proving program correctness. There is of course some overlap. Results of weak interpretation are sometimes helpful in proving program correctness -- especially in showing where the program is not correct -- and the output of symbolic evaluation would be useful to a compiler.

4.1.2. Parts of a weak interpreter

A weak interpreter for ECL consists of two parts, the control part and a property set. The control part is common to all weak interpreters, and is often called *the* Weak Interpreter. This module "knows" about the control structures of ECL, but it is helpless without a property set. A property set is constructed to answer questions of interest to the user (probably a compiler or optimizer). It also supplies routines to the Weak Interpreter.

The Weak Interpreter and a property set coordinate each other's activities. One can imagine changing the syntax and semantics of ECL in a number of ways. Assuming there is a set of weak interpreters for the original ECL, a corresponding set for the changed ECL could be obtained by changing only the control part. A more likely scenario is that we would like a new weak interpreter to gather different information. In order to do this, it is necessary to construct only a new property set. As we shall see, property sets themselves are broken into several modules. When the type of information is changed only slightly, it is often possible to reuse modules.

4.2. The control component

4.2.1. Arguments and results

The routine `InterpretProgram` is the interface to the Weak Interpreter. Its single argument is a symbol, whose top level binding is the EXPR at which analysis is to begin. While property sets can be constructed to limit the analysis to the given EXPR, the Weak Interpreter is set up so that inter-procedural analysis is the more natural thing to do. The "result" of `InterpretProgram` is a database of information that has been constructed by the property set. This will typically include information about the EXPRs that were analyzed, as well as data on the top level variables used by the program.

4.2.2. Stabilization

A key characteristic of weak interpretation is that there may be any number of passes over the program or, more precisely, a varying number of passes over different parts of the program. A property set may be defined so that only one pass is necessary, but such property sets are usually not powerful enough to gather useful information. The theoretical basis of weak interpretation is described in several articles [7, 10, 12]. The method rests on the notion that the information gathered on early passes may be too strong and, therefore, incorrect; but that eventually the information becomes self-consistent, and, at that point, the

theory guarantees that it is correct. Suppose a program includes the assignment X ← 1, followed by a loop containing the assignment X ← X + 1, and that those are the only places X is modified. After encountering only the first assignment, a property set could detect that X is the constant 1, that is correct up to that point in the analysis but is an incorrect assumption about X in general. After encountering the second assignment and perhaps after the Weak Interpreter makes additional passes over the loop, a property set could realize that X is a positive integer and will remain so in all possible computations.

4.2.3. Internal structure

This section contains a brief discussion of each of the modules of the Weak Interpreter. The first of these is the *Program* module. This module handles all of the function-call and function-return semantics -- those parts of the language by that a set of EXPRs becomes a program. This module implements the "stabilization" test and thus controls the number of passes the Weak Interpreter makes.

The *Mark-Return* module has the duties that its name implies. It works closely with the part of the property set known as the Mark Stack.

The *Statement* module handles the constructs that introduce statement-level environments, i.e. BEGIN and FOR, and the constructs that must occur at statement level, namely, exit conditionals and DECL.

The *Construct* module handles CONST and CONSTRUCT. It ensures that the user mode-behavior functions for generation and conversion are "called", i.e. Weakly Interpreted, in the proper way. This module works closely with the *System Generate* module, which recursively uses the routines in the *Construct* module to evaluate components, SIZE parameters, etc.

The *Branch* module has routines for analyzing the various conditional execution operations of ECL. These include CASE, simple conditionals, utilities for the exit conditionals, and the n-ary operators AND and OR. All these constructs are reduced to a uniform notion of a two-way branch, so that a property set is unaware of the variations in the surface syntax.

The *Select* and *Assign* modules have the obvious role. Remember that in each case, they must worry about "calling" any applicable user mode-behavior functions. The *Mode*

Constructor module handles PTR, ONEOF, ::, etc. The *Subr* module is responsible for the remaining builtins, all of which take evaluated arguments. The many possibilities are reduced to a small number of constructs which the property set must worry about, principally unary, binary and ternary operations.

4.3. A property set for SPECL

The first property set to be paired with the Weak Interpreter will be an acceptor (or validator) of SPECL (recall that SPECL is a subset of the full language). This weak interpreter also does some constant-folding and gathers other information of use to the SPECL compiler.

In the sections that follow, we review the modules that make up this property set. It will become clear as we proceed that some of these modules would carry over unchanged to other property sets, but that others would not. It is often the case that a specific restriction in SPECL is implemented in a very localized place in one of these modules. The order of presentation here is mostly "bottom-up", so that when the data structures for a module are defined, their constituents have already been discussed.

4.3.1. Pure value set

A *pure value set* specifies the set of values that a location can have. This property set allows the following possibilities:

- Whether there is only one possible value (*i.e. a constant*), and if so, what that value is
- For all but routine values, the mode of the value
- For routine values, the set of routines that are possible.

Thus, a pure value set always specifies a known fixed mode (no generics), and for non-routines the only extra information it keeps is whether the value is a constant. Since the set of routines in any program is finite, a pure value set for a routine can contain all the possible values of the location.

4.3.2. Top level variables

This module keeps track of which SYMBOLs are used as top level variables, it detects sharing of the top level bindings of these variables, and, with each binding, it associates a pure value set (see above), that specifies the set of values which the binding can have.

4.3.3. Location value set

In ECL, each expression returns a locative result. Sometimes a special place, not pointed to by anywhere else, has been reserved to hold the value pointed to by the result. This is called a *pure value* and motivates the term *pure value set* that was used above. Often results point to places that are shared; the possible sharing patterns are what is described by a *location value set*. Corresponding to the elementary operations by which sharing may arise, there are the following possibilities for location value sets:

- A top level variable
- A stack variable
- A subscripted location value set (note recursion), subscripted either by a constant or by an arbitrary index
- A finite union of location values sets (note recursion).

4.3.4. Stack variable

A location introduced by a DECL may be a pure value or a shared value or both. A *stack variable* is a pair consisting of a pure value set and a location value set, with the understanding that either set may be null (implemented by NIL), in addition to the values already described for each. A variable that always has a pure value thus has a null location value set, and one that is always shared has a null pure value set. However, some variables may have both components non-null, as in `DECL A:INT LIKE ()P => B; 0().`

4.3.5. Value stack

A *value stack* is literally a stack of *stack variables* as defined in the previous section. It is used in this property set to describe "what is known" about local variables of an EXPR at any point in the process of weak interpretation.

4.3.6. Predicates

A *predicate* consists of a structure that records what is known about top level variables (see section 4.3.2), and what is known about the stack (see the previous section). The nature of the predicates affects the power of a property set. For this property set, the predicates say nothing about non-local variables that are not top level. Since SPECL does not allow the use of such variables, this omission is allowable, but other property sets might choose something different. Also, there is no way, given the structures we have discussed, to relate the values of two variables (i.e. I=J or A GE B). Other property sets might do this, but, for this property set, we want something simple and relatively fast.

4.3.7. Free variables

Free variables are recorded on a per-EXPR basis. When a function is called inside another, we check whether the callee's free variables are in the caller's list of local variables. If so, there is a violation of SPECL's requirement that non-local variables be top-level. If not, the free variables of the callee are added to those of the caller. When weak interpretation is complete, any violations present will have been caught. This module is responsible for the operations on a free variable set.

4.3.8. Name stack

This module maintains the stack of local names. It is probably the least interesting module in the entire property set. There would be more to discuss if the property set allowed the use of non-local, non-top level variables.

4.3.9. Mark stack

A mark stack consists of a list of mark stack entries, each of which contains the name of the mark (a SYMBOL), a pointer to the point on the name stack where the mark was established, and a predicate (see section 4.3.6) that describes the state of the computation upon return to the mark. (A NIL predicate means that no return to the mark has been encountered.)

In SPECL, only returns to local marks are allowed. This module will discover any illegal returns.

4.3.10. Expr predicates

The *Expr Predicate* data structure is what the Weak Interpreter attaches to each EXPR. The collection of these predicates constitutes a part of the result of weak interpretation. The pieces of an Expr Predicate that are most useful in this regard are the Predicate (see section 4.3.6), the Free Variable information (see section 4.3.7), and the IsRecursive and UsedLocatively flags. There are other fields that are used for internal purposes.

4.4. Status

The entire weak interpreter has been written, entered, and unparsed. There is substantial documentation on some of the more delicate facets of its implementation. The property set for SPECL is very nearly complete: six of the ten modules are ready to be debugged; the remaining four -- Stack Variables, Top Level Variables, Pure and Location Value Sets -- are among the simpler modules.

5. The program development system

5.1. Overview

During the last three years, we have added several new tools and improved several existing tools in the Harvard Program Development System (the PDS), and we have used the PDS to develop and maintain several large programs.

The PDS supports a programming methodology, which we call transformational refinement [3]. When using this approach to solve a problem, a programmer starts the development of a program by producing an abstract model. The model, a program also, is a procedural specification of a solution to the problem. The programmer typically creates new nomenclature and notation, corresponding to concepts in the problem domain, to use in coding the model. The model should be easy to read and precise, yet not overly constrain implementation decisions. Oftentimes the model is a forebear of several implementations, thereby spawning a program family. The model may or may not be directly executable; it can be parsed and unparsed. The programmer produces an executable version by providing definitions of abstract constructs. The definitions need not be in terms of the concrete language; more often they also involve some abstract constructs. The refinement of the model into one or more executable programs is a gradual process. A refinement might include a definition of a procedure or data type or be a transformation rule for rewriting a

construct. Because the use of transformation rules has a significant impact on the design of a programming environment, we use the phrase transformational refinement in describing the PDS.

We also mentioned that the PDS supports the development of program families. Many of our projects result in program families where one member of the family is a prototype, viz. the EL1 interpreter. Production-quality versions are derived by developing different refinements that result in more efficient code.

During the past year, we worked on several tools of the PDS -- in some cases, making them more efficient and, in other cases, expanding their capabilities. We also developed some new programs using the PDS, to gain more experience with it.

5.2. Enhancement of PDS tool set

5.2.1. The analysis tool

The PDS analysis tool (named FUI) looks for undefined and/or multiply defined constructs and prepares a cross-reference listing. We have found this tool helpful in the early stages of debugging; however, the original FUI was limited to dealing with concrete program modules. During the past year we re-developed the tool and extended it[1] to deal with abstract program modules as well as with concrete modules.

In order to analyze abstract program constructs, the user supplies *semantic analogies*. A semantic analogy "explains" an abstract construct in terms that are sufficient for FUI to do its job but without necessarily defining the construct completely. A complete description of the new FUI will be available in the updated PDS User Manual.

5.2.2. The Package tool

The Package tool produces a representatiaon of the entities in a collection of concrete program modules that is ready (1) for loading into an interpretive ECL environment and (2) for compiling by the EL1 Compatible Compiler. One of the major functions of the Package tool is to order the entities so that quantities are defined before they are used. In addition, it detects recursive mode definitions and gathers a set of mutually recursive modes into a

[1]Actually, there are two distinct tools -- one dealing with abstract modules and the other dealing with concrete modules. They are both derived from a single abstract analysis model.

single group for processing by the cyclic mode package, developed during phase one of the project.

The original Package tool was put together rather hastily and was based on a simple (and rather naive) analysis routine. The Package tool has been redone. The new version is better structured and uses the new concrete analysis tool (FUI). Further, it produces a file that can be loaded into an ECL environment or be utilized by the Compile tool (see section 5.2.4 below).

5.2.3. Rewrite package

During the past year we developed a new rewrite package that will eventually replace the original one (which was written in the early 1970s). It uses a discrimination net to represent a set of applicable rewrites, which speeds up the matching of a rewrite pattern, particularly when there are many rewrites with similar patterns. We have changed the transform and concretize tools to use the new package.

5.2.4. Compiler interface

We had hoped to incorporate the EL1 Compatible Compiler as a tool in PDS. However, because the combined size of PDS and the compiler requires more memory than is available on the PDP10, we are unable to do this. What we therefore had to do was provide a stand-alone tool that accepts a package module containing a collection of concrete entities plus instructions to the compiler and produces a binary file. The user calls upon the tool with the

Compile(P)

where P names the package module to be compiled; the other information needed by the compiler is contained in "instructions" inserted into the package module by the Package tool.

5.3. Experimental use of the PDS

The PDS has been used on a variety of projects, including those discussed in the paragraphs below.

1. The PDS is maintained within itself, and all the new tools were developed and are being maintained within PDS. With the exception of the tools re-developed this year (see section 5.2 above), the components of the PDS do not provide good examples of the transformational refinement methodology. Most have evolved from a succession of predecessors that were, in some cases, originally developed several years ago. The new tools do, however, provide good examples of the use

of the methodology.

2. Prototypes of two components of a new formula simplifier/theorem prover were developed using PDS and the transformational refinement methodology. The prototypes were developed rapidly, with little concern for their efficiency, to provide a basis for experimental use.

3. During the spring of 1981 a number of groups of students (seven groups of one to three students each) used the PDS and the transformational refinement methodology to develop a collection of term projects. One of these, providing database management facilities, is being further developed and documented.

The projects cited above reenforce our belief in the viability of the transformational refinement methodology and the utility of PDS in supporting the development and maintenance of individual programs and families of programs. However, there are a number of problems that have been revealed or confirmed by the projects. Those divide into problems with PDS itself and problems related to the methodology; we discuss these below.

5.3.1. PDS problems

The PDS is somewhat slow at times. In a number of cases this is due to the fact that the implementations chosen are not as efficient as they could be; re-programming parts of various tools and certain PDS utilities would alleviate these problems. A more serious problem is that there are certain operations that are inherently expensive (both in terms of time required and space required) and, so long as one is using PDS as one of several users in a time-shared machine like the PDP10, these will constitute a problem. The solution here is to move the system to a high performance personal computer network like the Apollo Domain.

There are circumstances in which the PDS is not very robust and/or is awkward to use. PDS is still an experimental system and therefore many of the components do not have a good human interface; the problems can all be alleviated by a careful re-engineering of the user interface and of a number of the utilities that do command decoding and error handling.

A final problem is that the user interface has been designed for low bandwidth terminals and therefore suffers the usual problems that that design decision implies. For future use it will be important to redesign the user interface to take advantage of a high bandwidth terminal with a high resolution graphic display.

5.3.2. Problems in using the transformational refinement methodology

The transformational refinement methodology as supported by PDS can have a dramatic impact on the cost of developing and maintaining application programs and on the reliability of the resulting products. However, it is clear that training programmers to use the methodology and the system is non-trivial. There are reference manuals both EL1 and for the PDS, but these are helpful only when the user knows how to develop abstract models and the transformational refinements required to derive concrete programs. What is now needed is a tutorial on the EL1 language, plus a collection of appropriate, well-documented examples of transformational refinement.

6. Conclusion

We are pleased with our progress during the course of this contract. Many of the issues that seemed complicated at the outset now seem easy to understand and tractable. The choice of the implementation language for the EL1 model seems good. We see how to refine the model to that language, and we know that it is relatively easy to implement the language on diverse machines. The model is well along, and we are confident that it will meet the transfer requirements. We cannot say how efficient the system will be, but that we can investigate once we complete the refinement of the model and compile it. It is clear what should be done next in pursuit of the project's goals.

In terms of the model, we urge that the model be fully refined. This includes modelling the remaining parts of ECL and confining the constructs we use to only those that the SPECL compiler accepts. A second implementation of the model, for a different machine, should be produced, and the implementations have to be compiled, metered, and tuned. We recommend that materials -- such as tutorials -- be produced to help new users learn the language and its accompanying packages.

The present compiler is nearly complete; we expect that it would require a modest effort to debug the code generator for the Apollo. The weak interpreter should be completed and debugged, and we could use it in the place of the graph-maker as phase one of the compiler. It would be appropriate to pursue the construction of additional property sets, to produce a more powerful weak interpreter for use by the compiler and other tools.

Once ECL is available on a personal computer, particularly one that has extensive screen handling features, it would be good to add a screen editor and network I/O. We would also

recommend that the PDS be moved to a personal computer.

We propose these recommendations based on the experience gained during the course of the present contract. Each is an extension of work already begun. We hope interesting problems lie within their pursuit; on the other hand, we feel that each is tractable and can enhance the state of our programming tools.

References

[1]
Cheatham, T. E., Jr., Holloway, G. H., Townley, J. A.
A system for program refinement.
In *Proc. 4th Int. Conf. on Software Engineering.* IEEE, Munich, 1979.

[2]
Cheatham, T. E., Jr., Holloway, G. H., Townley, J. A.
Symbolic evaluation and the analysis of programs.
IEEE Trans. Software Engineering SE-5(4):402-417, 1979.

[3]
Cheatham, T. E., Jr., Holloway, G. H., Townley, J. A.
Program refinement by transformation.
In *Proc. 5th Int. Conf. on Software Engineering.* IEEE, San Diego, 1981.

[4]
Cheatham, T. E., Jr.
Comparing programming support environments.
In Hunke, H, editor, *Software Engineering Environments*, . North-Holland Publishing Co, 1981.

[5]
Cheatham, T. E., Jr.
Overview of the Harvard Program Development Systems.
In Hunke, H, editor, *Software Engineering Environments*, . North-Holland Publishing Co, 1981.

[6]
ECL Programmer's Manual
Harvard University, Center for Research in Computing Technology, 1974.

[7]
Karr, M.
Gathering information about programs.
Technical Report CA 7507-1411, Massachusetts Computer Associates, 1975.

[8]
Karr, M.
Affine relationships among variables of a program.
Acta Informatica 6(2):133-151, 1976.

[9]
Karr, M.
Summation in finite terms.
JACM 28(2):305-350, 1981.

[10]

Kildall, G. A.
A unified approach to global optimization.
In *Proc Symp on Principles of Programming Languages.* ACM, October, 1973.

[11]

Townley, J. A.
PDS User's Manual.
Technical Report, Harvard University, Center for Research in Computing Technology, 1981.

[12]

Wegbreit, B.
Property extraction in well-founded property sets.
Technical Report, Technical Memo, Harvard University, Center for Research in Computing Technology, February, 1973.

SAEC-0 User's Manual

This manual describes both operational details and the semantic subset of the Stand-Alone ECL Compiler.

21 January 1981

Revised 6 November 1981

By

General Systems Group, Inc.

51 Main Street
Salem, NH 03079
Tel. (603 893-1000

1700 No. Moore Street
Suite 800
Rosslyn, VA 22209
Tel. (703) 524-7242

Table of Contents

1. Introduction

The use of the SAEC-0 compiler is introduced by means of a simple example. The example we will use is reproduced in Appendix I (it is a listing of the file DEMO.ECL). The first routine, demo, is the "main" program. Its action is simple: it reads characters from the console teletype, finds substrings of contiguous digits, makes these into INTs, and prints out the numbers on the console teletype, each on a separate line. If a number is followed by an escape, demo returns after typing it. The second routine is a predicate for determining whether an integer is the ASCII code for some digit. The third routine is the classical recursive algorithm for printing an integer. In addition to the three routines, there are also some definitions of INT and CHAR constants.

The "program" in DEMO.ECL is self-contained, with one exception, the use of the subroutine QUOTIENT\REMAINDER. The ECL source for this routine is contained in the file ECL:SACMAC.ECL. To execute demo interpretively, it is necessary to LOAD SACMAC. Once this is done, one uses the command demo() at top level in ECL, and the program will execute as advertised, returning to top-level when it is done.

We now consider compiling demo. The first step is to get into ECL and

```
->RESTORE("ECL:SAEC0");
->LOAD "ECL:DEMO";
```

Beware that SACMAC.ECL (and ECL:MACH.BIN) must *not* be loaded at this point. Doing so will destroy the proper compilation of macros (see the section Macros). At this point, one may execute demo just as before (the routines in SACMAC come in with the RESTORE). To compile demo, the next step is to set the variable ProgramName to the SYMBOL which names the main program:

```
->ProgramName <- "demo";
```

Next, one issues the Compile command. Its first argument is the primary name of the .MAC file which will be produced, and subsequent arguments name the entities to be compiled:

```
->Compile("DEMO", demo, IsDigit, PrintInt);
```

When this command finishes, the file DEMO.MAC will contain the machine code for the three routines. A listing of DEMO.MAC is in Appendix II. To prepare the PDP-10 program which can actually be run, one uses the standard commands:

```
COMPIL DEMO
LOAD DEMO
SAVE DEMO
```

To use the compiled version of DEMO, the following command is issued at monitor-level:

```
RUN DEMO
```

This corresponds to the command demo() which, when given from ECL's top-level, executes demo interpretively. Naturally, when the compiled version of DEMO exits, the user console is left sitting back at monitor level, the point from which the execution started.

This concludes the discussion of the preparation, compilation, and running of demo. The reader may duplicate as much of the protocol as he desires, if only seeing is believing (the file DEMO.ECL is maintained in ECL:). We now turn to a more thorough discussion of using the compiler.

2. ProgramName

As discussed above, the variable ProgramName is set to a SYMBOL whose value is the EXPR for the main program. This EXPR must have no arguments, and any result it returns will be ignored. The reason that the user must name the main program is that the job start address (required by the RUN command) must be indicated in one of the .MAC files which constitute the compiled version of the program. The file which contains the main program also has in it the cover routine which sets up the stacks and calls the main program.

The Compile command requires that ProgramName be non-NIL before beginning any compilation (it is NIL after RESTORing SAEC0). If ProgramName is NIL, Compile BREAKs, and a SYMBOL can be supplied via CONT.

3. Stacks

SAEC-0 produces object code which manages two stacks, called the main stack (MS) and the auxiliary stack (XS). The stacks allow recursive programs, and support ECL's locative expression semantics.

SAEC-0 does not support expandable stacks. The default sizes for both the MS and XS stacks is 128. These may be overridden by setting the variables LengthMS and LengthXS to the desired values. Space for the stacks is allocated in the .MAC file which contains the main program.

SAEC-0 also does not support graceful handling of stack overflow. If this happens, it is trapped by the monitor, which issues the standard error message.

4. FileEntitiesStructure

This variable is essentially a list of sublists, where the first element of each sublist is a filename, and the remainder is a list of names of the entities in the file. To examine the variable, use UNPARSE. Doing so after the above example of Compiling demo will result in the following output:

```
< DEMO(demo, IsDigit, PrintInt) >;
```

In this case, there is only one sublist because there is only one file. In general, there may be several. As this example shows, the data structure pointed to by FileEntitiesStructure may be modified by the Compile command. The rule is that if Compile sees a non-NIL list of entities (second and subsequent arguments), it will associate that list with the file name (first argument) in FileEntitiesStructure (replacing a previous list associated with the file, if the file is already there). But if Compile's argument is only the file name, that file must be in FileEntitiesStructure, and the list of entities is obtained from that already associated with the file.

5. The Dictionary and Incremental Compilation

The variables ProgramName, LengthMS, LengthXS and FileEntitiesStructure are a subset of the variables which constitute the "dictionary" for a program. The most complicated data structure in the dictionary is the symbol table, associating with the top-level variables of the user's program various items necessary for compilation, including "assembler compatible names." To understand how to do incremental compilation, i.e., recompiling a single file of a multi-file program, it is helpful to understand the role of assembler compatible names. Suppose a function in one file calls a function in another file. This is accomplished by defining all functions with global names, so that cross-file references are resolved by the linking loader. Since ECL SYMBOLs do not follow MACRO-10's naming rules, it is necessary to maintain a correspondence between ECL names and assembler compatible names. This is done automatically by SAEC-0, which guarantees that different ECL SYMBOLs have different assembler compatible names. But in order to make this automatic, it is necessary to know all of the assembler compatible names which have been defined so far. And this brings us to incremental recompilation. When recompiling one file, it is necessary to know the correspondence which has already been set up, because .MAC files which are not being changed already have certain assembler compatible names in them. Thus, after first compiling a program, if one expects to do any incremental recompilation, it is necessary to save the dictionary. This is done with the command :

```
->DumpDictionary(...);
```

The argument is the primary name for the dictionary file; the extension .DCT is supplied automatically.

To do an incremental compilation, one first RESTOREs SAEC0, as for any compilation. Then the entities which will be needed in the new compilation are defined. These need not include entities which appear in files that are not being recompiled. *After* all this is done, the dictionary for the program is obtained with the following command:

```
->LoadDictionary(...)
```

The user may issue dictionary commands only *after* defining the entities to be compiled. To guard against mistakes, LoadDictionary tries to guarantee that no improper commands have been issued, by refusing to load if there is a non-empty dictionary. If it is desired to over-write the dictionary, the following command may be issued before LoadDictionary, or before CONTinuing from the "non-empty dictionary" BREAK:

```
->FlushDictionary();
```

Understand that this flushes any dictionary commands which may have appeared when setting up the program to be recompiled. Changing the dictionary is discussed in a subsequent section.

The user must be careful to dump the dictionary after any incremental compilation, so that it is always up to date. Failure to do so may result in files which don't linking-load properly, or worse, which do, but which don't execute properly.

6. Default Dictionary Behavior

In order to spare the user the tedious task of preparing a dictionary, SAEC-0 tries to construct a reasonable dictionary on the basis of the top-level bindings of the ECL SYMBOLs. This is why the standard paradigm for compiling is to simply LOAD the program as if one were going to execute it -- all the top-level bindings will be strong indications as to the mode and behavior of the SYMBOLs which the compiler will encounter. The section Dictionary Commands describes ways to override default assumptions.

The first rule of the dictionary is that if a symbol is already in the dictionary, its TLB is irrelevant -- all pertinent information is also in the dictionary. This means that in recompilation, only these entities which have to be recompiled need be loaded. Those

SYMBOLs whose corresponding entities are in other files will have all relevant information in the reloaded dictionary. The rest of this section describes what happens when the SYMBOL is not in the dictionary.

The first step, as suggested above, is to look at the mode of the value of the SYMBOL's TLB. If it is NONE, then there is an error. The most common case is that the mode is ROUTINE, in which case there are four possible sub-cases: the ROUTINE is NIL, or is an EXPR, a SUBR or a CEXPR. If NIL, it is an error. For EXPRs, a PROC mode is constructed by looking at the EXPR header. This PROC mode and a newly constructed assembler compatible name are the only items necessary to complete the dictionary entry for the SYMBOL. If the mode is a SUBR or CEXPR, the PROC mode can be constructed from the information contained in those objects, and there is no need for an assembler compatible name. In all these cases, the SYMBOL is treated as a constant.

If the mode of value of the SYMBOL's TLB is a PROC mode, then that PROC mode is put in the dictionary, as well as an assembler compatible name. If the value is NIL, the name is treated as a routine-valued variable in the object code; otherwise it is treated as the entry point of a routine.

Objects other than routines fall into two classes: little and big. By "little", we mean objects whose size is one word (e.g., INT) or less (e.g., CHAR, BOOL). Little objects are treated as follows: if the value of the SYMBOL's TLB equals the default value of its mode, the symbol will be treated in the object code as variable of that mode. The mode and an assembler compatible name are entered into the dictionary. If the value is not the default one for the mode, the symbol will be treated as a constant, and its value will appear directly in the object code. For example, consider the SYMBOL ASCII\LF in DEMO.ECL. Its value is not equal to CONST(CHAR), and as a result, ASCII\LF is treated in the source exactly as if the programmer had written %<line feed>. In examining the MACRO-10 code, the integer 10 appears in its place. There is no assembler compatible name for the constant. Note that the rule "default value implies treatment as a variable" applies uniformly to ROUTINE, which are EXPRs (not default ROUTINEs), PROC's and little modes.

Big objects are treated by the same rule about defaults as little ones. But because they are big, all big objects are given assembler compatible names, so that all references of them will be to a shared object, even if it is constant. In the object code, all references to big objects will be treated essentially as variables, using the assembler compatible name to do so.

The difference between big constants and big variables will become apparent in the next section.

7. GetGraph and GenCode

The Compile command is actually a composite of two commands: GetGraph and GenCode. The GetGraph command takes its arguments in exactly the same way as the Compile command. It prepares "graphs", an intermediate data structure, for each of its EXPR-type arguments, and puts them into a file whose primary name is the first argument to GetGraph, and whose extension is GRF. GetGraph also puts a copy of each the big constants and variables into the GRF file. After preparing the GRF file, GetGraph exits.

After calling GetGraph, Compile puts its final argument into the file COMPIL.TMP, and RESTOREs ECL:CG. When this file is RESTOREd, it checks to see if the file COMPIL.TMP exists. It is the GenCode command which produces code from the GRF file. After finishing, it deletes COMPIL.TMP. If COMPIL.TMP does not exist when CG is restored, then the user is sitting at top-level in ECL. He may issue the command GenCode ("xxx") to produce machine code from the xxx.GRF file.

The whole truth has now been told about the Compile command.

8. What Compile Really Does

It is now possible to examine in detail what the Compile command does with each of the arguments in the entity list. It first looks up the symbol in the dictionary. If there is no entry for it, or if the entry indicates that it is a little (i.e., unnamed) constant, an error is indicated. If there is a name for the entry, then it is either a variable or a "big constant" (*including* EXPRs). Variables are assembled into the read-write area (low segment on the PDP-10) and big constants into the read only area (high segment on the PDP-10). In both cases the assembler compatible name is issued as the label. For EXPRs, the EXPR compiler is used to produce code. For variables and big constants, the constant compiler is used to produce MACRO-10 text which will properly generate the internal representation for the value.

9. Dictionary Commands

It is clearly necessary to have a means of overriding the default dictionary behavior. One would like, for instance, to declare that a particular SYMBOL is the constant integer 0, or that another SYMBOL is an integer variable, initialized to 1 when the program starts. These may be achieved with the following commands:

```
EnterConstantIntoDictionary ("zero", INT, 0);
EnterVariableIntoDictionary ("count", INT, 1);
```

In each case, if the third argument is omitted, it is taken from the TLB of the first argument. But if it is supplied, the TLB of the first argument is never examined. From this rule, it is apparent that the proper time to issue such commands is *after* the program has been loaded. In a reasonable sized program, the usual practice would be to prepare a file of such commands, which would then be LOADed.

A third dictionary command is available which has several uses:

```
EnterSynonymIntoDictionary("new Symbol", "old symbol");
```

The dictionary entry for the new symbol is made identical with that for the old symbol. If the old symbol is not in the dictionary, it is put there using the default rules. One use of this is to provide sharing of EXPRs (and other big constants). For example, if the source file has the command `NewExpr <- OldExpr`, so that both SYMBOLs refer to the same DTPR, they should be made synonymous. Another use of this is to circumvent the problem of NOFIX operators. Suppose `OldExpr` has been NOFIXed, so that the command `Compile("file", OldExpr)` does not have the proper internal structure (because `OldExpr` is represented as a function call with no arguments). Then one can construct a *synonym to* `OldExpr`, and use that name in the `Compile` command.

10. The SAEC Library

In certain situations SAEC-0 will emit code which relies on a few routines of a run-time support library. The `DEMO` program did not do this, which is why we were able to issue the simple command `LOAD DEMO`. When code relies on run-time routines, it is necessary to load (via the PDP-10 loader, not the ECL command) the file ECL:ECLLIB.REL, preferably in "library mode", so that only the required routines will be loaded. The syntax is:

```
LOAD    DEMO,ECL:ECLLIB/LIBRARY
```

If in the process of loading (not the ECL-command) there are undefined globals beginning with "$", then ECLLIB is not being properly searched. Other undefined globals are the

user's problem.

To summarize, it never hurts to load ECL:ECLLIB in library mode, and sometimes it is required.

11. Note on Built-in Operators

In the introduction, we discussed the fact that when DEMO is loaded after the RESTORE command, it can be executed just as if DEMO had been loaded after having loaded just SACMAC into a fresh ECL. This statement, of course holds for all programs, and is true even if the loading of the program involves redefining of SYMBOLs which name built-in operators. For instance, suppose the program redefines the symbol +:

```
OldPlus <- +;

+ <- EXPR(...) ... OldPlus ... ;
```

Then all occurrences of + in the program will be treated as calls on the EXPR, whereas the use of OldPlus inside the new + operator will be treated as the built-in operator which + defines in unmolested ECL. The reason that this works is that all of the dictionary lookup is keyed on the values of SYMBOLs, not on the SYMBOLs themselves -- the SAVEd SAEC0 has already associated the value of + with the proper internal data, and will recognize it no matter who points to it.

The only caveat in all of this is that while the SAEC-0 is still running interpretively, any new definitions of built-in SYMBOLs that it uses, such as +, had better operate consistently with the built-in definitions for the modes for which they were intended. And too, redefining + as above would presumably slow up the compiler considerably. But the principle is clear, and once SAEC-0 is compiled completely (on the compatible compiler), this caveat will not apply.

12. Note on Sharing

Again in the spirit of reproducing in compiled code the semantics of the source program, SAEC-0 pays attention to shared top-level bindings. Suppose, for example, that in the preparation of the source program, for execution and therefore for compilation, the TLBs (not just the values) of A and B are set to the same REF, as would be the effect of

```
"A".TLB <- "B".TLB
```

Then A and B are the same variable in any SAEC-0 program. In fact, this is a theorem

automatically guaranteed by a successful SAEC-0 compilation, because manipulation of SYMBOLs' TLBs is forbidden in SAEC-0 programs. To preserve the semantics, both of these SYMBOLs map to the same dictionary entry, and thus are treated as the same variable in the object code. It is as if EnterSynonymIntoDictionary were used. But only one of the SYMBOLs needs to be the argument of a Compile command, since that will take care of both. In fact, if both are given in a Compile command, a "multiply-defined label" error will arise while either assembling or loading.

In the same vein, if two SYMBOLs which are entered into the dictionary as routine constants have the same *value* (not necessarily the same TLB), they too will be treated as one by the SAEC-0 compiler. Specifically, all references to either will be a reference to the same piece of compiled code, and only one should be the argument of a Compile command. A corollary to this is that the program may have routines as compiled constants (or initialized variable values), and these constants will be shared in the proper manner. The only restriction is that the SYMBOLs referring to the EXPRs must be entered in the dictionary before the constant is compiled. For example, suppose that the following sequence is in the load file which prepares the program:

```
A <- EXPR(x:INT; INT) ...
B <- EXPR(y:INT; INT) ...
C <- CONST(VECTOR(2, PROC(INT; INT)) OF A, B)
```

Then if the assembler-compatible names are the same as the ECL names, and if A and B are entered before the command Compile(..., C, ...), the following text will appear for C in the object code file:

```
C::
A
B
```

When assembled and loaded, C will correctly serve as a vector of two routines.

13. Macros

In most cases, a call on an ECL routine will be compiled as code to set up the arguments, followed by a machine call instruction. However, there are a few routines, calls upon which will be treated in some special way by the compiler. These routines are collected in the file ECL:SACMAC.ECL, whose acronym means "SAEC-0 macros". The routines presently in SACMAC are:

```
QUOTIENT\REMAINDER MOD ABS LSH LROT LAND LOR LXOR LNOT
        FetchLH FetchRH LoadByte DepositByte
```

Currently, the user cannot add to this list, but it is a trivial matter for the compiler maintainers to do so.

The remarks in the section Note on Built-in Operators apply to the routines in SACMAC as well as SUBRs such as +. Thus if Q\R is set to QUOTIENT\REMAINDER, calls on Q\R will be treated in a special way. It is because of this feature that SACMAC may not be loaded after having restored SAEC0. Loading SACMAC causes the loading of ECL:MACH.BIN (for LROT and the routines used by LXOR), which explains why ECL:MACH.BIN may not be loaded after having restored SAEC0.

14. The SAEC-0 Subset

SAEC-0 handles only a very restricted subset of ECL, with the restrictions arising from two sources: the necessity to produce stand-alone code, and the unwillingness/insufficient resources to produce a compiler for a richer subset.

The most obvious restriction which an SAEC-0 program must obey is that there are no PTRs. Unrestricted use of PTRs clearly leads to a requirement for considerable run-time support, and the detection of the non-necessity of, say, garbage collection, in a program using PTR's involves more analysis then we are willing to put in SAEC-0. Similar reasoning leads to the disallowing of generic modes (ONEOF(...) and ANY). Naively implemented, generic modes require that MODE be an allowable data type, and MODE is a PTR(DDB), and PTRs are not allowed. One can easily imagine a compiler's being smart enough to achieve the effect of generic modes without actually using the data type MODE (non-naive implementation), but this leads to more analysis than we care to handle in SAEC-0 programs. For the same reasons that disallow generic modes and pointers the modes which do occur in the program (namely in DECLs, etc.) must be constants which can be detected by the compiler. We thus have our first restriction on SAEC-0 programs:

- All modes must be constants, non-generic, and not have PTRs. They must appear only in DECLs, EXPR headers, CONST, marks and RETURNs, LOWERs and LIFTs.

This restriction leads directly to the following:

- The UNEVAL and LISTED bindclasses are prohibited.

Both of these yield PTRs. (Note that some SUBRs which have LISTED bindclass are allowed, such as AND.) A further consequence of not having pointers and of compiling one

routine at a time is:

- Non-local names must be top-level variables.

It is easy enough for a compiler to handle local names and top-level names. But referencing names DECLared in other routines requires either a SYMBOL -- PTR(ATOM) -- to do the dynamic lookup, or else analysis to figure out which variable is being referred to. Neither is available in SAEC-0. Because SYMBOLs are not available, we also have the following restriction:

- The SYMBOLs in marks and RETURNs must be constants.

Because all modes must be known constants:

- Only "dot-selection" is permissible from a STRUCT.

Because of lack of implementation time, RETURNs must also obey the following rule:

- The INTs in RETURNs must be constants, and the RETURN must arrive at a local mark.

Clearly SAEC-0 cannot handle all of ECL's built-in operators, if for no other reason that many operate on pointers. Those which are handled include all the SYSTEMs but one:

```
BEGIN FOR DECL EXPR CASE => #> -> +> . [ <-
```

(Poser: what's missing?) It also includes the following SUBRs:

```
<< RETURN
AND OR NOT
CONST
+ - * /
LT GT LE GE
# =
INT\CHAR CHAR\INT
PRINT INCHAR
```

The mode of the first argument to PRINT must be CHAR, and a second argument is not allowed. Interpretively, this results in the second argument of COPORT, and what is compiled is a monitor call to print one character on the user console. Similarly, INCHAR is required to have no arguments, and is compiled to a monitor call to read one character from the user console.

15. Extended Mode Behavior Functions in SAEC-0

Since the SAEC-0 compiler does not support SYMBOLs, FORMs, or MODEs, it cannot support EL1's normal behavior functions for extended MODEs. In place of these functions, the compiler provides a means of automatically inserting a function call at the point where a MODE behavior function normally would have been invoked. Although this new type of behavior function cannot be passed a MODE or FORM, different functions can be inserted in the different circumstances that are normally inferred from the behavior function's arguments.

15.1. UGF

The user generation function for a *MODE* is normally given the desired *MODE*, the bindclass, and the specification values used in generation. If the MODE argument is omitted, then a function can be used as the UGF of only one MODE. This restriction is not serious, since it would also arise from the absence of generic MODEs in SAEC-0. The bindclass can also be discarded, if distinct functions can be inserted for the different bindclasses. The list of specification values must be passed to the function. Since SAEC-0 does not provide a FORM LISTED bindclass, the behavior functions can accept only a fixed number of objects.

The compiler is informed of a UGF by calling:

```
SetUGF(m, bindclass, f),
```

where,

m	is the MODE to which the generation function is attached,
bindclass	is the SYMBOL of the bindclass that is handled by this UGF, and
f	is the generation function.

Consider the following examples. Assume that ParserLexemeUGF\OF is an EXPR with three arguments and ParserLexemeUGF\BYVAL is an EXPR with one argument. First we call:

```
SetUGF(ParserLexeme, "OF", ParserLexemeUGF\OF);
SetUFG(ParserLexeme, "BYVAL", ParserLexemeUGF\BYVAL);
```

Then the compiler will replace:

```
CONST(ParserLexeme OF State, FixityFlags, TerminalCode)
```

by

```
ParserLexemeUGF\OF(State, FixityFlags, TerminalCode)
```

and:

```
DECL Token:ParserLexeme BYVAL OldLexeme;
```

with:

```
DECL Token:ParserLexeme BYVAL
  ParserLexemeUGF\BYVAL(OldLexeme);
```

The UGF takes only one argument, the example value, in generation by example. In generation by size or generation from components, the UGF can have zero or more arguments.

15.1.1. SUPUGF

The compiler is informed of a SUPUGF flag by calling:

```
SetSUPUGF(m)
```

where,

m — is the MODE whose SUPUGF flag is being set.

15.1.2. Generation By Example

Since SAEC-0 treats the bindclass LIKE in the same manner as the bindclass BYVAL, we need only consider the bindclasses BYVAL and SHARED.[1] At each point where a generation by example occurs, the compiler considers the following factors, in this order, while inserting a UGF.

1. Does the expected MODE have a UGF? The compiler looks for a UGF with the same expected MODE, MODE in hand, and bindclass. If the expected MODE does not have a UGF, nothing is inserted and the generation is handled by the compiler.

2. Does the expected MODE have its SUPUGF flag set and does the expected MODE COVER the MODE of the value in hand? If both conditions are met, nothing is inserted.

3. If the compiler knows of a suitable UGF, then a call on that function is inserted into the program.

---

[1] Remember that the class of SHARE MODEs (SHARE(m)) must be treated as if their bindclass was SHARED, instead of LIKE.

15.1.3. Generation By Size

The compiler tries to choose a UGF of the correct type that has arguments equal in number to the specifications. If there is no such UGF, then the UGF with the largest number of arguments less than the number of specifications is used, and the extra arguments are discarded.

All of the dimension specifications are evaluated, even if the extra ones are discarded. Missing specifications are supplied from the DADV of the MODE. Notably, in generation by default (e.g. DECL X:M), SAEC-0, like ECL, treats this case as generation by size, with M's DADV as the specification value.

15.1.4. Generation From Components

This type of generation is for user-defined bindclasses, as well as the conventional bindclass "OF". These bindclasses do not require a fixed number of specifications, so SAEC-0 allows a MODE to have more than one UGF for each of these bindclasses. The compiler tries to use the UGF with the number of arguments equal to the number of specifications. If none of the UGFs have the appropriate number of arguments, SAEC-0 uses the function with the largest number of arguments less than the number of specifications.

Specification values are treated as arguments to a normal function. That is, only those values corresponding to an actual parameter are evaluated and missing ones are supplied by constructing a default value.

15.2. UCF

UCFs are inserted when the result of a generation is not COVERed by the expected MODE and the necessary conversion function exists.

The procedure for associating a UCF with a MODE is:

```
SetUCF(m, TargetMode, f),
```

where,

m	is the MODE whose UCF is being defined,
TargetMode	is the MODE of the argument to the UCF, and
f	is the UCF. Its argument is the object being converted.

15.3. UAF

The assignment function is passed the two arguments to the assignment operator. Separate assignment functions are necessary for each MODE of the right operand. To preserve the semantics of EL1, the assignment function should return the left operand.

The procedure for associating a UAF with a MODE is:

```
SetUAF(LM, RM, f)
```

where,

LM	is the MODE whose UAF is being defined;
RM	is the MODE of the right operand of the UAF; and
f	is the UAF. Its first argument is the left operand and it second argument is the right operand of the assignment operator. The first argument should be taken SHARED.

15.4. USF

The selection function for dot selection is chosen on the basis of the MODE of the selectee and the number of selectors. A USF, f, that accepts a certain number of selectors as arguments is associated with the MODE of its selectee by calling:

```
SetUSF\BRACKET(m, f)
```

where,

m	is the MODE of the object accepted by the USF and
f	is the USF.

The first argument to f is always the selectee, and should have the bindclass SHARED. The other arguments are the selectors, in the order in which they appeared in the source program. Given an object of MODE m and n selectors, the compiler attempts to find a USF that accepts MODE m and has n + 1 arguments (remember that the object as well as the selectors must be passed to the USF). If no USF meets these conditions, the compiler takes the following steps.

- If there is no USF for m then the compiler uses conventional bracket selection.
- If there are USFs of the correct MODE, but none of them have the right number of arguments, then the compiler uses the function that accepts the largest number of arguments less than the number of selectors. The extra selectors are used to perform conventional bracket selection on the result of the USF.

- If all the USFs accept more selectors than are provided, the compiler uses conventional bracket selection.

In dot selection, the USF is chosen on the basis of the MODE of the object being selected and the SYMBOL of the selector. A USF, f, is associated with objects of a particular MODE and a selector by calling:

```
SetUSF\DOT(m, selector, f)
```

where,

m	is the MODE of the object accepted by the USF,
selector	is the SYMBOL of the selector, and
f	is the USF.

The only argument to f is the object being selected from, which should have the bindclass SHARED.

15.5. UDF

This function will not be supported until length unresolved objects are supported by the compiler.

15.6. UPF

There are no UPFs since the PRINT functions only accepts CHARs and prints on the TTY.

16. Installation

Moving SAEC-0 from one ARPA-net node to another is a simple matter. First, transfer from ECL: of the source node to ECL: of the destination node the following files:

```
SACMAC.ECL ECLUNV.UNV ECLLIB.REL DEMO.ECL
```

Then, using the file RETCG.CMD in the source area of the source node, move all the source files to the source area of the destination node (the commands in RETCG.CMD may have to be edited depending upon whether one is doing a "retrieve" or a "store" in FTP). Then, in the destination nodes's source area, get into ECL and perform the following commands:

```
->LOAD "CG";
  [type out of rewrite timings]
->SAVE("SAEC0");
```

Finally, move SAEC0.LOW into ECL:, and go through the protocol of the first section to make sure everything works.

I. ECL:DEMO.ECL

```
demo <-
  EXPR()
    BEGIN
      DECL c:INT BYVAL 0;
      REPEAT
        REPEAT
          c <- CHAR\INT(INCHAR());
          IsDigit(c) => NOTHING;
        END;
        DECL i:INT BYVAL c;
        REPEAT
          i <- i - CodeFor0;
          c <- CHAR\INT(INCHAR());
          IsDigit(c) #>
            [] PrintInt(i); PRINT(ASCII\CR);
               PRINT(ASCII\LF) [];
          i <- 10 * i + c;
        END;
        c = CodeForEsc => NOTHING;
      END;
    END;

IsDigit <-
  EXPR(c:INT BYVAL; BOOL) CodeFor0 LE c AND c LE CodeFor9;


PrintInt <-
  EXPR(i:INT BYVAL)
    BEGIN
      i GE 10 ->
        BEGIN
          DECL qr:VECTOR(2, INT) BYVAL
            QUOTIENT\REMAINDER(i, 10);
          PrintInt(qr[1]);
          i <- qr[2];
        END;
      PRINT(INT\CHAR(i + CodeFor0));
    END;
```

```
CodeFor0 <- CHAR\INT(%0);

CodeFor9 <- CHAR\INT(%9);

CodeForEsc <- 27;

ASCII\CR <- INT\CHAR(13);

ASCII\LF <- INT\CHAR(10);
```

II. DEMO.MAC

```
TITLE DEMO
 SEARCH ECLUNV(ECL:ECLUNV)
 RADIX 10
$MAIN$:
 MOVE MS,[IOWD 128,$MSBEG]
 MOVE XS,[IOWD 128,$XSBEG]
 PUSHJ MS,DEMO
 EXIT
 $MSOVF::
 OUTSTR [ASCIZ /MS Overflow/]
 EXIT
 $MSBEG: BLOCK 128
 $XSBEG: BLOCK 128
DEMO::
$$1:
PUSH MS,[0]
$$2:
INCHWL $0
MOVEM $0,-<0>(MS)
PUSH MS,-<0>(MS)
PUSHJ MS,ISDIGI
JUMPE $0,$$2
$$3:
PUSH MS,-<0>(MS)
$$4:
MOVE $0,-<0>(MS)
SUBI $0,48
MOVEM $0,-<0>(MS)
INCHWL $0
MOVEM $0,-<1>(MS)
PUSH MS,-<1>(MS)
PUSHJ MS,ISDIGI
JUMPE $0,$$6
$$5:
MOVEI $0,10
IMUL $0,-<0>(MS)
ADD $0,-<1>(MS)
MOVEM $0,-<0>(MS)
JRST $$4
$$6:
PUSH MS,-<0>(MS)
PUSHJ MS,PRINTI
MOVEI $0,↑D13
OUTCHR $0
MOVEI $0,↑D10
```

```
OUTCHR $0
MOVE $0,-<1>(MS)
CAIE $0,27
TDZA $0,$0
SETO $0,
JUMPE $0,$$8
$$7:
SUB MS,[XWD 1,1]
SUB MS,[XWD 1,1]
POPJ MS,
$$8:
SUB MS,[XWD 1,1]
JRST $$2
ISDIGI::
$$9:
MOVEI $0,48
CAMLE $0,-<1>(MS)
TDZA $0,$0
SETO $0,
JUMPE $0,$$13
$$10:
MOVE $0,-<1>(MS)
CAILE $0,57
TDZA $0,$0
SETO $0,
JUMPE $0,$$13
$$11:
MOVEI $0,1
$$12:
SUB MS,[XWD 2,2]
JRST e2(MS)
$$13:
MOVEI $0,0
JRST $$12
PRINTI::
$$14:
MOVE $0,-<1>(MS)
CAIGE $0,10
TDZA $0,$0
SETO $0,
JUMPE $0,$$16
$$15:
MOVE $0,-<1>(MS)
IDIVI $0,10
PUSH MS,$0
PUSH MS,1+$0
MOVE $0,0+-<1>(MS)
```

```
PUSH MS,$0
PUSHJ MS,PRINTI
MOVE $0,1+-<1>(MS)
MOVEM $0,-<3>(MS)
SUB MS,[XWD 2,2]
$$16:
MOVE $0,-<1>(MS)
ADDI $0,48
OUTCHR $0
SUB MS,[XWD 2,2]
JRST @2(MS)
END $MAIN$;DEMO
```


DATE
ILME