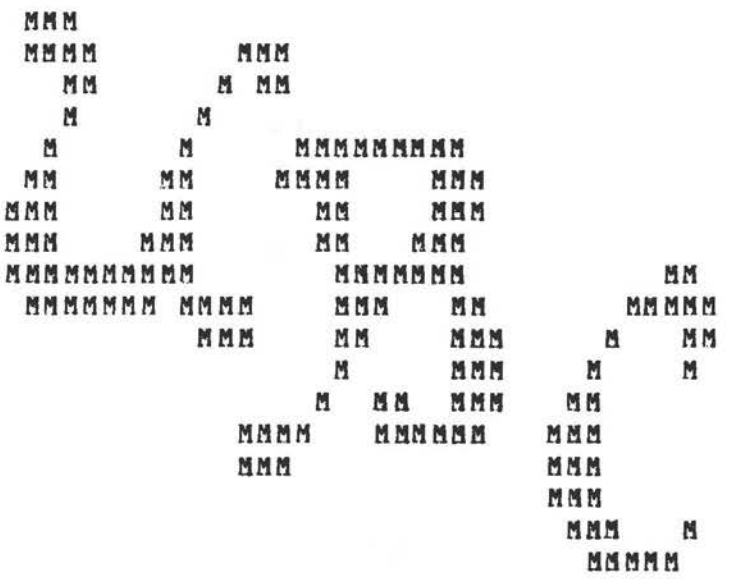

FUNL Semantics: Work Towards an UNCOL

by

R. A. Fraley

Technical Report 77-9

August 1977

Department of Computer Science
University of British Columbia
Vancouver, B. C.

FUNL Semantics:
Work Towards an UNCOL

R. A. Fraley
University of British Columbia
August 27, 1977

ABSTRACT

An intermediate semantics language, applicable to many source languages and machines, is proposed in this paper. Over its domain and range it promises many of the advantages of the original UNCOL project. Data abstraction is used to hide machine features. The language hides from the source compiler all implementation representations and conventions, except for a few descriptive constants. The semantic model is expandable by means of a library. Higher level semantic models may be implemented in FUNL, reducing compiler writing effort.

FUNL Semantics: Work Towards an UNCOL

The Unlanguage processor design philosophy encourages the development of modular, extensible, evolutionary languages which are transportable. FUNL is the First UNLanguage processor [4,5]. This paper presents the semantic model which has been developed for FUNL.

As the design of FUNL progressed, its capabilities showed many similarities to those of UNCOL [6,11]. UNCOL, a UNiversal COmpiler Language, was conceived in the late 50's, but was abandoned as being an impossible task. Because of this failure the UNCOL objective is often considered to be impossible [9]. This conclusion seems to be based on the failure of the original project and the obvious conclusion that no one language can contain all possible ideas.

The FUNL semantic model is not universal. But for some class of source languages the FUNL semantics primitives seem to provide an efficient description of their concepts. Likewise, the FUNL semantics can be efficiently implemented for some class of computers. While FUNL will be most useful if these classes are large, reduction of implementation effort will occur if there are more than two members in each [6]. Because modular compilers can be constructed using FUNL, the system offers greater potentials than those described for the original UNCOL. When new compilers are constructed, portions of existing compilers may be utilized. Because of its evolutionary syntax, languages developed using FUNL are more flexible than those designed with conventional techniques.

There are a number of reasons to believe that FUNL will achieve UNCOL's goals where the original effort failed.

1. It has weakened the goal of universality.

2. Its description employs data abstraction methods which have been developed in the past fifteen years.

3. The semantic model is built using higher level primitives.

Data abstraction solves one of the few documented problems which existed with the original UNCOL. Its authors were concerned about using indirect addressing on those machines which have it, yet working correctly for those which don't. FUNL contains an abstract data type LOCATION, which is defined for each implementation. The contents of this type cannot be interrogated by a compiler specification. The semantic model

contains primitives for manipulating locations.

Comparisons

Intermediate languages fall into three general categories: high-level, mid-level, and low-level. High-level intermediate languages tend to restrict the available source languages since their model includes a part of a source language model. Low-level intermediates, on the other hand, tend to exclude some target machines since they are based on a specialized machine model. Mid-level intermediates attempt to exclude the biases of the source language, yet not introduce machine details. The FUNL semantics forms a mid-level intermediate language.

The high-level intermediates, such as GRAIL [1], P-Code [7], and OCODE [10] are generally based on a stack model or a postfix notation model. In either case, a large amount of implementation is still needed to produce machine code for non-stack machines. These intermediates are designed to work with a single language or, in the case of GRAIL, a restricted class of languages. Low-level intermediates, such as INTCODE [9], LOWL [3], portable C intermediate langauge [7], and FLUB [12] are based on a specific machine model. To implement this model in an efficient manner on a different machine requires a large implementation effort if the structures differ greatly. When code has been generated to bypass a resource limitation of the model machine, it must be detected and eliminated by the implementation. Some operations of the model machine must be simulated by the implemntation translator so that they needn't be performed at run time. In many cases, facilities of the real machine can't be used by the implementation because applicable situations cannot be detected.

Only one mid-level language has been found by the author. The "Storage" language which forms the middle abstraction for an Algol 68 compiler [2] appears to be one, but its details have not yet been located.

The Design of FUNL

FUNL was inspired by work on a Pascal compiler [8]. The work performed by the author in re-designing and expanding the compiler suggested some internal structures which were independent of the source language details. At the same time, the implementation of the structures was ignored by most routines. These structures and the routines for manipulating them were isolated and refined. The resulting primitives were subjected to a number of "situations", where each situation was

a code segment in some programming language. A sequence of primitives was needed to handle each situation.

The goal of an UNCOL is flexibility and simplicity for both the compiler and the implementation. There is no way to determine how closely FUNL approaches these goals. Many compilers must be built for many languages, on many machines, and by many people, before the FUNL primitives can be refined to an acceptible degree of universality. Readers are urged to try these primitives with their favorite languages and machines, and to submit descriptions of problem areas and suggested improvements.

FUNL's main advantage is flexibility in memory allocation primitives. FUNL is capable of simulating COBOL structures which have dynamically computed displacements. (This facility is also used in memory management, even when dynamic displacements are not a part of the source language.) It allows the compiler to pass packing information to the implementation without knowing the machine's internal structure. The implementation can choose to pad fields; it knows when comparisons of the packed fields might be performed. FUNL is capable of some type checking. While the compiler is free to attach any type to any position in memory, the implementation can be sure that this assigned type is being used in a consistent manner.

The C semantic model, like many of the others, has a "call" primitive. FUNL attempts to add distinguishing features to procedures so that many different calling sequences can be generated. For example, a compiler might generate different sequences for its own procedures, external procedures, and support routines. Some of the decisions for the details of the calling sequence are left to the compiler, while others are determined by the implementation.

The type-checking facilities are not absolute. Because data fields can be arbitrarily overlapped, the compiler designer can access one form of data using a different description. The choice of descriptor for a given location gives him the power to bypass type checking. This implies that programs might produce different results in different implementations if the compiler allows access to the data representation. Some design for portability therefore remains with the language.

The FUNL semantic model is not small. It has about 100 primitives, not including library modules. These primitives are defined in Appendix I. The definition of these primitives is, of course, subject to change at this stage of FUNL's development. An example which uses some of the primitives is presented in Appendix II.

Machine Types

The data abstraction MACHINE_TYPE defines the representation type being used in an implementation. This type is distinct from source language types. A selected machine type determines the amount of memory needed to store a piece of data. The selection of types includes a BIT type. This suggests a bias towards machines capable of using a binary number representation, although digit tests could replace bit tests.

A number of primitives are provided for dealing with machine types. The FIXED primitive machine types hold integer values. FLOAT types can hold real values. Fields of CHAR type hold a single value from an arbitrarily large character set. An object of POINTER type holds a reference to memory, while a specified number of bits will fit in a BIT type.

Replecation of a type, as in an array, forms another machine type. DYNAMIC types may be constructed for variable length replications. A dynamic type may also be created using the stored size of a data element.

Areas

An AREA is a compound storage unit composed of fields, each field having a machine type. The area can be allocated (given a piece of real memory which can hold its fields), or can be part of a larger area. The fields which form an area are normally concatenated, so that one follows another; they may also overlay each other if desired.

There are two varieties of areas: standard and comparison. In a standard area, the implementation can leave space between fields. This "padding", used to improve access time to variables, will contain an indeterminate value. If two areas are to be compared, all bits become significant. The areas can only be equal if all bits, even padding, agree. (The compiler could do a field by field comparison, but the implementation doesn't have the required information.) If the implementation decides to add padding to a "comparison" area, it must be sure that different copies are not made unequal by their fill bits. If a field allocation caused padding to be added to an area, any assignment to that field must place a standard value in the fill bits. Likewise, if a subfield is referenced, and the subfield is adjacent to the padding, the padding bits must be set again.

Packing

To allow more flexibility in creating data structures, a packing factor may be specified as part of the machine type description. A packing factor specifies the trade-off between the expense of retrieval and the data storage space. Since each implementation will have a different number of choices for the representation of a value, a compiler must be specified without knowledge of this number. An arbitrary decision was made to use values from 0 to 100 for packing factors. To reduce the amount of tuning required when moving a compiler to a new installation, the following rule may be used for assigning representations to packing factors:

> Make a grid showing reference expense (time and space) vs data storage space. For each representation being considered for a data type, "compute" its expense and storage space values. Connect the relevant points by a (minimal) monotonically decreasing function. Divide the expense scale into 100 intervals. A representation which lies on the decreasing function is used for all packing factors from its coordinate to the next representation having greater access expense.

This process is quite arbitrary, but may aid in compiler transportability.

The ends of the packing factor scale have not been specified by the above technique. We can say that "10" means "A little extra overhead is okay if you can save a fair amount of space", while "85" means "Make a dense packing, but don't do anything which requires an inordenate amount of time."

Locations

A LOCATION is an abstract data type which refers to an implementation location. There are three types of locations: actual, displacement, and literal. An actual location refers to a data location in the implementation, while a displacement is the distance from one location to another. A literal contains a constant value, which may be assigned storage space in the final object code.

Locations can be permanent and transient. A transient object disappears after it is referenced in a FUNL primitive. (Certain primitives do not destroy their location parameter.) The primitive routine USE_COUNT allows an object to be used several times before it disappears. For efficient code generation, transient objects should be discarded if all of their specified uses are not needed.

The location model will probably be difficult to implement on a stack computer. Stack machines will probably lie outside the target space of FUNL. (While the UNCOL model could be simulated to re-create the stack sequencing, there is some question as to the desirability of the technique.)

Labels

Positions in the generated code may be marked by LABELs. A label is an abstract type which isolates the assignment of memory locations from the compiler code. Labels may be manipulated by primitives, but label values and relative positions may not be determined. GO TO's are used to branch to these labels.

Lest the reader be alarmed by the GO TO capability, recall that the FUNL semantics lie between the high-level language and the machine language. Flexibility must be given to the compiler writer, not structure. The compiler may impose structure on the source programmer, and prevent his access to labels and the GO TO mechanism.

Functional Primitives

The FUNL functional primitives include all data transformations which can be generated. The selection of these primitives is quite difficult: we want as many as possible so that the compiler may be more easily specified and more machine features can be utilized, yet as few as possible so that the implementation is compact.

It is obvious that the basic arithmetic operators are needed. But the selection of more obscure operators is harder to accomplish. Such operators must be well defined in terms of the available machine types, independent of the representations used in various machines. When duplication was found in the selected primitives, one of the duplicates was omitted. Some common machine instructions produce different results on different machines. For example, an algebraic right shift is usually regarded as being a division by a power of two. But most 2's complement machines truncate away from zero for negative numbers. Instead of providing an explicit right shift, the division operator can check for division by a power of 2 if appropriate.

The results of real number computations usually differ from machine to machine, due to the difference in hardware implementation or word size. The author views this as a fault of the usual representation of real numbers. The FUNL system does not attempt to solve these problems, but should be no worse

for transportability of programs than other systems.

A number of primitives perform tests. These primitives should be able to hide the differences between machines which use a conditional branch or jump based on sign from those which use an instruction skip facility. Special forms of the test primitives are provided for producing a numeric true/false value in place of a transfer of control.

Generation Libraries

There are a number of operations performed by hardware instructions which are not available on many machines or not used by many source languages. Rather than require that primitives for these operations be available at all times, they are placed in the generation library. Standard library routines will probably include edit, translate, table search, bit shift and rotate, and bit count packages.

Non-standard primitives can obviously be added to a library. This technique can be used to experiment with new primitives, and to adapt FUNL to the construction of machine-oriented and special-purpose translators whose transportability is of no concern.

Procedure Invocation

Procedure invocation poses one of the most difficult problems for the FUNL semantics, due to the variety of conventions used in different machines and languages. A single compiler might generate a number of different calling sequences. In addition to the principal calling conventions used by the language, there may be special sequences for support routines, external library routines, code sequences within the program, and so on.

In addition to the transfer of control, a call involves the following actions:

-- Handling the return address.

-- Reserving space for parameters.

-- Passing and receiving parameter values.

-- Saving and restoring registers.

-- Returning a result value.

-- Returning an exit condition code.

Not all of these factors will be a part of all calling conventions, but their very absence is significant for code generation. The exit condition, used in some machines and languages, indicates exit conditions or selection of multiple exits.

Exception Conditions

The handling of exceptional conditions (interrupts) differs widely from machine to machine and language to language. These have been ignored for the most part, in the design of FUNL; they must be provided by support routines which interface the machine and operating system. Likewise, I/O facilities must be provided through support routines.

Three exceptions are integer arithmetic overflow, null pointer tests, and division by zero. The compiler can check an environment constant to see if the implementation is providing zero divide interrupts. The compiler can test to see if null opinter tests and integer overflow interrupts are provided by hardware. If not, the compiler is free to provide its own checks. Overflow checking for integer operations can be defeated.

Object Modules

Sometimes the compiler needs to have control over the final object output modules. It may wish to group certain procedures together for efficient loading, and may wish others to be separated for individual loading. Unfortunately, the formats of different systems vary widely. In some systems the desired grouping may be impractical. An additional problem is created by procedure names. Different systems have different conventions regarding the length of names and the available character set for constructing names. In languages which allow procedure nesting, a name may even be duplicated if inner procedures in two distinct outer procedures have identical names.

One solution is for FUNL to handle all of its own object modules. FUNL needs to manage its own libraries anyway, since library routines may include syntax definitions or user procedures needed during compilation.

Unfortunately, this solution is not practical. To obtain acceptance, FUNL must interface existing systems. It must be able to use external libraries and should be able to produce code which is callable from some other language implementation.

It would be impractical in some instances for an object module to contain code for all library routines, because of the resulting module size. Finally, the combined module must still be executed by the system, and there is no standard method for preparing this final output.

The proposed solution to this chaos is sure to be inappropriate for some implementations. It does, however, provide some expressive power to the compiler yet leave some flexibility for the implementation.

A module is a code unit which can be loaded separately from other modules. Each module can contain procedures and data areas. A module may also contain a main program. One module, the program module, is the main program for a run; the main programs of other modules can be called like procedures. Certain procedures and data areas of a module can be designated external, and may be referenced from other modules. The remaining ones can only be referenced by other procedures of the same module. In many implementations the isolation of local procedures will be accomplished by naming conventions.

Code Generation

Many compilers require that code be generated sequentially. This requires that the compiler perform all of the reordering of the parse tree, never rearranging the code generated at lower levels. FUNL normally uses this method of code generation. The FUNL primitives, generated by the compiler, may optionally be placed in a tree. This tree must be traversed to provide the linear ordering required for the primitives.

In certain situations, reordering of the code generated by the FUNL primitives would be desirable. For example, consider the statement:

FOR N TIMES DO S END

This statement specifies that "S" should be performed N times. We must have some auxilliary counter for use in this statement. If we compile the code for S first, we can determine whether there is an extra register which may be used for the counter, or whether the counter would be better off in storage. Use of the code reordering primitives would enable this action to be taken.

Code reordering is costly, especially when a small implementation is required. Therefore, code reordering primitives are placed in a library module and only included for those compilers which need them. This will enable the evaluation of the reordering technique.

When code reordering is included, there must be additional mechanisms for handling the assignment of temporary storage. In particular, each block of code must contain a description of the temporary storage required by that block. It should be possible to specify that assigned temporaries do not include any used by some specific block. The use of temporaries must be managed on a block basis, so that no temporary created inside a block is used on the outside.

The Environment

The compiler may pass status information to the implementation by means of the OPTION primitive. The options control handling of interrupts, compilation output options, and optimization level. The compiler must also specify the required library routines needed for this compilation, but these primitives are not described in this report.

Several items about the implementation can be utilized by the compiler. The environment specification contains the range of integers supported, and the precision and exponent range of real numbers. It indicates whether integer overflow and division exceptions are trapped, and whether attempts to dereference the null pointer are trapped. A flag indicates whether short (procedure-sized) object modules are required.

Using a Limited UNCOL

The FUNL semantics form a fairly low-level language. The construction of a compiler which outputs FUNL semantics can still be a major undertaking. To save effort, an additional semantic model, using primitives of a higher level, could be implemented using the FUNL primitives. GRAIL is one such language [1]. Let's call this model H. For best results, H should completely enclose the FUNL model, preventing direct reference to any FUNL primitives. If the semantic model is designed appropriately, a number of high-level languages may be implemented in this model. In all likelyhood, a number of source languages which could be implemented in terms of the FUNL semantics could not be implemented in H.

Let M be a collection of machines which have a similar architecture, but which are inappropriate for implementing FUNL. A new unlanguage processor (SUNL?) could be implemented for these machines, and H could then be implemented in SUNL. Compilers which generate H could then be run on the machines of M. The expense of the operation is the design of SUNL and the extra implementation of H. If the class M contains only one or two members, the SUNL semantic model could be simply H, though the potential source language space would be smaller.

Simplification of implementation through construction of new semantic models is not limited to the high-order side of FUNL. Given a number of machines with similar addressing and register structures, models can be constructed for implementing locations and performing register allocation. This will reduce the amount of effort needed to move FUNL to a new machine of the class.

The FUNL Primitives

The reader is cautioned that the primitives described in the index form an initial proposal. As the FUNL semantics package has not received much use, it is subject to change. As experience is gained in several implementations, weaknesses of the primitives will become apparent. Suggestions for change will be gratefully appreciated by the author.

To some, the number of primitives will be appalling. Surely there must be a way of reducing that number. Each primitive has been included to solve a specific implementation problem. Due to the many features of languages and machines, the number of primitives is large. As we gain expreience with FUNL, and as the trend in computer languages changes, perhaps the primitive set can be reduced. The design chosen uses a large number of primitives instead of primitives which have a large number of parameters or parameters which are optionally included.

Bibliography

[1] Applebe, W. F., "A Semantic Representation for Translation of High-Level Algorithmic Languages", Ph D Thesis, Univ. of British Columbia, Vancouver, 1977.

[2] Boom, H., "The Organization of the Object Code Generator in ALGOL 68H", Tech Report IW 33/75, Math. Centrum, Amsterdam, 1975.

[3] Brown, P.J., "Levels of Language for Portable Software", CACM, 15:12, 1059-1062 (1972).

[4] Fraley, R. A., "An Unambiguous Scanner for Special Character Tokens", submitted for publication.

[5] Fraley, R. A., "Unlanguage Grammars and their Uses", Dept. of Computer Science Tech. Report 77-6, University of British Columbia, 1977.

[6] Mock, O. et al., "The Problem of Programming Communication with Changing Machines: A Proposed Solution", CACM, 1:8-9, 12-18, (1958).

[7] Nori, K. V. et al., "The Pascal P Compiler: Implementation Notes", Tech. Report 10, Institute for Information, ETH, Zurich, 1974.

[8] Pollack, B. W. and Fraley, R. A., "Pascal/UBC User's Guide", Tech. Manual TM-2, UBC, Vancouver, Sept. 1976.

[9] Richards, M., "Bootstrapping the BCPL Compiler Using INTCODE", in Machine Oriented Higher Level Languages, van der Pohl and Maarssen (ed), North Holland, Amsterdam, 1974, pg. 271.

[10] Richards, M., "The Portability of the BCPL Compiler", Software P&E, 1:2, 135-146 (1971).

[11] Steel, T. B., "A First Version of UNCOL", Proc WJCC, 371-377, 1961.

[12] Waite, W. M., "The Mobile Programming System: STAGE2", :7, 421-429 (1970).

APPENDIX I: Semantic Primitives

The current formulation of FUNL primitives is presented below. They are presented as procedures with argument lists and results. This procedural description may be replaced by an equivalent tree or tuple description if desired. References to "code" in the descriptions refers to the output of the procedures, not necessarily the final processor output. Some of the data types are abbreviated in the descriptions, as follows:

mt	MACHINE_TYPE
loc	LOCATION
labtab	LABEL_TABLE
cb	CODE_BLOCK
imp_id	IMPLEMENTATION_IDENTIFIER
int	INTEGER

MACHINE TYPE PRIMITIVES

TY_FIXED (min, max, base): mt
Returns a FIXED machine type which is capable of storing numbers in the range from "min" through "max". "base" is 2 or 10, indicating a preference towards binary or decimal numbers. (The compiler cannot force a specific representation.)

TY_FLOAT (precision, min_exp, max_exp): mt
The result type can hold a real value of at least the specified precision and exponent range. The precision is specified in bits.

TY_CHAR (number): mt
Machine type for a single character. The number specifies the number of characters in the character set. The character representation is 0 to number-1.

TY_POINTER (packing): mt
The result type can hold a memory pointer. The parameter is the maximum packing level of the referenced field.

TY_BITS (number): mt
The bit type can contain "number" bits, and may be used for logical operations. If the parameter is zero, the result type requires no space.

TY_PROC (proc_form): mt
The result type can hold a pointer to a procedure of the

indicated form.

TY_LABEL : mt
The result type can hold a pointer to a label.

TY_REPEAT (mt, number, packing, compare): mt
The result type will hold "number" copies of the specified machine type. "packing" controls the placement of objects within the array. If "compare" is true, there should be no unused space (padding) between the elements. Packing factors used in constructing the parameter type limit the packing used in the repetition.

TY_CONCAT (mt1, mt2, packing, compare): mt
The result type can contain both of the parameter types. Additional parameters define the packing conventions. If mt2 is a composite type having packing factor "x", then x is an upper bound on the packing actually used for mt2.

Dynamic Machine Types

The size of a dynamic type must be computed at run time. Each dynamic type contains a location which holds its length. The location may be used at any time the machine type is referenced. Because this may occur at arbitrary references to the type or to locations having the type, the use_count of the location should not be changed. While the size of an object cannot be obtained at compile time, it may be stored for later use.

TY_DYNAMIC (mt, loc, packing, compare): mt
"loc" is the result of an integer expression. It gives the number of occurrences of the machine type. The size of the result type is generally determined at run time. The result type may only be inserted into one location; the last use of this location will invalidate the type. "packing" and "compare" refer to the spacing between elements, correspoinding to parameters of TY_REPEAT.

SIZE_OF_TYPE (mt): loc
The result refers to the size of the specified type. The type may be static or dynamic. It is rounded up to the next nearest addressing unit if necessary.

TY_VARIABLE (loc): mt
This primitive constructs a dynamic type whose implementation length is contained in the specified location.

REPEAT_CT (mt1, mt2): loc
The result contains the number of occurrences of type "mt2" in "mt1". "mt1" usually has a variable size.

Area Construction Primitives

The primitives in this section are used to construct areas. All locations in this section are displacement locations, and uses of locations as parameters do not constitute a use of that location. Machine types used for these primitives must be static. It is possible to overlay two fields using the area construction primitives. It is the responsibility of the compiler to safeguard the use of this capability so that user programs cannot be representation dependent.

NEW_AREA (compare): area
Creates a new area. If "compare" is true, the area might be used in a comparison. If the addition of a field causes padding, this space must be cleared whenever the field is stored.

AREA_CONCAT (area, mt, packing): loc
Allocates space for a field having type "mt" at the end of the area. The result is a permanent displacement location. Its type may differ from the parameter type, depending on the packing factors.

AREA_TYPE (area): mt
Returns a machine type corresponding to the space required for the area as currently composed.

AREA_SIZE (area): loc
The size of the argument, at the end of the current procedure, is contained in the result location. This is the amount of space allocated by an ALLOCATE call.

AREA_FOLLOW (area, loc, mt, packing): mt
Allocates space in the specified area for an object of type "mt" to follow displacement "loc". The result is a permanent displacement location.

AREA_OVERLAY (area, loc, mt, packing): loc
Allocates space in area for a field of type "mt" to overlay "loc". The result is a permanent displacement location. This primitive should be used only if all remaining fields of the area may be overlayed, since representation size of "mt" can vary radically in different implementations.

AREA_MARK (area): area_status
Records the current condition of the area for later restoration. If the area is a temporary area, information on temporary use is also recorded.

AREA_RESET (area, area_status)
Resets the area to the given status. The "area_status" must have been generated for the specified "area". Fields allocated since the status was generated may be overlayed, and temporaries allocated since then are invalid.

AREA_INIT (area, loc1, loc2)
Specifies that the field at displacement "loc1" should be initialized to constant value "loc2" when the area is allocated.

Allocation Primitives

There are three classes of storage known to FUNL: static, stack, and heap.

ALLOCATE (area, class): loc
Allocates enough space to accommodate the area. The result location is a permanent actual location, except for heap allocation, which gives a temporary result. The amount of space reserved is the value of AREA_SIZE (area). If the class is stack or heap, the size allocated is computed at the end of the current procedure, so that additional fields may still be added.

GETSPACE (mt, class): loc
Allocates space for the specified machine type. "mt" must have fixed size if "class" is "static".

DEALLOCATE (loc)
Frees the space and invalidates the actual location passed as a parameter. If the location resulted from a static allocation, nothing is deallocated. If it resulted from a stack allocation, all higher stack locations are also invalidated. "loc" should specify a run-time value for a heap deallocate. The location must specify an entire block created via ALLOCATE or GETSPACE, and cannot be "static".

Literal Location Construction

INT_LOC (integer): loc
Creates a literal location containing the specified value.

REAL_LOC (real): loc
Creates a literal location containing the specified value.

STRING_LOC (string): loc
Creates a literal location containing the specified value.

NILPTR_LOC: loc
Creates a literal location containing a null pointer.

EXTERN_LOC (string): loc
Result references the external routine specified by the string.

PROC_LOC (imp_id): loc
The result addresses the specified procedure.

DATA_LOC (imp_id): loc
The result addresses the specified data area.

LABEL_LOC (label): loc
The result is the address of the label. The label must be located in the current procedure. Forward references are allowed.

CON_CONCAT (loc1, loc2, packing): loc
The result is a literal location which is the concatenation of the parameter literal locations. Explicit types should be inserted into the locations before invoking this primitive. If they are not, a default representation of the constant is selected.

CON_REPEAT (loc1, int, packing): loc
The result is a literal location which contains the constant "loc1" repeated "int" times. The constant fields are packed with the specified packing factor.

Location Operations

USE_LOC (loc, number, store): loc
The result location references the original value as the original, but can be referenced "number" times in primitives. If "store" is true, one or more of these references will be a store. If "store" is false, the value may be moved to a temporary. If some references are not used, "LOC_DESTROY" should be called.

LOC_DEREFERENCE (loc, mt): loc
The result location results from dereferencing the original. The parameter location must be an address or refer to a pointer. The result location has the specified machine type.

LOC_ADDRESS (loc): loc
The address of the data specified by the first location is referenced by the result. The parameter must be an actual or literal location. Note: if the location is a temporary location, it might not be directly addressable. This will require an implementation to move the data as part of this function. The compiler writer must be certain that all future references to this location, or branches to previous uses, will still be valid. For this reason, obtaining the address of a "fast" temporary is discouraged.

LOC_DISPLACE (loc1, loc2): loc3
The displacement specified by "loc2" is applied to "loc1" to obtain "loc3". The result location is a displacement if and only if "loc1" was a displacement. The type of the result is obtained from "loc2".

LOC_SUBCOMP (loc, mt, packing, compare): loc
The parameter location must reference an integer value. The result is the displacement of the ith element of an array, where each element has type "mt" and the location specifies the value i, starting at 0. Negative indexes are permitted. The result has type "mt".

LOC_SUBRANGE (loc1, mt, packing, compare, loc2): loc
Operates like LOC_SUBCOMP, but "loc2" specifies the number of repetitions of "mt" included in the result type. Loc2 must be non-negative. The location derived by calling LOC_SUBRANGE and LOC_SUBTYPE, then combining the results with LOC_DISPLACE must be the same as a single call to LOC_SUBTYPE with the sum of the original indices.

LOC_TYPE (loc): mt
Returns the machine type associated with a location.

LOC_NEXT (loc, mt, packing): loc
The result is a field located after "loc" which has type "mt" and is allocated with the specified packing. This primitive is most useful if "loc" was computed with a variable size. It does not imply an extension of the area; the compiler must ensure that adequate space exists.

LOC_DESTROY (loc, perm)
Destroys a location, making it invalid. This is intended for destroying temporary locations which have uses remaining. It may also be used to destroy permanent locations if "perm" is true. This may reduce the space needed during compilation.

Temporary Storage

The temporary storage primitives allow the implementation to control the allocation of temporaries. If the compiler maintained its own record of temporary storage usage, additional temporaries could not be acquired during code generation.

TEMP_AREA (area, loc)
This primitive specifies the area which may be used for temporary allocation. The area cannot be a "compare" area. The location is a specific allocation of the area. The call applies only to the current procedure. Some implementations may fail during code generation if this call is not made. Packing level 0 is always used when adding to the area.

LOC_TEMP (mt, fast): loc
Creates a temporary location. "fast" is true if fast memory should be used. This may limit the implementation if an expression will be evalutated between this call and the final use of the temporary.

TEMP_RELEASE (loc)
Release the use of the temporary. This primitive does not count as a reference to "loc". Used in conjunction with TEMP_REUSE.

TEMP_REUSE (loc)
Reserves the temporary specified by the location. The location must have been the parameter to "TEMP_RELEASE" as its last reference. This does not count as a use of "loc".

Functional Primitives

INT_DYAD (kind, loc1, loc2, min, max, flag): loc

Perform the integer operation specified by "kind" on the integer operands "loc1" and "loc2". The result value will occupy a space capable of holding a numeric range from "min" to "max". The machine type of the parameters must reflect that of the data, especially regarding the decimal or binary preference. "flag" may include "oflo_test" if overflow testing should occur, and can contain "no_oflo_test" to turn off overflow testing. If neither is specified, the implementation may choose the easier. The possible "kind"s are:

1	+	Addition
2	-	Subtraction
3	*	Multiplication
4	/	Division with truncation towards zero.
5	REM	Remainder of division.
6	MOD	Modulus. Result is non-negative.

INT_MONAD (kind, location, min, max, flag): loc

Performs a integer function on the parameter location. The result is stored in a field capable of holding "min" and "max". The following kinds are available:

1	-	Negation
2	ABS	Absolute value
3	NABS	Negative of absolute value.

REAL_DYAD (kind, round, loc1, loc2): loc

Specifies a function to be performed on two real values. The parameters must have the same precision and exponent range; this is the precision and exponent range of the result. Rounding occurs if "round" is TRUE; truncation occurs if "round" is FALSE. If an implementation does not include the option requested, a single warning should be issued at the end of the compilation indicating that that the other form has been used.

1	+	Addition
2	-	Subtraction
3	*	Multiplication
4	/	Division

BIT_DYAD (kind, loc, loc): loc

Perform a dyadic operation on bit strings. The result has the maximum size of the bit strings. The ten logical dyadic operators are available. These are given in the truth table below:

	loc1	1	1	0	0
	loc2	1	0	1	0
1	OR	1	1	1	0
2	AND	1	0	0	0
3	EOR	0	1	1	0
4	IMP	1	0	1	1
5	DIFF	0	1	0	0
6	NOR	0	0	0	1
7	NAND	0	1	1	1
8	EQV	1	0	0	1
9	REVIMP	1	1	0	1
10	REVDIFF	0	0	1	0

In addition, the operators have assignment forms (11-20), with the result remaining in the first field. The result location is the first operand.

DYAD (kind, loc, loc): loc

Performs a dyadic operation on arguments of equal type. For the current operators, arguments may be FIXED, FLOAT, or CHAR.

1	MIN	Minimum
2	MAX	Maximum

MONAD (kind, loc): loc

Perform a monadic operation. The first three operators require a real location, the next two a bit string.

1	-	Negation.
2	ABS	Absoulte value.
3	NABS	Negative of absolute value.
4	INV	Invert the bits in a set.
5	INVP	Invert in place. Result is parameter loc.

MAKE_BITS (loc1, loc2, mt): loc

The result is a bit string of type "mt" having bits "loc1" through "loc2" set to 1, and the remaining bits to 0.

ASSIGN (kind, loc1, loc2)

Copies loc2 to loc1. The locations must have the same type. The available kinds are:

1	:=	Assignment.
2	:=:	Exchange.

INT_ASSIGN (kind, loc1, loc2, flag)
Perform an integer assignment operator. "flag" defined as for "INT_DYAD". The available kinds are:

10	+:=	Add loc2 and loc1 and assign to loc1.
11	-:=	Subtract and assign to loc1.

MOVE (loc1, loc2, just, fill, loc3): loc
Causes the first location to be moved to the second. Any data type other than fixed and float can be used. If the locations have different lengths, "just" determines whether the right-hand or left-hand sides will be aligned. "fill" determines the contents of any unused portion of loc2, as follows:

1	none	Leave unused portion unchanged.
2	repeat	Repeat the loc1 value through field.
3	fill	Fill remainder of field with loc3.

CONVERT (kind, loc, mt): loc
Convert the value described by "loc" to a value of the specified form. The permitted conversions are: packing changes, fixed base change, fixed to float, float to fixed, and CHAR to and from FIXED. BIT to FIX and its reversal are explicitly omitted so that an implementation can select its own bit ordering and number representation. "kind" indicates rounded (1) or truncation (2) for float to fixed.

Label Primitives

NEW_LABEL : label
Create a new label.

LABEL_DEF (label)
Defines the label as referring to the current position in the output code.

LABEL_EQUATE (label1, label2)
Defines the first label to have the same location as the second. The second must have been previously defined in a LABEL_DEF or LABEL_EQUATE. "label1" and "label2" are associated with the same procedure.

GOTO (label)
Generates a branch to the specified label, which may or may not have been defined. The label must be located in the current procedure.

GOTO_LOC (loc)
Generates a branch to the location, which usually contains a computed label.

EXIT_DEF (imp-id)
Defines the implementation identifier as referring to the current location.

Jump Table Primitives

NEW_TAB: labtab
Creates a new label table.

TAB_ENTRY (labtab, label)
Places the specified label in the table. May not be used for a table if TAB_INCLUDE or TAB_LOC has already been called for the same table. The label must reside in the current procedure.

TAB_INCLUDE (labtab)
The specified table is generated in the code.

TAB_LOC (labtab, loc): loc
The result is an entry of the label table. The parameter location contains the entry number, starting at zero. It may be used in a GOTO_LOC, or a label value may be moved into it.

Conditional Primitives

TEST (kind, loc1, jump, loc2)
Generates a test of "loc1". The value of "jump" (TRUE or FALSE) determines whether a jump is made to loc2 on a true condition or a false condition. "loc2" must be a label or computed address. Only one test is currently provided.

1	ODD	Test low bit of fixed item.

TEST_3WAY (loc, label, label, label)
Tests the value of loc1, and branches to the first label if negative, second if zero, and third if positive.

RELATION (kind, loc1, loc2, jump, loc3)
Tests a relationship between loc1 and loc2. A jump is made to loc3 if the result is the same as "jump" (true or false). Data types of the two parameters must have identical machine types. The following kinds are available:

Fixed, float, char, and char structures:	< <= = >= > ¬=
Arbitrary types:	= ¬=
Bit strings:	SUBSET DISJOINT
Fixed and Bit:	BIT_ON BIT_OFF

COMPARE (kind, loc1, loc2, just, fill, loc3, jump, loc4)
A comparison is performed between loc1 and loc2. Alignment and fill is performed as for MOVE, except that either field can be filled. The jump selection is made as for TEST.

Procedure Module Primitives

These primitives declare the structure of modules, and acquire implementation identifiers for each procedure and data area of the module. Procedure identifiers may be used for procedure bodies, alternate entries, and exit labels.

MODULE (name)
Indicates the beginning of a module having the specified name. Modules may not be nested; the module is terminated by the next MODULE primitive, the end of the program, or an OBJECT_GENERATE primitive. No primitive which generates code may be issued before a module has been established. A module may be re-opened (by specifying a MODULE primitive with the same name) if the previous use of the module only declared names (i.e.: no code was generated).

MODULE_ENTRY : imp_id
The resulting identifier may be used to reference the main program of the current module.

EXTERN_PROC (name): imp_id
The result identifier refers to a procedue of the specified name. This procedure may be referenced by this name from outside the module.

EXTERN_DATA (name): imp_id
Produces an identifier for a data area which may be referenced from outside the module.

LOCAL_PROC : imp_id
Produces an identifier for a local procedure, which may only be referenced within this module.

LOCAL_DATA : imp_id
Produces an identifier for a local data area, which may only be referenced within this module.

OBJECT_GENERATE (ret)
Generate code for all modules which have been output. This assures that the modules have been completely compiled, and may be loaded at compile time if desired. If "ret" is false, no return is made to the compiler.

Procedure Declaration Primitives

Procedure declarations define a calling sequence. The declaration may be shared by many procedures.

PROC_DECL (storage, arguments, saving, result, condition)
: proc_form
The result format defines the calling conventions for a procedure. The compiler is responsible for seeing that the proc_form used in defining a procedure is identical to that used for its call. The parameters may have the following values:

storage
Specifies the type of storage to be used for the return address and register save area.

STACK_STORAGE
Return address is placed on the stack.

STATIC_STORAGE
Return address is placed in static memory.

LOCAL_STORAGE
Return address placed in storage local to surrounding procedure.

arguments
Specifies the argument passing conventions.

STACK_ARGS
Arguments are placed in the stack, within the stack area used for the called procedure.

LOCAL_ARGS
Arguments are placed in memory local to the calling procedure, allocated in the temporary area.

EXTERNAL_ARGS
An external convention is used.

FAST_ARGS(n)
A fast calling convention is used. This assumes at most n arguments, which must have primitive types. (Structured parameters may only be passed by passing a pointer.)

NO_ARGS
No arguments are passed.

saving
Save conventions for registers.

STD_SAVE
Use the standard compiler register saving conventions.

EXTERN_SAVE
Use the standard external routine conventions.

LOCAL_SAVE
Use conventions for calling code with LOCAL_STORAGE.

SUPPORT_SAVE
Use conventions for support procedures.

result
Describes the method used for function results.

EXTERNAL_RESULT
Use external conventions.

STD_RESULT
Use the standard compiler return conventions.

FAST_RESULT
Use the fast return conventions.

NO_RESULT
This procedure cannot return a value.

condition
Indicates whether an exit condition is returned.

NO_COND
There is no exit condition.

COND_VALUE
The condition is a non-negative numeric value less than 100.

Procedure Body Primitives

PROC_BODY (impl_id, proc_form)
Begin the procedure body of the indicated procedure. Procedure bodies may be nested if desired.

MAIN_PROGRAM (imp_id, proc_form)
Identical to PROC_BODY except that the procedure becomes the main program.

PROC_STACK (area, level): loc
"area" is the local data area to be placed on the stack. "level" is the lexic level for the procedure. (The main program has level 0.) The result is the local data location.

PROC_ARGS (area): loc
Returns the permanent location of the procedure argument area. The layout of the area must be specified. Used for STACK_ARGS and LOCAL_ARGS procedures only.

GET_NEXTPAR (mt): loc
Returns the location of the next parameter, which has the specified type.

PROC_ALTENTRY (name, proc_form): loc
Declares the location of an alternate entry point to the procedure. The area describes its parameters, and the result is the location of the parameters.

DISPLAY (int): loc
Returns the location of the data area for the procedure having the indicated lexic level.

RESULT_LOC (mt): loc
Returns the location of the result value. The value will have the indicated type. This primitive may only be used with STD_RETURN procedures.

PROC_RESULT (loc)
Sets the return value for the procedure. If the return class is FAST_RESULT or EXTERNAL_RESULT, this procedure must be followed by PROC_RETURN or PROC_RETCODE.

PROC_RETCODE (int)
Sets the return condition for the procedure. Return conditions must lie in the range 0 to 63. This call implies a call to PROC_RETURN.

PROC_RETURN
Returns from the procedure with the indicated condition. The condition is optional.

PROC_GOEXIT (level
Exit the procedure with a GOTO to a procedure of the specified lexical level, returning control to the specified external label. The stack storage for all called procedures being terminated is released, but no additional clean-up for source structures may be performed.

PROC_END
End of the procedure. All data and parameter list locations generated for this procedure become invalid after this primitive is executed. This primitive does not imply a return.

Procedure Call Primitives

CALL_START (proc_form)
Initializes a call to a procedure of the specified description. If proc_form has FAST_ARGS, no computations may be performed within this call.

CALL_LEVEL (int)
Provides the lexic level of the procedure being called. The outermost level is 0. If this primitive is omitted, the display is not modified.

CALL_ARG_LOC (area): loc
Establish the location of the parameter area. The specified area contains the parameter format. This primitive is only used for procedures whose argument format is STACK_ARG or LOCAL_ARG.

CALL_PASSPARAM (loc)
Pass the specified value as the next parameter. This primitive is used for FAST and EXTERNAL argument forms.

CALL_ROUTINE (loc)
Generate a call to the routine.

CALL_RETCOND : loc
Location of the return condition code, if specified by the called procedure.

CALL_RESULT (mt): loc
Returns the location containing the stored result of the called procedure. For FAST_RESULT and EXTERNAL_RESULT calls, this location must be used "immediately".

CALL_COMPLETE
End of calling sequence.

Code Ordering Primitives

These primitives appear in the generation library. They are only available if the library module is included. This allows simplification of the implementation for simple compilers.

NEW_CODEBLOCK
Create a new code block, which becomes active. A previous active block is stacked for later completion.

CODE_DISJOINT
Declares that the new block will never be included in any block currently on the stack. This implies that the current code block will be a procedure, or will never be generated.

CODE_CONTAINS (cb)
Declares that the active block will eventually contain block "cb". This information is used when assigning

temporaries in the current block, so that they will not duplicate the temporaries of "cb". This primitive should be used before code is qnerated in the active block, or immediately after "cb" (or its containinq block) is removed from the active stack.

CODE_INCLUDE (cb)
Block cb is included at the current position in the code.

END_CODEBLOCK : cb
Terminate the active code block and return it.

Standard Generation Library

The standard qeneration routines have not been selected yet. This section qives an indication of expected entries.

EDIT (loc, strinq): loc
The first location is edited under the specifications found in the string. The resultinq location contains the resulting character strinq, and often a dynamic machine type.

TRANSLATE_TABLE (strinq1, strinq2): loc
Generates a table for character conversion. Each character in strinq1 is to be replaced by the corresponding character of strinq2. Other characters remain unchanqed.

TRANSLATE (loc1, loc2): loc
Translates the character strinq "loc1" usinq the translate table "loc2". The result is the translated strinq.

Semaphores
An additional data type, the "semaphore", may be included for process synchronization. Variables are automatically initialized to a "reset" value. Primitives exist for setting and testing the semaphore, and for resettinq it.

Additional Operators

1. An alternate form of the INT_DYAD and INT_MONAD primitives is provided, with the final operand replaced by a location. Control flows to this label or address location if overflow occurs.

2. Additional bit operators include rotations riqht and left and end out shifts of bit strinqs. These must work for arbitrarily lonq strinqs.

3. Additional supported operators:

SIGN	-1, 0, or +1 for neg, zero, or pos value.
COUNT	Count the 1 bits in a bit item.
PARITY	Parity of a bit string.
*2**	Compose real number from two parts.
*10**	Compose real number--decimal exponent.

DIVIDE (loc1, loc2, loc3, loc4)
Divide integer loc1 by loc2. loc3 and loc4 become the locations of the quotient and remainder.

CODE_COPY (cb): cb
Produce a copy of a code block. Any branches from the block to itself become branches from the copy to itself. Branches from the outside to the original will still go to the original.

Array Operations
Array operations, for pipelined machines could be provided. Table searches and string scans could also be devised.

Implementation Option Control

The compiler can change certain options of the implementation package by using the OPTION primitive.

OPTION (kind, direction)
Set the specified option "ON" or "OFF" depending on the value of "direction".

OBJECT_GEN	Generate the code.
OBJECT_PRINT	Print the generated code.
DEBUG1	First level debug output.
DEBUG2	Second level debug output.
OPTIMIZATION	Perform object optimizations.
LEXIC_DISPLAY	Maintain lexic nesting display.
TEST_OFLO	Test for overflow of integer arithmetic operations.

The Implementation Environment

The following constant values are provided by the implementation:

MININT	Minimum integer
MAXINT	Maximum integer
MAX_PRECISION	Maximum real precision.
MIN_EXPO	Minimum exponent.
MAX_EXPO	Maximum exponent.
REENTRANT	Generated object code is reentrant.
OFLO_TRAPPED	Integer overflows are trapped.
DIV_TRAPPED	Divide exceptions are trapped.
NIL_TRAPPED	Use of the null pointer trapped.

Additional information below could be provided but could lead to implementation differences if used improperly.

STD_CHAR	Standard character set size.
STD_MININT	Usual maximum for integers.
STD_MAXINT	Usual maximum for integers.
WORD_SIZE	Bits per word.
ADDR_UNIT	Bits per addressing unit.
CHAR_SIZE	Bits per character.

Appendix II

The hand-coded example below illustrates the use of some of the primitives. The primitives are shown in a Pascal-like notation. FUNL semantics are normally represented in an internal format; this representation is for illustration only.

The Pascal program below is used for the example. The code generation mimics that generated by the PASCAL/UBC compiler [8]. Some simplifications and changes are made to illustrate the semantics and reduce the length of the example.

```
PROGRAM FACTORIAL_TABLE;
VAR L,I: INTEGER;
    A:    ARRAY [1..20] OF INTEGER;
  FUNCTION  FACT (N: INTEGER): INTEGER;
    BEGIN
      IF N > 0 THEN FACT := FACT (N-1) * N
      ELSE FACT := 1
    END (* FACT *);
BEGIN
  READ (L);
  FOR I := 1 TO N DO A[I] := FACT (I)
END.
```

Generated Code:

```
                                    Standard Prelude Segment

std_proc := PROC_DECL (STACK_STORAGE, STACK_ARGS, STD_SAVE,
                       STD_RESULT, NO_COND);
io_proc := PROC_DECL (LOCAL_STORAGE, FAST_ARGS(4), SUPPORT_SAVE,
                       FAST_RESULT, NO_COND);
int_ty := TY_FIXED (minint, maxint, 2);
int_zero := INT_LOC (0);
MODULE ('*Pascal Library*');
  lib1 := EXTERN_PROC ('READ_INT');

                                    PROGRAM FACTORIAL_TABLE;
MODULE ('*main*');
  a0 := NEW_AREA (false);

                                    VAR L, I: INTEGER;
  l_loc := AREA_CONCAT (a0, int_ty, 10);
  i_loc := AREA_CONCAT (a0, int_ty, 10);
   (NOTE: l_loc and i_loc would be stored in the symtol table.)

                                    A: ARRAY[1..20] OF INTEGER
  arry_ty := TY_REPEAT (int_ty, 20, 10, true);
  a_loc := AREA_CONCAT (a0, arry_ty, 10);

                                    FUNCTION FACT
MODULE ('FACT');
```

```
   arg_area := NEW_AREA (false);
   a1 := NEW_AREA (false);

                                    (N: INTEGER): INTEGER;
   n_loc := AREA_CONCAT (arg_area, int_ty, 10);

                                    BEGIN
   fn1 := MODULE_ENTRY;
PROC_BODY (fn1, std_proc);
   stk := PROC_STACK (a1, 1);
   arg := PROC_ARGS (arg_area);
   TEMP_AREA (a1, stk);

                                    IF N > 0 THEN
   m1 := NEW_LABEL;
   RELATION (>, LOC_DISPLACE (stk, n_loc),
             INT_LOC (0), false, m1);

                                    FACT := FACT (N-1)*N
   CALL_START (std_proc);                    Call FACT.
      CALL_LEVEL (1);
      a2 := CALL_ARG_LOC (arg_area);
      d1 := LOC_DISPLACE (arg, n_loc);       Compute N-1.
      i1 := INT_DYAD (-, d1, INT_LOC (1),
                      minint, maxint, [ ]);
      d2 := LOC_DISPLACE (a2, n_loc);        Store parameter.
      ASSIGN (:=, d2, i1);
      CALL_ROUTINE (PROC_LOC (fn1));
      i2 := CALL_RESULT (int_ty);
   CALL_COMPLETE;
   d3 := LOC_DISPLACE (arg, n_loc);          *N
   i3 := INT_DYAD (*, i2, d3,
                   minint, maxint, [ ]);
   PROC_RESULT (i3);                         Store FACT result

                                    ELSE
   m2 := NEW_LABEL;
   GOTO (m2);
LABEL_DEF (m1);

                                    FACT := 1
   PROC_RESULT (INT_LOC (1));
LABEL_DEF (m2);

                                    END (* FACT *);
   PROC_RETURN;
PROC_END;

                                    BEGIN (* MAIN *)
MODULE ('*MAIN*');
   fn2 := MODULE_ENTRY;
   MAIN_PROGRAM (fn2, std_proc);
   stk := PROC_STACK (a0, 0);
   TEMP_AREA (a0, stk);
```

```
                                        READ (L);
   CALL_START (io_proc);
      d1 := LOC_DISPLACE (stk, l_loc); Pass by reference.
      d2 := LOC_ADDRESS (d1);
      CALL_PASSPARAM (d2);
      CALL_ROUTINE (PROC_LOC (lib1));
   CALL_COMPLETE;

                                        FOR I := 1 TO N DO
   t1 := LOC_TEMP (int_type, true);
   t1 := LOC_USE (t1, 5);
   ASSIGN (:=, t1, INT_LOC (1));          Initial value.
   m3 := NEW_LABEL;
   m4 := NEW_LABEL;
LABEL_DEF (m3);Looping location.
   RELATION (<=, t1, int_zero, false, m4);
   ASSIGN (:=, LOC_DISP (stk, i_loc), t1);
   TEMP_RELEASE (t1);

                                        A[I] := FACT (I);
   t2 := LOC_DISP (stk, i_loc);
   t2 := USE_LOC (t2, 2);                 Optimization

   CALL_START (std_proc);                 Call to FACT.
      CALL_LEVEL (1);
      a3 := CALL_ARG_LOC (arg_area);
      ASSIGN (:=, LOC_DISPLACE (a3, n_loc), t2);  Pass arg.
      CALL_ROUTINE (PROC_LOC (fn1));
      i4 := CALL_RESULT (int_ty);
   CALL_COMPLETE;

                                          Compute A[I]
   i5 := LOC_SUBCOMP (t2, INT_LOC (1), arry_ty,
                      10, true);
   i6 := LOC_DISP (LOC_DISP (stk, a_loc), i5);
   ASSIGN (:=, i6, i4);                    Assign result.

                                       Complete FOR loop.
   TEMP_REUSE (t1);
   ASSIGN (:=, t1, LOC_DISP (stk, i_loc));
   INT_ASSIGN (-:=, t1, LOC_DISP (1), [ ]);
   GOTO (m3);                                Loop to top.

LABEL_DEF (m4);

                                        END.
   PROC_RETURN;
   PROC_END;
OBJECT_GENERATE (false);
```