The

Common Language Encoding Form (Clef) Design Document

Glen E. Weaver, Brendon D. Cahoon, J. Eliot B. Moss, Kathryn S. McKinley, Eric J. Wright, James Burrill

Technical Report 97-58

Version 1.0

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
(413) 545-1249 (fax)

August 1997

Contents

Chapter 1

Introduction

Scale is a compiler *framework* which simplifies the task of developing a compiler for new hardware. Scale is not a compiler in the sense that its objective is not to translate a particular language or to generate object code for a particular machine. Rather it is a *flexible* framework which provides all the components of a compiler along with modules that facilitate the assembling of these components into a complete compiler for a particular language and machine pair.

Scale accepts multiple source languages and generates code for multiple target architectures. Accepting multiple source languages largely consists of supplying a common notation for expressing the semantices of each language. Similarly, to generate object code for different machines, Scale must have a common notation for expressing the capabilities of each target. This common notation for describing hardware also permits Scale to generate object code for multiple machines *simultaneously*. This ability is the base features required by a compiler for heterogeneous systems.

Firgure 1.1 shows the flow of a program through the initial phases of the Scale compiler. In its initial design, Scale supports the translation of imperative and object-oriented programming languages (*i.e.*, C, C++, Modula-3, Fortran 77, and Ada-83). Scale uses the Edison Design Group (EDG) front ends for C and C++ (EDGCPFE) and Fortran (EDGFFE) to handle parsing and semantic error checking. The EDG front ends pass a proprietary AST to EDG2CLEF which uses the *generation interface* to translate it to the common high-level representation form used by Scale. For Modula-3, Scale uses the Digital Equipment Corporation (DEC) Systems Research Center (SRC) Modula-3 front end to generate an AST representation. M32CLEF reads the Modula-3 AST and uses the generation interface to convert it to Scale's high-level representation.

The generation interface is a set of routines for constructing high-level program representations such as ASTs. The interface itself does not determine the structure of the generated interface; however, the Scale's initial design provides only one implementation of the interface: CLEFBULDER. CLEFBUILDER constructs a Clef (Common Language Encoding Form) representation. Clef is a high-level AST representation which

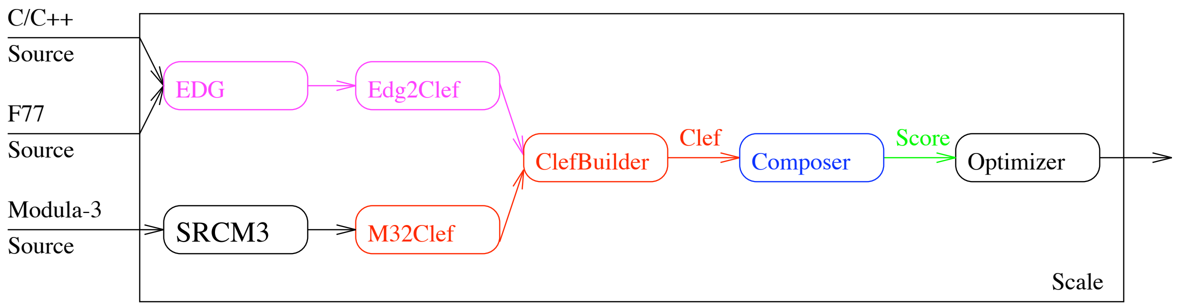


Figure 1.1: The Scale compiler framework.

provides representations for all the features found in source languages supported by Scale. Clef may be used for certain program transformation and annotation. Clef also provides a common representation from which the Composer module can build the Score representation. Score is a lower level representation than Clef and is the form on which Scale performs most of its program transformations.

This book describes components of the Scale compiler framework that relate to the Clef representation. Chapter 2 describes the generation interface and lists all of its routines. Chapter **??** presents the Clef representation node set, and Chapter **??** documents the design of CLEFBUILDER. For information on the EDG and SRC representations see their respective documentation. Score is discussed in more detail in [3].

Chapter 2

Generation Interface

This chapter describes the *generation interface*. The generation interface provides an Application Programming Interface which supports the generation of an intermediate representation. The interface hides the structure of the intermediate representation from the code which is calling the interface. This separation of concerns allows user code (code which calls the interface) to handle source language dependent issues, and client code (which uses the generated representation) to handle language independent or machine dependent issues. We use the term *language parser* to refer to code that handles source language issues, and *user code* to refer to code that directly calls the generation interface. For this interface to generate a particular representation, programmers must supply an *implementation* which provides the bodies for the routines defined in the interface. The code which uses the resulting representation is called a *client*.

We designed the generation interface with the following goals:

- Souce language independent
- Target architecture independent

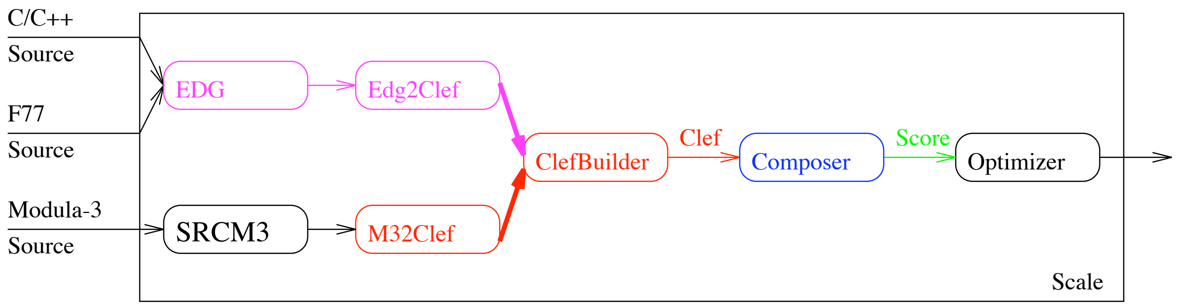


Figure 2.1: Generation interface's role in Scale compiler framework.

User code:	Implementation code:
• Syntactic and semantic error checking	• Memory layout
• Semantic resolution	• Initialization code generation
• Enumeration constant assignment	
• Insertion of explicit type conversions	
• Generation of unspecified brands	

Table 2.1: Division of responsibilities between user code and implementation code.

- Extensible
- Support generation of high-level representations
- Fully describe original program
- Prevent user errors where possible
- Support C, C++, Modula-3, and Fortran 77, and consider Java, Ada, and parallel Fortran variants

Though our design permits the detection of many invalid sequences of routine calls, it cannot detect all such sequences. In particular, the generation interface is not intended to support syntax or semantic error checking. Source language issues must be handled before the generation interface may be used. The interface provides some routines that allow the original program to be fully described, even though the information provided by the routines is not strictly necessary.

To support our goal of source language independence, we have divided responsibility between the language parser and the client as shown in Figure 2.1.

The generation interface derives its language independence by supersetting the capabilities of the languages which we wish to support. Hence, no one language is likely to need all the routines defined in the interface. However, implementations should implement all the routines if possible. If not, they are required to issue appropriate error messages.

Clef Implementation

Our implementation of the generation interface builds the Clef representation. Our implemenation realizes the generation interface as a C++ class. The Clef class hierarchy does not include the interface's class hierarchy. This design point highlights the separation of the interface from the representation. Nevertheless, many of the interface's routines end up being a call to a single constructor in the Clef class hierarchy.

2.1 Supporting Components

This section describes the support that the generation interface requires from an implementation.

2.1.1 Exception Handling

The generation interface requires that the implementation language provide an exception handling mechanism. Individual sections and routines specify exceptions that are unique to them. The exceptions listed below may be returned by any interface routine:

Possible Exceptions:

InternalError The implementation has detected an error which should not be able to occur. This indicates a error in the implementation itself.

InvalidParameters The top most elements on the stack do not match the types of elements required by the current procedure.

InvalidSemantics The called routine is not valid within the context in which it is called. This error is most likely to be generated by attempting to push an element on the stack (by calling the current routine) that is not valid between the current begin/end pair.

OutOfMemory The implementation is not able to allocate sufficient memory to complete the routine.

Unimplemented The implementation has chosen to not implement the called routine.

Implementations may handle exceptions directly or rely on a default error handler to process all errors. The interface provides a routine for setting the default error handler to a user defined procedure.

void **SetErrorHandler** *(void (*er)())*

This routine makes the procedure pointed to by `er` the default error handling procedure. The procedure `er` has no arguments and returns *void*.

2.1.2 The Implementation Stack

We have modeled the generation interface after the stack IR used in the SRC Modula-3 compiler. The two stacks in the stack IR serve very different purposes. The first stack, the implementation stack, allows different implementations of the stack IR to be connected together. Beginning at the top implementation in the implementation stack, each called routine first executes its version of the routine, and then calls the routine of the same name in the next lower implementation. This process allows user code to generate several layers of representation (or perform several functions) with a single call.

In the generation interface, the implementation stack has a limited set of manipulation routines. Unlike the other routines in the interface, the routines which operate on the implementation stack do not recursively execute on lower representations. Moreover, user code can only directly call routines in the top most implementation. Therefore, the implementation stack must be built bottom-up. User code is unlikely to want to change the implementation stack dynamically, so this capability is not supported.

Note for the implementation stack concept to work, the program must be able to distinguish between different implementations of the same routine. Therfore, the generation interface is implemented as an abstract base class, `GenerationInterface`. Implementations of this abstract base class inherit the routine names and provide routine bodies. The implementation stack contains instances of these implementation classes (generally one instance per implementation class).

void **AppendImplementation** *(GenerationInterface gi)*

This routine appends the implementation `gi` at the bottom of the stack.

2.1.3 The Operand Stack

Section 2.1 discusses one use of a stack data structure in the generation interface. This section discusses the other, more important, use of a stack, the operand stack. User code uses the generation interface to generate a high-level representation of a program by calling interface routines. Each routine builds a small piece of the final representation. Between calls, the interface must store these pieces somewhere. The generation interface's solution is to use the operand stack to pass information between routines.

The generation interface uses a stack in order to allow a cleaner division between interface and implementation. The generation interface seeks to catch as many errors as possible, but this requires making some assumptions about the represention being generated (*e.g.*, the representation has an expression node). These assumed nodes (see Figures 2.2–2.6[1]) are the elements placed on the operand stack, according to the interface definition. However, user code is not permitted to directly manipulate the stack (only implementation code may do so), so implementations are free to use whatever nodes they want on the stack, as long as error

[1]These figures currently show more detail than the interface needs to know. This situation will be fixed when Chapter **??** is written.

checking is performed according to the type information used in this document. This frees implementations to use a different class hierarchy than the one shown (or no hierarchy at all). Without a stack (or some similar entity), the interface would have to return intermediate results to the user code, and so the an implementation would have to create objects with the type name specified for each routine.

Therefore, the generation interface uses an abstract stack machine as its execution model. Before programmers invoke a routine, they must ensure that the top of the stack contains the necessary arguments. A **StackPop** routine is provided for removing unwanted elements from the type of the stack. Adding elements to the top of the stack requires calling appropriate interface routines.

void **StackPop** *()* $\succ \lfloor \mathsf{Node} \rfloor \Rightarrow \lfloor \rfloor$

This routine removes an element from the top of stack. Since stack elements are not accessible to user code, no value is returned and the stack element is simply no longer on the stack.

Each interface routine can accept input both from its argument list and from the operand stack. Routines can return results through both their return value and the operand stack. Most Interface routines obtain input from the operand stack and push a single return value on the stack. Only a few routines use a *return* statement to return a value. Argument lists are used to pass leaf information to the interface. Throughout this chapter, interface routines are shown in the following form:

return-type **routine-name** *(routine-arguments)* $\succ \lfloor \mathsf{arguments\text{-}on\text{-}stack} \rfloor \Rightarrow \lfloor \mathsf{type\text{-}of\text{-}result\text{-}on\text{-}stack} \rfloor$

In arguments-on-stack, the rightmost stack element is the top of stack. Though this chapter gives names to the arguments, these names are only for clarity. The $*$ symbol, which is usually found in an end routine, indicates that a routine removes an unknown number of elements from the stack. All the elements specified in arguments-on-stack are replaced by an element of type type-of-result-on-stack. Currently, no routine pushes more than one value on the operand stack, but this is permissible.

Not all invocations of a routine will need all the routine's parameters. To handle the case when the unnecessary parameters are expressions, the generation interface defines a special value, NoExpression, to represent no value.

Unbounded Input

When a construct allows a dynamic number of repeated items (*e.g.*, fields in a record, statements in a block, arguments in a list), the interface uses a begin/end pair to bracket the repeated elements. For example, calls to **FormalsBegin** and **FormalsEnd** surround calls to **DeclFormal**. The begin routine does not have a visible effect on the operand stack, but the end routine removes all the intervening stack elements, forms a list of these elements, and pushes a new stack element on the top of the stack which represents this list. Unless otherwise stated, user code may build an empty list by calling the end routine when the stack is identical to when the begin routine was called.

Figure 2.2: Top level of class hierarchy model used in design of generation interface.

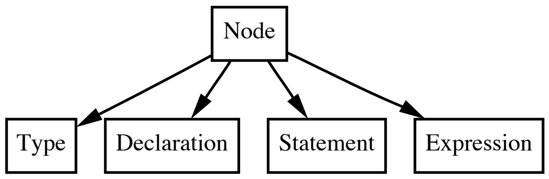
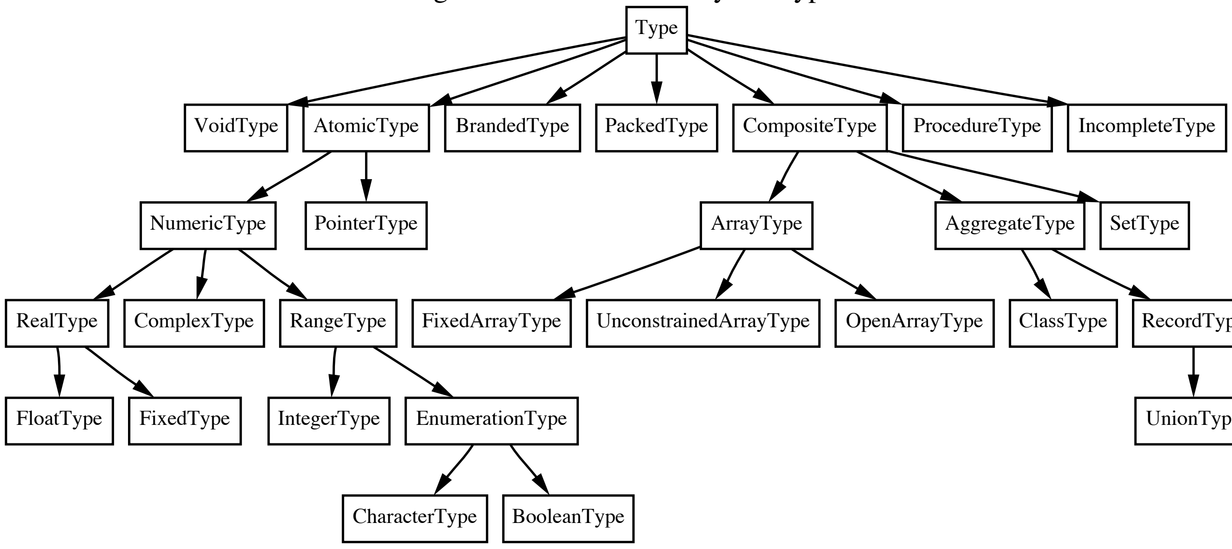


Figure 2.3: Class hierarchy for Types.



The generation interface design requires unbounded lists to be completely pushed on the operand stack before any of the list elements can be removed. For some lists (*e.g.*, statements in a procedure or statements in a file), this approach appears wasteful of space. However, implementations are most likely pushing the actual program representation on the stack, and this representation must exist regardless of whether or not it is on the stack. So, no extra space may be required. If space is still a concern, then implementations may choose to devise a scheme by which appropriate lists are incrementally compressed on the operand stack, as long as the semantics of the begin/end pair are preserved.

Implementation suggestion

Implementations may choose to use an auxillary stack, the list stack, for begin/end pairs. Each time a begin occurs, an element is pushed on the list stack that records the top of stack when the begin routine was called. Therefore, when the corresponding end routine is called, all the elements of the operand stack down to the recorded location may be removed and formed into a list.

Figure 2.4: Class hierarchy for Declarations.

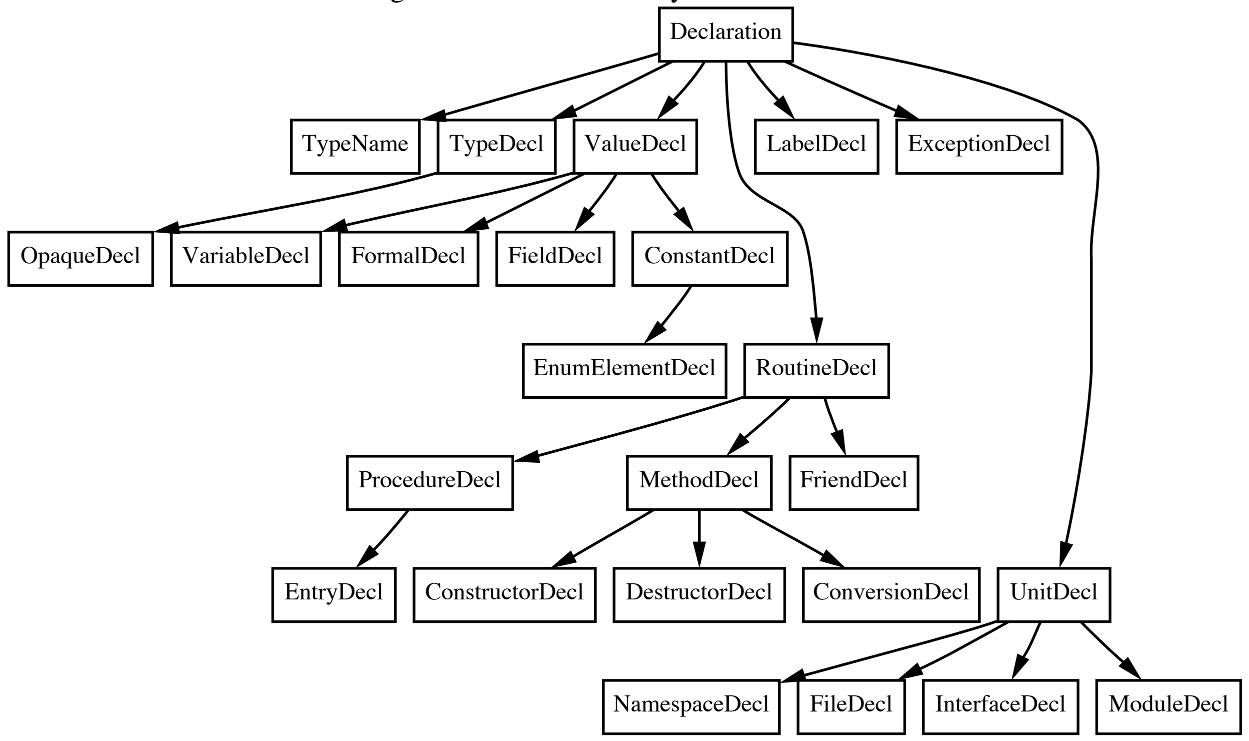


Figure 2.5: Class hierarchy for Statements.

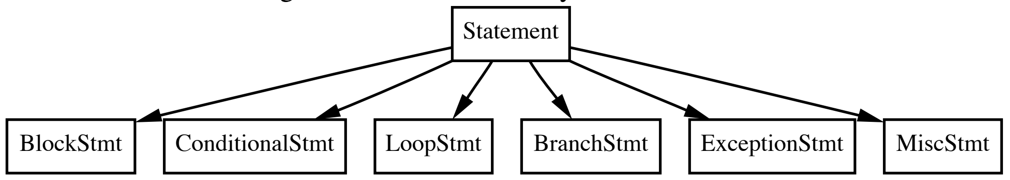
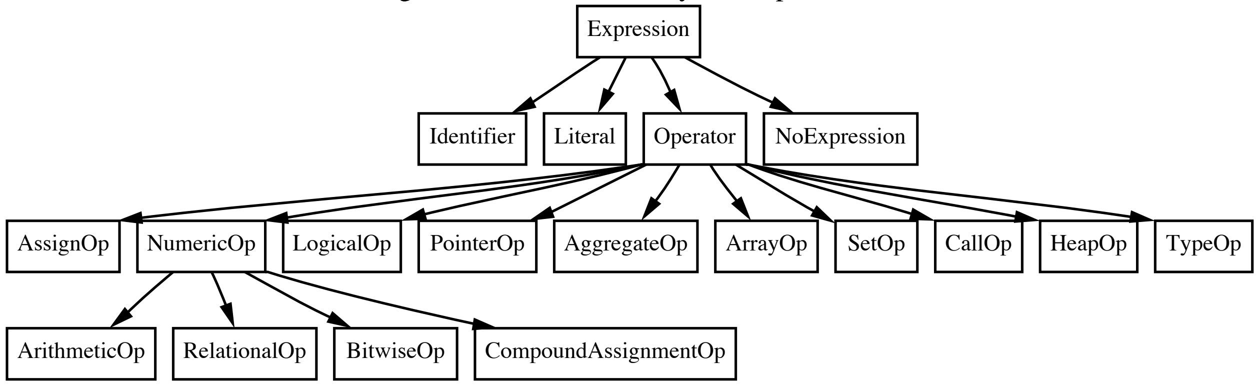


Figure 2.6: Class hierarchy for Expressions.



Implementation Suggestion
We strongly recommend creating two global variables to aid in debugging. These variables are intended to help catch errors with begin/end pairs. One variable, `listType`, holds an indication of the type of an arbitrary element popped off the stack by an end routine. The other variable, `listLength`, holds a count of how many elements were popped. These variables' values are valid until the next end routine. The initial values of these variables are `NotAType` and zero, respectively.

2.1.4 The Symbol and Type Tables

The generation interface assumes that implementations maintain two tables related to declarations: a symbol table and a type table. A symbol table maintains correspondences between programmer introduced names (and some language defined names) and their value. A type table maintains a representation of each type's structure. Since the language parser is responsible for semantic resolution, it already has all the information contained in these two tables, and indeed most likely has its own version of these tables. However, to maintain its independence from an particular language parser, the generation interface cannot simply accept the parser's tables directly. Instead, these two tables must be built by an implemenation by repeated calls to type constructor (see Section 2.3) and declaration routines (see Section 2.4).

Since the interface does not know anything about the user code's tables, any reference to an identifier or type must refer to entries in the interface implementation's tables. Hence, user code needs some way of referring to an entry in an interface implementation's tables. Appropriate interface routines pass back an entry identifier. Whenever the implementation makes an entry in one of its tables, it passes back either a *NameID* which refers to an entry in the symbol table, or a *TypeID* which refers to an entry in the type table. User code must retain a mapping of *NameID*s and *TypeID*s to entries in its own tables. Whenever an identifier is used, the user code can than translate from the entry in its symbol table to the *NameID* or *TypeID* which identifies the entry in the implementation's tables. Implementations should provide `NoName` and `NoType` to represent that the corresponding item does not exist.

2.1.4.1 The Symbol Table

This section describes the function of the symbol table in more detail and lists the routines used to control and access the symbol table. A symbol is a name. Most symbols are created by the programmer (*e.g.*, variable names and procedure names). Some names are introduced by the compiler either for compiler generated entities such as compiler generated labels or as compiler generated names for anonymous entities.

User code also generates a few names with calles to the primitive type declaration routines[2].

Many program elements have names: variables, procedures, exceptions (in some source languages), etc. The generation interface refers to any nameable program element as an *entity*. Every entity in a program must have a corresponding *entry* in the symbol table, which maps the name to its value (where a value can have several parts). The value of an entity depends on what kind of entity it is. A value of a type entity is a type; the value of a procedure entity is its signature and body. In order for user code to refer to a particular entity (or equivalently, entry), interface routines return a *NameID* whenever a new entry is created in the symbol table. Each implementation is free to define *NameID*'s structure, but it is conceptually an index into the implementation's symbol table.

After an entity is declared, user code always uses the *NameID* to refer to the entity. By not using identifiers, the generation interface avoids having to understand the source language's scoping and name conflict resolution rules. In other words, the symbol table does not limit the number of times an identifier may appear in a single scope. On the other hand, the generation interface cannot provide name lookup.

2.1.4.1.1 Specifying Scopes Client code needs to understand scoping, in order to transform code and generate object code. Unfortunately, different source languages may have slightly different scoping rules. The generation interface handles this issue by requiring user code to explicitly mark where a scope begins and ends by using the **ScopeBegin** and **ScopeEnd** routines. These routines do not visibly affect the contents of the stack.

Since user code is responsible for semantically resolving the program before using the generation interface to build an new intermediate representation, the interface can use a generic block structured symbol table. Declarations may appear anywhere within a scope, and the declared entity is useable anywhere from the point of declaration to the end of the enclosing scope. In addition, entities in enclosing scopes are accessible from an inner scope (note that *visibility* is a language parser issue).

Note that namespaces are essentially named scopes (see Section 2.4.6.1).

void **ScopeBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **ScopeEnd** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

2.1.4.1.2 Entering Entities into the Symbol Table The generation interface specifies that all programmer defined entities (*e.g.*, types, labels, procedures) are stored in the symbol table. Many entities have names, but some may not. Entities are entered into the symbol table either by *declarations* or *specifications*. A declaration defines an entities; whereas, a specification provides partial information about an entity.

[2]Primitive types would not have to be named in order to representat a program. However, it may be useful in debugging and quite easy to provide.

2.1.4.1.3 Entity References In addition to the notion of declarations, the generation interface maintains the idea of an entity *reference*. An entity reference represents an entity independently of its name. Uses of an entity in a program are represented by its reference node, rather than its declaration. In most cases, a reference is indistinguishable from a declaration, but not always. For example consider two variables which are declared to be aliases of each other. They have separate declarations but a single reference. Because of the similarity between references and declarations, their class hierarchies are identical and only the declaration hierarchy is shown (see Figure 2.4). *NameID* maps into declarations, but a routine is provided for mapping from *NameID* to references.

2.1.4.1.4 Accessing the Symbol Table To avoid having to know the details of how a particular language manages its symbol table, the generation interface relies on the requirement that user code semantically resolves identifiers before using the generation interface. Hence, user code knows to which declaration each use of an identifier refers (*i.e.*, user code does its own name lookup). User code passes this knowledge to the generation interface by using a *NameID*. It is a simple type which uniquely identifies a declaration. Every declaration routine passes a *NameID* back. User code is responsible for maintaining a mapping between entries in its symbol table and NameIDs.

User code is only permitted to lookup entities which it has already entered into the symbol table. Hence, few errors should occur. Nevertheless, implementations ought to check the validity of NameIDs.

void **LookupDecl** *(NameID name)* $\succ \lfloor\rfloor \Rightarrow \lfloor$Declaration$\rfloor$

void **LookupRef** *(NameID name)* $\succ \lfloor\rfloor \Rightarrow \lfloor$Reference$\rfloor$

void **LookupType** *(TypeID type)* $\succ \lfloor\rfloor \Rightarrow \lfloor$Type$\rfloor$

void **LookupTypeDecl** *(NameID type)* $\succ \lfloor\rfloor \Rightarrow \lfloor$Type$\rfloor$

2.1.4.2 Type Table

The most complicated values in the symbol table are type descriptions. In fact, type information is maintained in a separate table. The type table has its own index, *TypeID*. If the only use of the type table were to describe types, the design of the table would be straightforward. However, the generation interface must also capture sufficient semantic information for client code to perform type equivalence checks. Determining type equivalence is difficult because each source language defines its own type equivalence rules. The generation interface's approach is to include as much information as possible in the type table, and then define a general routine which traverses the type table using a subset of that information to compare two types.

Adding Information

Besides the normal information required by type constructors, the generation interface places two other kinds of information in the type table. The first is type names, which are useful for type equivalence. To simplify the specification of type names across the interface, the generation interface takes the name from the type declaration routine and inserts it into the type table. The name is added via a nameBrand node, which is distinct from a UserBrand node. Note that a NameBrand node may include multiple names to handle types with multiple names. The other kind of information is language defined type matching rules. These rules are captured at the compilation unit level and include such information as whether or not the source language uses field order to distinguish record types.

Type Equivalence

A general routine is used to perform all type equivalence checks. This routine accepts as input an indication of which of the possible features that distinguish types should be used for the current comparison.

Multiple Type Names

Source languages generally allow programmers to declare types, which is naming a type. Source languages generally allow multiple names to be associated with the same type. For example in the following C code, both `gew` and `GEW` are names for the same type:

```
 typedef struct gew {
int a; } GEW;
```

The generation interface does not guarantee that two equivalent types will have the same *TypeID*. Hence, user code cannot use *TypeID*s to determine type equivalence, but user code should already have its own way of finding type equivalence. Implementations may wish to provide a way for client code to compute type equivalence.

Decorated Types

One reason for not guaranteeing a unique *TypeID* for each type is because types can be decorated with attributes. Section 2.3.2.11 discusses the different type attributes in more detail. Type attributes are associated with an individual type node, and any type node may be annotated though the generation interface does not guarantee that all attributes make sense. Attributes do not affect the type of an object, but do represent information about the object that must be maintained. Hence, the *TypeID* carries the attribute information as well as the type.

Recursive Types

One of the hardest aspects of describing types is describing recursive types. To handle this situation, the generation interface provides **Incomplete*** routines for types which may refer to instances of themselves (*i.e.*, aggregate types). Within the definition of the aggregate type, recursive references may refer to the incomplete type. After the definition of the aggregate type is complete, user code should call the **CompleteType** routine to establish the relationship between the incomplete and complete types.

Branding

The generation interface supports a branded type which is used to distinguish two types which are otherwise type equivalent. Branding is a way to introduce a name into the type table, and thereby obtain name equivalence in a structural equivalence framework. Branding is in fact the only way of introducing a name into the type table.[3]

Separate Compilation

Separate compilation complicates type matching because not all type information is available for each compilation. For separate compilation, C/C++ uses name equivalence between files. Hence, two types are considered equivalent if they have the same name, regardless of their actual type declaration. This unfortuante rule works fine if the two files never exchange data of this type, otherwise the programmer has made a non-detectable error. To support this mis-feature of C/C++, only the brand (not the type structure) is used to determine the equivalence of branded types.

2.1.5 Identifiers

All practical programming languages use identifiers, user selected strings, to represent program entities such as variables, types, and labels. The generation interface does not use identifiers to reference these entities, since many programming languages allow the same identifier to have different meanings in different contexts. However, the generation interface does permit user code to pass identifiers through to client code.

The generation interface represents identifiers as strings. A null string is interpreted as representing that the programmer did not specify an identifier. Among other possible uses, a null string can be used to indicate anonymous declarations.

2.1.6 Stable Storage

Once the user has finished generating a representation with the generation interface, he or she is likely to want to save the representation to some form of stable storage. The generation interface provides the following routine for performing this function. The interpretation of *key* is implementation dependent, and

[3]A name is specified when primitive types are declared, and these are recorded in the type table. However, these names are source language defined (*e.g.*, int) rather than user or compiler defined, and they are not used to discriminate types. Indeed, their only purpose is debugging support.

the interface does not provide a mechanism for reading back in the representation. This functionality should be provided by other code which manipulates the generated representation.

bool **WriteRepresentation** *(String key)* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

This function writes the representation out to an implementation defined stable storage. The *key* argument supplies an identifier, most likely a file name, for the written representation. When multiple implementations are stacked, calling this routine will write out all the different representations. Therefore, we suggest that each implementation modify the key somehow to uniquely identify its output (*e.g.*, a file name extension). The return value indicates success or failure, with the value from multiple implementations combined with a logical or function.

2.2 Source Language Specific Comments

This section records suggestions of how to translate specific source languages using the generation interface. The generation interface directly reflects the syntax and semantics of C/C++ and Modula-3, so these languages are likely to have fewer comments.

2.2.1 Modula-3

- inc/dec

  Modula-3's *inc* and *dec* statements are not directly supported in the generation interface. User code should represent them as **eval**ed expressions.

- assignment

  In Modula-3 assignment is a statement, but the generation interface provides only an expression form of assignment. User code should represent an assignment statement as an **eval**ed assignment expression.

- Refany and Address

  Refany is represented as an unattributed pointer to void type. Address is represented as a pointer to void type with the untraced attribute.

- Safe modules

  The language parser and user code is responsible for checking whether or not modules are safe. This information is not passed through the generation interface.

2.2.2 Fortran 77

This section consists of a list of issues and our recommendations for handling them.

- Generic functions

Fortran 77 allows a small amount of function overloading by permitting library writers to create generic functions. The identification of generic functions is considered a front end issue, so the generation interface only provides specific function names.

- Declarations

  Declarations in Fortran 77 are distributed across several statements rather than collected together as expected by the generation interface. Basically, user code is responsible for collecting the distributed information together and making the appropriate generation interface call. The following lists some of the declaration statements that must be transformed:

  - Implicit

- Strings

  User code should transform Fortran 77 strings into character arrays.

- Intrinsic Functions

  Some intrinsic functions are mapped to primitive operators, while others will have to be implemented in a library. See Tables 2.3–2.4. [¡

- At least once *do* loops.

  Some older version of Fortran have *at least once* semantics for their *do* loops. The generation interface does not directly provide these semantics, so user code will have to build equivalent code from the available routines.

2.2.3 C++

- Declarations in conditions and for loop initialization

  C++ allows programmers to declare variables in conditions (*i.e.*, the test in loops and in conditional statements). The value of the variable comes from its initialization.

2.3 Specifying Types

This section describes the routines which user code can use to pass type information to clients (see Table 2.2. The routines for declaring type names can be found in Section 2.4. Type information is stored in the type table (see Section 2.1).

Type Constructor Correspondences			
Method Name	**C++**	**Modula-3**	**Fortran 77**
TypePrimitiveCharacter	char	char	character
TypePrimitiveInteger	int	integer	integer
TypePrimitiveFixedPoint	*N/A*	*N/A*	*N/A*
TypePrimitiveReal	float/double	real/longreal/extended	real
TypePrimitiveVoid	void	null	*N/A*
TypePrimitiveBoolean	bool	boolean	logical
TypeArrayFixed	[]	array-of	$(), *$
TypeArrayOpen	*N/A*	array-of	*N/A*
TypeArrayUnconstrained	*N/A*	*N/A*	*N/A*
TypeEnum begin/end	enum	$\{\}$	*N/A*
TypeRecord begin/end	struct	record	*N/A*
TypeIncompleteRecord	struct	*N/A*	*N/A*
TypeUnion begin/end	union	*N/A*	equivalence
TypeIncompleteUnion	union	*N/A*	*N/A*
TypeClass begin/end	class	object	*N/A*
TypeIncompleteClass	class	*N/A*	*N/A*
TypePointer	$*$	*N/A*	*N/A*
TypeIndirect	&	ref	*N/A*
TypeOffset	::*	*N/A*	*N/A*
TypeProcedureType	()	procedure	*N/A*
TypeSet	*N/A*	set-of	*N/A*
TypeRange	*N/A*	[..]	*N/A*
TypeBrand	*N/A*	branded	*N/A*
TypePacked	:	bits-for	*N/A*

Table 2.2: Correspondence between generation interface routine names and language type constructors.

Different languages employ slightly different versions of derived types and subtypes. A derived type has all the associated operators of the parent type, but is not type compatible with other types derived from the same parent. A subtype is much like a derived type, except that a subtype may have additional fields and operators to those of the parent type. The generation interface provides type attributes and branding for representing derived types. Subtyping is only available for classes.

2.3.1 Primitive Type Constructors

This section describes routines for declaring primitive language types. In term of type representation, these routines specify the leaves of type DAGs. Different languages use different names for the same primitive type, and some languages do not completely specify what their primitive types mean. The generation interface requires user code to indicate what name and format their source code needs. The type name is only useful for debugging.[4] These routines may be called multiple times with different values to create different primitive types (*e.g.*, C's *short*, *int*, and *long* are all integers).

The generation interface offers two mechanisms for defining primitive types. User code may either specify the type with constraints on the number of bits used in the representation, or the user code may use purely symbolic names for the size of the types. Symbolic names will be mapped to the natural size supported by the target architecture. For symbolic types, their size is specified with a symbolic value from the following enumerated list:

```
enum SymbolicSize {cVeryShort, cShort, cNormal, cLong, cVeryLong};
```

2.3.1.1 Sized Primitive Types

TypeID **TypePrimitiveCharacter** *(Identifier name, CharacterFormat cf)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

Ideally, the language should not specify a particular character format. However, many C programs rely on the ANSI character set, and Java prescribes the use of Unicode. Hence, user code may use this routine to specify the format (and implicitly the size) of the character set. If the language does not require a particular format it may specify `Any`. The generation interface assumes a default of `Any`.

The generation interface provides the following enumeration:

```
enum CharacterFormat {cAny, cAnsi, cEbcdic, cUnicode};
```

[4]The name does not distinguish primitive types. Hence, implementations should be able to handle having the same primitive type with two different names. Without a name, a debugger could not identify (to the user) the primitive types.

TypeID **TypePrimitiveInteger** *(Identifier name, int minBitSize, IntegerRepresentation rep)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

This routine creates a primitive integer type in the symbol table.

Parameters:

`name` The name of the type.

`minBitSize` The minimum number of bits required to represent this type.

`rep` Machine representation for this integer type. Its value comes from the following enumerated type.

```
enum IntegerRepresentation {cUnsigned, cTwosComplement};
```

TypeID **TypePrimitiveSymbolicInteger** *(Identifier name, SymbolicSize size, IntegerRepresentation rep)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

This routine creates a primitive integer type in the symbol table.

Parameters:

`name` The name of the type.

`size` The symbolic size of the integer type. Our expectation is that for current machines, `cVeryShort` equals eight bits, `cShort` equals sixteen bits, `cNormal` equals 32 bits, `cLong` equals 64 bits, and `cVeryLong` equals 128 bits.

`rep` Machine representation for this integer type. Its value comes from the following enumerated type.

```
enum IntegerRepresentation {cUnsigned, cTwosComplement};
```

TypeID **TypePrimitiveFixedPoint** *(Identifier name, int minBitSize, int minScaleBits)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

This routine creates a primitive fixed point type in the symbol table.

Parameters:

`name` The name of the type.

`minBitSize` The minimum number of bits required to represent this type.

`minScaleBits` Minimum number of bits by which to scale the representation. This parameter may be negative. `minBitSize` includes `minScaleBits`.

TypeID **TypePrimitiveReal** *(Identifier name, int minBitSize)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

Parameters:

`name` The name of the type.

`minBitSize` The minimum number of bits required to represent this type.

Note that the format of real numbers is considered a back end issue.

TypeID **TypePrimitiveSymbolicReal** *(Identifier name, SymbolicSize size)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

Parameters:

`name` The name of the type.

`size` The symbolic size of the real type.

Note that the format of real numbers is considered a back end issue.

TypeID **TypePrimitiveComplex** *(Identifier name, int realMinBitSize, int imaginaryMinBitSize)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

Parameters:

`name` The name of the type.

`realMinBitSize` The minimum number of bits required to represent the real part of this type.

`imaginaryMinBitSize` The minimum number of bits required to represent the imaginary part of this type.

Note that the format of real numbers is considered a back end issue.

TypeID **TypePrimitiveVoid** *(Identifier name)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

Void is the null type.

TypeID **TypePrimitiveBoolean** *(Identifier name)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

Boolean values are defined in Section 2.6.5.

2.3.2 Type constructors

Languages such as C and C++ which provide weak type naming capabilities encourage programmers to use type constructors when describing the type of non-type entities (*e.g.*, variables) instead of type names. Implementations may choose to handle such circumstances by internally creating an anonymous type for the entity.

2.3.2.1 Array Type Constructors

These routines support the construction of arrays. In some languages, arrays carry knowledge of their length, and in other languages (*e.g.*, C and C++) they do not. Though knowing their length does impact data layout, it does not affect type equivalence. Hence, we mark each array as to whether or not it must carry a length:

TypeID **TypeArrayFixed** *(bool lengthField)* $\succ$ ⌊RangeTypes indexType, Type elementType⌋ $\Rightarrow$ ⌊Type⌋

This routine constructs a fixed length array.

Parameters:

`indexType` Indicates the type of the index expression (a list of ranges - for a single dimension array it will be a list containing one range).

`elementType` Indicates the type of the array elements.

TypeID **TypeArrayUnconstrained** *(bool lengthField)* $\succ$ ⌊RangeTypes indexType, Type elementType⌋ $\Rightarrow$ ⌊Type⌋

This routine represents an unconstrained array type, as found in Ada. An unconstrained array is an incomplete type, because it lacks an index range. Therefore, the type of indexType should not include range information (*i.e.*, the bounds should be NoBounds).

TypeID **TypeArrayOpen** *(bool lengthField)* $\succ$ ⌊Type elementType⌋ $\Rightarrow$ ⌊Type⌋

This routine constructs an open array. The size of an open array is determined at runtime but cannot change once set. An open array acts like an unconstrained array, where the only unknown entity is the maximum index value.

2.3.2.2 Enumeration Type Constructors

The generation interface assumes that the front end has assigned a value to each enumeration item. This approach frees implemenations of the interface from concern about source language specific vagaries in enumeration element assigment.

NameID **DeclEnumElement** *(Identifier id)* $\succ$ $\lfloor$Expression e$\rfloor$ $\Rightarrow$ $\lfloor$EnumElementDecl$\rfloor$

void **TypeEnumBegin** *()* $\succ$ $\lfloor\rfloor$ $\Rightarrow$ $\lfloor\rfloor$

TypeID **TypeEnumEnd** *(Identifier id)* $\succ$ $\lfloor * \rfloor$ $\Rightarrow$ $\lfloor$Type$\rfloor$

At the point that the **TypeEnumEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **TypeEnumBegin** call) should contain only EnumElementDecl nodes.

2.3.2.3 Incomplete Type Constructors

The generation interface requires that all entities be defined before being used. This restriction keeps the interface free of language specific name resolution rules. Unfortunately for recursive types, this restriction implies that user code needs a mechanism for specifying a type before definining it completely. The generation interface uses a mechanism similar to forward declarations found in many languages.

The generation interface allows the creation of *incomplete types*. An incomplete type does not carry any additional type information, and therefore may be used anywhere a type is valid. User code should build recursive type structures with recursive references pointing to the incomplete type. After the recursive type is completely built, user code should use **CompleteType** to allow implementations of the interface to patch up its representation to reflect the true recursive structure. Incomplete types are only to aid in conveying the actual type structure through the interface. Hence, user code must associate complete type with each incomplete type.

For languages such as Modula-3 which do not require forward declarations for mutually recursive types, user code must generate incomplete types.

Incomplete types can be created at any time. Hence, user code can begin building an aggregate structure and generate an incomplete type only if the aggregate type turns out to be recursive. However, the user code must be careful to clean up the Type node on the stack.

TypeID **TypeIncompleteType** *(Identifier name)* $\succ$ $\lfloor\rfloor$ $\Rightarrow$ $\lfloor$Type$\rfloor$

TypeID **TypeCompleteType** *(TypeID tic, TypeID tc)* $\succ$ $\lfloor\rfloor$ $\Rightarrow$ $\lfloor$Type$\rfloor$

This routine completes the declaration of an incomplete type. It informs the implementation that incomplete type `tic` is really the completed type `tc`. The Type node left on the stack corresponds to `tc`.

Parameters:

`tic` *TypeID* identifying incomplete type.

`tc` *TypeID* identifying complete type.

2.3.2.4 Aggregate Type Constructors

void **TypeRecordBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

TypeID **TypeRecordEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor\text{Type}\rfloor$

At the point that the **TypeRecordEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **TypeRecordBegin** call) should contain only FieldDecl nodes. In C++, a class struct is represented as a ClassType.

void **TypeUnionBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

TypeID **TypeUnionEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor\text{Type}\rfloor$

At the point that the **TypeUnionEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **TypeUnionBegin** call) should contain only FieldDecl nodes. In C++, a class union is represented as a ClassType

void **Superclass** *(NameID class, AccessSpecifier as)* $\succ \lfloor\rfloor \Rightarrow \lfloor\text{SuperClass}\rfloor$

Parameters:

`class` Indicates the super class.

`as` Indicates the access specifier (as defined by C++). For other languages, a suitable value for `as` should be selected. The value for `as` comes from the following:

```
enum AccessSpecifier {cPrivateAccess, cProtectedAccess,
                                cPublicAccess};
```

void **SuperclassBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **SuperclassEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor\text{SuperClasses}\rfloor$

At the point that the **SuperclassEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **SuperclassBegin** call) should contain only SuperClass nodes.

void **SingleSuperclass** *(NameID class, AccessSpecifier as)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{SuperClasses} \rfloor$

This routine is a shortcut for cases when only a single super class exists (as with single inheritance).

void **TypeClassBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

TypeID **TypeClassEnd** *()* $\succ \lfloor \mathsf{SuperClasses}, * \rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

The generation interface uses its own variation of C++ and Modula-3 terminology when discussing objects. An *object* is an instance of a *class* datatype. The components of a class are called *members*, which may be either data *fields*, *routines*, conversion functions, constructors, and destructors.

At the point that the **TypeClassEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **TypeClassBegin** call) should contain only Declaration nodes.

2.3.2.5 Pointer Type Constructors

The routines in this section allow the construction of pointer types.

TypeID **TypePointer** *()* $\succ \lfloor \mathsf{Type\ t} \rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

This routine constructs a pointer type which must be explicitly dereferenced.

TypeID **TypeIndirect** *()* $\succ \lfloor \mathsf{Type\ t} \rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

This routine describes automatically dereferenced pointers. In Modula-3 these pointers are used in implementing *with* aliases, *var* formals, etc.

TypeID **TypeOffset** *(NameID aggregate)* $\succ \lfloor \mathsf{Type\ t} \rfloor \Rightarrow \lfloor \mathsf{Type} \rfloor$

This routine represents a pointer-to-member as found in C++. A pointer-to-member is an offset to a member of *aggregate*.

2.3.2.6 Procedure Type Constructors

A method is a procedure who is a member.

Parameters

void **FormalsBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **FormalsEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor \textsf{Formals} \rfloor$

At the point that the **FormalsEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **FormalsBegin** call) should contain only FormalDecl nodes.

Exceptions

void **RaiseException** *(NameID exception)* $\succ \lfloor\rfloor \Rightarrow \lfloor \textsf{Raise} \rfloor$

This routine represents a Modula-3 style exception, when specifying a procedure's throw list.

void **RaiseType** *(TypeID type)* $\succ \lfloor\rfloor \Rightarrow \lfloor \textsf{Raise} \rfloor$

This routine represents a C++ style exception, when specifying a procedure's throw list.

void **RaisesAny** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor \textsf{Raise} \rfloor$

This routine indicates that the associated procedure can raise any exception. By default, C++ functions may raise any exception; Modula-3 procedures must explicitly indicate that they can generate any exception.

void **RaisesNone** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor \textsf{Raises} \rfloor$

This routine indicates that the associated procedure cannot raise any exceptions. By default, Modula-3 procedures cannot raise any exceptions; C++ procedures must explicitly indicate that they cannot raise exceptions. Using this routine is equivalent to specifying **RaisesBegin/RaisesEnd** without any Raise nodes in between.

void **RaisesBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **RaisesEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor \textsf{Raises} \rfloor$

At the point that the **RaisesEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **RaisesBegin** call) should contain only Raise nodes.

void **Signature** *()* $\succ \lfloor \textsf{Formals f, Type ret, Raises r} \rfloor \Rightarrow \lfloor \textsf{Signature} \rfloor$

This function defines a routine signature. All signatures must indicate what exceptions it can raise. For those languages which do not support exceptions, they should simply call **RaiseNone**.

TypeID **TypeProcedure** *()* $\succ \lfloor \textsf{Signature s} \rfloor \Rightarrow \lfloor \textsf{Type} \rfloor$

2.3.2.7 Set Type Constructors

The routines in this section permit the construction of set types.

TypeID **TypeSet** *()* $\succ \lfloor$Type type$\rfloor \Rightarrow \lfloor$Type$\rfloor$

2.3.2.8 Range Type Constructors

Bounds

Bounds specify a minimum and maximum value. Bounds are primarily used for specifying the minimum and maximum for range types. Support for defining bounds varies between languages, so the minimum and maximum values are arbitrary expressions. We also use ranges to represent array indicies. For languages (such as Ada) that support true multi-dimensional arrays, ranges may be chained together since a range type contains a single bound. However, the common case for C/C++ and Modula-3 is to use a single range.

void **Bound** *()* $\succ \lfloor$Expression min, Expression max$\rfloor \Rightarrow \lfloor$Bound$\rfloor$

This routine creates a single bounds which can be chained together with a begin/end pair.

void **BoundsBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **BoundsEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor$Bounds$\rfloor$

At the point that the **BoundsEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **BoundsBegin** call) should contain only Bound nodes.

void **Bounds** *(Expression min, Expression max)* $\succ \lfloor\rfloor \Rightarrow \lfloor$Bounds$\rfloor$

This routine is a short cut for specifying a single dimensional bounds specification.

void **Nobounds** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor$Bound$\rfloor$

This routine pushes onto the stack a special Bounds that connotes that no bounds have been specified. This value may be used to construct arrays without bounds.

TypeID **TypeRange** *()* $\succ \lfloor$Type basetype, Bound b$\rfloor \Rightarrow \lfloor$Type$\rfloor$

This routine creates a range type with a single bound.

void **RangeBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **RangeEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor$RangeTypes$\rfloor$

At the point that the **RangeTypeEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **RangeTypeBegin** call) should contain only RangeType nodes.

2.3.2.9 Branded Type Constructors

Branded type constructors are unique in that though they do indeed introduce a new type, they do not change the structure of the type. Branding is useful in languages with structural equivalence (*e.g.*, Modula-3).

TypeID **TypeBrand** *()* $\succ \lfloor$Type t, Expression b$\rfloor \Rightarrow \lfloor$Type$\rfloor$

This routine creates a new type which is structurally identical to type `t`.

2.3.2.10 Packed Type Constructors

The generation interface treats packed types a unique types. For C/C++ this distinction between types would not be necessary. However, other languages such as Modula-3 clearly distinguish between a base type and its packed version. This distinction implies that user code must insert explicit conversions between the base type and packed type.

TypeID **TypePacked** *()* $\succ \lfloor$Type basetype, Expression bitSize$\rfloor \Rightarrow \lfloor$Type$\rfloor$

This routine builds a new type which compresses type `basetype` into `bitSize` bits.

2.3.2.11 Alias Type Constructors

C++ provides a type constructor for defining alias. C++ calls these aliases *references*. An alias does not have memory space of its own, but rather refers to another entity's memory.

I've considered representing references as an alias declaration which looks better, but return types can be references. I've also considered treating references as indirect pointers. However, this approach might hurt alias analysis.

TypeID **TypeAlias** *()* $\succ \lfloor$Type type$\rfloor \Rightarrow \lfloor$Type$\rfloor$

2.3.3 Type Attributes

This section describes how to associate an attribute with a type. Attributes are non-type information that is associated with a type. Attributes are associated with a type rather than a TypeDecl because they can be associated with just a part of a type declaration. Attributes should generally not affect type equivalence (except perhaps in some minor cases). The notion of associating an attribute with a type has strong implications for the implemenation. Ideally, equivalent types would share the same physical representation. However, types that differ only in attributes should be equivalent, yet cannot share attributes.

The possible attributes are provided by the following enumeration:

```
enum TypeAttribute { cTraced, cUntraced, cConstantType, cVolatile,
                     cOrdered, cUnordered };
```

where:

cTraced Supports Modula-3's traced data types.

cUntraced Supports Modula-3's untraced data types. This is the default attribute for all types.

cConstantType Indicates that instances of this type have a constant value. By default, instances of a type are mutable.

One could reasonably argue that the immutability of a value is not a property (or attribute) of a type. However, C++'s typedef construct permits immutability to be included with the type.

cVolatile Marks a value which may be changed by something which a compiler cannot detect.

cOrdered For types with substructures, this attribute indicates if the source language requires the data layout to preserve the order of substructures (declaration order is assumed).

cUnordered For types with substructures, this attribute indicates if the type may be layed out in an arbitrary order.

void **SetTypeAttribute** *(TypeAttributes ta)* $\succ \lfloor \text{Type t} \rfloor \Rightarrow \lfloor \text{Type} \rfloor$

This routine associates an attribute with a type.

2.4 Generating Declarations

This section covers how declarations are handled by the generation interface. Declarations are one way of binding attributes (or values) to an identifier (or name). The generation interface divides a declaration into three parts: a name, a value, and miscellaneous attributes. Anonymous declarations exist, but they behave as though the system creates a unique identifier for the declaration. The generation interface supports the declaration of types, values (*e.g.*, variables and record fields), labels, procedures, and exceptions (for Modula-3). The interface also supports a variety of additional attributes.

2.4.1 Declaring Types

NameID **DeclType** *(Identifier name)* $\succ \lfloor \text{Type t} \rfloor \Rightarrow \lfloor \text{TypeDecl} \rfloor$

This routine creates a new type by making an entry for it in the symbol table.

Implementation Note: To support our type matching scheme, this routine should perform an extra step. It should first brand *t* with a NameBrandType, and then use the updated type for the declaration.

NameID **NameType** *(Identifier name, TypeID type)* $\succ \lfloor \rfloor \Rightarrow \lfloor \text{TypeDecl} \rfloor$

This routine creates an alias for a type which may already have an entry in the symbol table. This routine implements the semantics of C++'s *typedef* construct. Notice the use of *TypeID* rather than Type.

Implementation Note: If the type already has a name brand, then this name should simply be added to the existing name brand. Otherwise, a name brand should be created for it.

2.4.1.1 Forward Declarations

NameID **ForwardDeclProcedure** *(Identifier name)* $\succ \lfloor \text{Signature} \rfloor \Rightarrow \lfloor \text{RoutineDecl} \rfloor$

In Java terms, this routine has been **deprecated**. Use `SpcfyProcedure` instead.

2.4.1.2 Opaque Types

Opaque types are type names for which the full type structure is unknown. Instead, we know that the type is a subtype of the specified type.

Modula-3 distinguishes between an opaque declaration and a partial revelation. This distinction is not important to the generation interface, so a single routine exists. From the generation interface's point of view, an opaque declaration creates a name for a partially specified type.

Fully revealed opaque types are really just a type declaration, and should be represented as such.

Note that implementations will most likely wish to create a special type constructor that marks an opaque type as representing the super type.

NameID **DeclOpaque** *(Identifier name, TypeID superType)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{TypeDecl} \rfloor$

> This routine handles the declaration of opaque types and partial revelations. An opaque type is a Modula-3 construct whereby a type is only specified as being a subtype of another type. Additional information may be provided by partial revelations, which merely provide a more specific super type.
>
> We considered associating partial revelations with the original opaque declaration. However, the interface does not need to do error checking, so this associationg is not necessary.
>
> A complete revelation provides the full type definition, and is therefore represented as a normal type declaration.
>
> *Implementation suggestion:* An opaque declaration associates a name with a type. However, the name is not an instance of the type, but rather an instance of a subtype of that type. Hence, we recommend creating a *subtype* type constructor. The symbol table entry for an opaque type may then point to a subtype node in the type table. Note that the generation interface does not provide any routines to directly generate a subtype type node.

2.4.2 Declaring Values

Many source languages optionally allow programmers to give a default value (default value for formal parameters and initial value for variables) to a named values in its declaration. Since the generation interface requires an expression corresponding to the default value to be on the stack, user code should use the **noExpression** routine to push a special expression onto the stack.

Implementation note: Implementations may choose to represent initial values for variables and formals as a simple assignment following the declaration. Since the generation interface is intended to support a variety of intermediate representations, it provides as much support for the original syntax as possible and allows implementations to choose how best to represent initial values. Moreover, an initialization of a C++ reference type is different from an assignment to it.

NameID **DeclVariable** *(Identifier name)* $\succ$ $\lfloor$Type t, Expression initialValue$\rfloor$ $\Rightarrow$ $\lfloor$ValueDecl$\rfloor$

This routine declares a variable.

Parameters:

`name` The variable's name.

`t` The variable's type.

`initialValue` The variable's initial value.

NameID **DeclTemporary** *(Identifier name)* $\succ$ $\lfloor$Type t, Expression initialValue$\rfloor$ $\Rightarrow$ $\lfloor$ValueDecl$\rfloor$

This routine declares a temporary variable. We provide this routine to distinguish programmer declared variables and compiler created variables.

Parameters:

`name` The variable's name.

`t` The variable's type.

`initialValue` The variable's initial value.

NameID **DeclFormal** *(Identifier name, Mode m)* $\succ$ $\lfloor$Type t, Expression defaultValue$\rfloor$ $\Rightarrow$ $\lfloor$ValueDecl$\rfloor$

This document uses the term *argument* to refer to an actual parameter and *formal* to refer to a formal parameter.

Parameters:

`name` The formal's name.

`m` The formal's parameter mode.

`t` The formal's type.

`defaultValue` The formal's default value.

Implementations of the generation interface must provide an enumerated list of parameter mode options, such as the following:

```
enum Mode {cInValue, cValue, cReference, cValueResult, cResult};
```

where:

cInValue Pass-by-value but the formal's value may not be altered. This mode is used for Ada.

cValue Pass-by-value but the formal's value may be altered. This mode is the default for Modula-3 and the only one for C/C++.

cReference Formal parameter is an alias for the argument. Therefore, an update to the formal parameter is a direct update the argument (actual parameter) as well. This mode is the only parameter mode for Fortran and represents the *var* mode for Modula-3.

cReadOnly Pass-by-reference, but the value cannot be updated. This mode supports Modula-3's *readonly* mode, and is efficient for large data structures.

cValueResult Represents copy-in and copy-out semantics. This mode is used for Ada's *inout* mode.

cInOut Represents Ada's *inout* keyword, which allows implementations to choose between **Reference** and **ValueResult**.

cResult Represents Ada's *out* mode, which allows implementations to choose between **Reference** and **ValueResult**.

Handling C++'s parameter modes is awkward. In C++, programmers may declare a formal to be constant (*i.e.*, pass-by-inValue) or to be a reference (*i.e.*, pass-by-reference). Unfortunately, this declaration may be buried in the type declaration used in the definition of the formal. These unfortunate declaration semantics requires searching through the symbol table to determine the parameter passing mode, and would require that clients recognize the duplication and ignore it. Hence, the generation interface dictates that for C++, all calls to **declFormal** should use pass-by-value.

void **UnknownFormals** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor$ValueDecl$\rfloor$

This routine indicates that the remainder of a routine's signature has an unknown number of formal parameters. Hence, this routine implements C++'s "..." construct.

void **AlternateReturnFormal** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor$ValueDecl$\rfloor$

Fortran provides an unusual version of a return statement called an alternate return. An alternate return returns to one of possibly many line numbers, where the possible return locations are provided in special procedure parameters. The Fortran syntax for these special parameters is an asterik ($*$), which this routine represents.

Alternate return parameters may occur anywhere in the parameter list; however, the alternate return construct views these special parameters as forming a simple linear list from 1 to n.

NameID **DeclField** *(Identifier name)* $\succ \lfloor$Type t, Expression initialValue$\rfloor \Rightarrow \lfloor$FieldDecl$\rfloor$

This routine declares a field, which is a component of an aggregate data structure.

Parameters:

`name` The field's name.

`t` The field's type.

`initialValue` The field's initial value.

NameID **DeclConstant** *(Identifier name)* $\succ$ $\lfloor$Type td, Expression value$\rfloor$ $\Rightarrow$ $\lfloor$ConstantDecl$\rfloor$

This routine declares a constant value. A constant value does not require storage to be allocated, which is different from other declarations with a constant attribute.

Parameters:

`name` The constant's name.

`t` The contant's type.

`value` The contant's value.

2.4.3 Declaring Procedures and Methods

Unlike most other declared entities, procedures are generally not fully declared at the point of their declaration. The generation interface refers to a function declaration without a body (*i.e.*, a C/C++ prototype or a Modula-3 procedure interface) as a *specification*, and a function declaration with a body as a *declaration*.

Hence, the compiler or linker must ultimately resolve all function specifications back to their corresponding declarations. For Modula-3 this process is straightforward because of the well-structured import/export mechanism. For C/C++, this process is a little more work.

C++ ([2], p. 138) and Modula-3 ([1], p. 27) define slightly different type matching rules for procedures. Since the generation interface currently supports only statically typed languages and then only when semantic resolution has already been done, we ignore these differences.

Initializers

C++ allows a list of initializers to be provided along with constructors. These routines handle the representations of initializers.

void **Initializer** *(NameID initializedEntity)* $\succ$ ⌊ArgumentList al⌋ $\Rightarrow$ ⌊Initializer⌋

void **InitializersBegin** *()* $\succ$ ⌊⌋ $\Rightarrow$ ⌊⌋

void **InitializersEnd** *()* $\succ$ ⌊$*$⌋ $\Rightarrow$ ⌊Initializers⌋

At the point that the **InitializersEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **InitializersBegin** call) should contain only ArgumentList nodes.

void **NoInitializers** *()* $\succ$ ⌊⌋ $\Rightarrow$ ⌊Initializers⌋

This routine creates an empty initializer list.

Routine Specification

A specification indicates the name and type of a procedure but does not provide its body.

NameID **SpcfyProcedure** *(Identifier name)* $\succ$ ⌊Signature s⌋ $\Rightarrow$ ⌊RoutineDecl⌋

NameID **SpcfyNestedProcedure** *(Identifier name, int level, NameID parentRoutine)* $\succ$ ⌊Signature s⌋ $\Rightarrow$ ⌊RoutineDecl⌋

NameID **SpcfyMethod** *(Identifier name)* $\succ$ ⌊Signature s⌋ $\Rightarrow$ ⌊RoutineDecl⌋

NameID **SpcfyFriend** *(Identifier name)* $\succ$ ⌊Signature s⌋ $\Rightarrow$ ⌊RoutineDecl⌋

NameID **SpcfyTypeConversion** *(Identifier name)* $\succ$ ⌊Signature s⌋ $\Rightarrow$ ⌊RoutineDecl⌋

NameID **SpcfyConstructor** *(Identifier name)* $\succ$ ⌊Signature s, Initializers i⌋ $\Rightarrow$ ⌊RoutineDecl⌋

NameID **SpcfyDestructor** *(Identifier name)* $\succ$ ⌊Signature s⌋ $\Rightarrow$ ⌊RoutineDecl⌋

Routine Declaration

A routine declaration includes the routine's body.

NameID **DeclProcedure** *(Identifier name, int level, NameID parentRoutine)* $\succ$ ⌊Signature s, Statement body⌋ $\Rightarrow$ ⌊RoutineDecl⌋

NameID **DeclMethod** *(Identifier name, TypeID class)* $\succ$ ⌊Signature s, Statement body⌋ $\Rightarrow$ ⌊RoutineDecl⌋

The class type is specified if the method is defined outside the class. Otherwise, use NoType.

NameID **DeclMethodReference** *(Identifier name, NameID proc)* $\succ$ ⌊Signature s⌋ $\Rightarrow$ ⌊RoutineDecl⌋

This routine supports Modula-3 style method definition in which a method is defined by a top level procedure.

NameID **DeclOverride** *(Identifier name, NameID proc)* $\succ$ ⌊Signature s⌋ $\Rightarrow$ ⌊RoutineDecl⌋

A Modula-3 override looks a lot like a method definition; however, its affects the method table differently.

NameID **DeclTypeConversion** *(Identifier name, TypeID class)* $\succ$ ⌊Signature s, Statement body⌋ $\Rightarrow$ ⌊RoutineDecl⌋

The class type is specified if the method is defined outside the class. Otherwise, use NoType.

NameID **DeclConstructor** *(Identifier name, TypeID class)* $\succ$ ⌊Signature s, Initializers, Statement body⌋ $\Rightarrow$ ⌊RoutineDecl⌋

The class type is specified if the method is defined outside the class. Otherwise, use NoType.

NameID **DeclDestructor** *(Identifier name, TypeID class)* $\succ$ ⌊Signature s, Statement body⌋ $\Rightarrow$ ⌊RoutineDecl⌋

The class type is specified if the method is defined outside the class. Otherwise, use NoType.

NameID **DeclEntry** *(Identifier name, NameID enclosingProcedure)* $\succ$ ⌊Signature s, Statement s1⌋ $\Rightarrow$ ⌊Statement⌋

Fortran 77 allows procedures to have multiple entry points. The handling of variables for multiple entry points is messy at best. The generation interface uses Fortran 77 semantics. Since each entry can have its own argument list, the location (on the stack) of a parameter depends upon which entry point into the procedure was called. Moreover, parameters for one entry point may not be defined by another in which case the latter entry point cannot execute code which refers to such parameters.

The operation of this routine is different from most routines in the generation interface. In order to mark where the entry point begins, the generation interface requires that the first statement in the body of the entry be passed in `s1`. Instead of returning a RoutineDecl like the other routines in this section, it returns a statement, which is in fact `s1`. The RoutineDecl created by this routine is accessible through the `NameID` returned by this routine.

Parameters:

`name` Name of the entry point.

`s` Signature of the entry point.

`enclosingProcedure` The procedure which encloses this entry point.

`s1` The first statement in entry. Note that `s1` is only the first statement executed, not the entire body of the routine as used in other routine declarations.

2.4.4 Declaring Exceptions

The routines in this section are for declaring exceptions. Modula-3 has entities of Exception type; C++ never actually creates an exception entity. Rather than pass an entity of type Exception, when a C++ program throws an exception, it passes the arguments to an exception handler.

NameID **DeclException** *(Identifier name, TypeID type)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{ExceptionDecl} \rfloor$

For exceptions without an argument, use NoType as the *type* argument.

2.4.5 Declaring Labels

C and C++ treat case alternatives as labels. This bizarre notion stems from their excessively low-level view of the switch statement. The generation interface handles case alternatives differently from general labels.

NameID **DeclLabel** *(Identifier i)* $\succ \lfloor\rfloor \Rightarrow \lfloor \mathsf{LabelDecl} \rfloor$

2.4.6 Code Units

A code unit groups related code. Code units include modules and namespaces.

2.4.6.1 Namespace

A C++ namespace can be thought of as a named scope.

void **DeclNamespaceUnitBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

NameID **DeclNamespaceUnitEnd** *(Identifier name)* $\succ \lfloor * \rfloor \Rightarrow \lfloor \mathsf{NamespaceDecl\ nd} \rfloor$

At the point that the **DeclNamespaceUnitEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **DeclNamespaceUnitBegin** call) should contain only Declaration nodes.

2.4.6.2 File

These routines are meant for demarcating a C/C++ file scope. These routines are not meant for annotating a graph with which source file a particular piece of code originates from.

void **FileUnitBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **FileUnitEnd** *(String name)* $\succ \lfloor * \rfloor \Rightarrow \lfloor \textsf{Unit} \rfloor$

At the point that the **FileUnitEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **FileUnitBegin** call) should contain only Declaration nodes.

2.4.6.3 Interface

void **DeclInterfaceUnitBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

NameID **DeclInterfaceUnitEnd** *(Identifier name)* $\succ \lfloor * \rfloor \Rightarrow \lfloor \textsf{InterfaceDecl} \rfloor$

At the point that the **DeclInterfaceUnitEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **DeclInterfaceUnitBegin** call) should contain only Declaration nodes.

2.4.6.4 Module

void **ImportUnit** *(NameID interface, Identifier name)* $\succ \lfloor\rfloor \Rightarrow \lfloor \textsf{Import} \rfloor$

This routine imports interface `i` as `name`. If `name` is NoName, then the interface is not renamed.

void **ImportMember** *(NameID interface, NameID declaration)* $\succ \lfloor\rfloor \Rightarrow \lfloor \textsf{Import} \rfloor$

This routine imports a single member (`declaration`) of `interface`. Note that this routine accepts more information (*i.e.*, `interface`) than it strictly needs.

void **ImportsBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **ImportsEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor \textsf{Imports} \rfloor$

At the point that the **ImportsEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **ImportsBegin** call) should contain only Import nodes.

void **ExportsBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **ExportsEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor \textsf{Exports} \rfloor$

At the point that the **ExportsEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **ExportsBegin** call) should contain only Identifier nodes.

void **DeclModuleUnitBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

NameID **DeclModuleUnitEnd** *(Identifier name)* $\succ \lfloor * \rfloor \Rightarrow \lfloor \text{Unit} \rfloor$

At the point that the **DeclModuleUnitEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **DeclModuleUnitBegin** call) should contain only Declaration nodes.

2.4.7 Setting Declaration Attributes

Declarations may have attributes associated with them. The current design specifies the attributes in an enumerated type.

An alternate design would use a separate routine for each attribute, so that a compiler/linker could catch non-compliance to the interface. However, it seems unnecessarily verbose when compared to a single function with an enumerated type

The attributes break down into roughly five categories. The first group of attributes define which entities have values that can change. The second group of attributes indicate in what part of memory an entity should be located. The most important aspect of the location decision is whether or not the entity's value is saved across procedure calls. The third group of attributes determine the visibility of the entity. These attributes correspond to C/C++'s notion of linkage. The remaining two groups of attributes control method visibility and overloading.

The `DeclAttributes` enumerated type is defined as follows:

```
enum DeclAttributes { cConstantDecl, cMutable, cLocationStack,
cLocationRegister, cLocationStatic, cLocationInline, cLinkageLocal,
cLinkageFile, cLinkageGlobal, cLinkageForeign, cPublic, cProtected,
cPrivate, cAbstract, cNonvirtual, cVirtual, cNoConstructorCalls,
cNoDestructorCalls, cDestructorCalls };
```

where:

Entity mutability

The mutability attributes only apply to variables, fields, and routines.

cConstantDecl Indicates that an entity is a constant. This attribute is equivalent to the type attribute of the same name, but this one is preferred.

cMutable Indicates than an entity is *not* a constant. This is the default value for mutability. It only needs to be explicitly specified to indicate that a component of a constant aggregate entity is not constant (as in C++).

Entity location

The location attributes only apply to variables and fields.

cLocationStack Marks a data value as being locally allocated (*i.e.*, on the stack). Local allocation is the default form of allocation and so seldom (if ever) needs to be explicitly specified. This attribute represents C/C++'s *auto* construct.

cLocationRegister Recommends that a value be assigned to a register. If the value cannot be assigned to a register, it should be allocated on the stack.

cLocationStatic Indicates that an entity is assigned to permanently allocated space.

cLocationInline Recommends that a procedure be inlined. This attribute applies only to procedures and methods.

Entity visibility

The entity visibility attributes apply to all declarations.

cLinkageLocal Indicates that an entity is visible only within its current scope. This is the default linkage.

cLinkageFile Indicates that the entity is visible only within its file scope. This attribute is intended to represent one meaning of the C/C++ *static* construct. Admittedly, this attribute should be redundant with `linkageLocal` when at the file scope level. However, this captures the notion of C/C++'s *static* construct; whereas, `linkageLocal` should never be used.

cLinkageGlobal Indicates that an entity has globally visible in the program. This attribute implements the common case of C/C++'s *extern* construct.

cLinkageForeign Indicates that an entity is visible to a different source language. This attribute eliminates C++'s overloading of the *extern* keyword.

Method visibility

The method visibility attributes potentially apply to all declarations, but for currently support languages, the attributes only affect fields and methods.

cPublic Specifies that an identifier is visible outside of its namespace. This value is the default.

cProtected Specifies than an identifier is visible only within its namespace. This attribute may only be used for class members, in which case it denotes the semantics of C++'s *protected* construct.

cPrivate Specifies that an identifier is visible only within its namespace.

Method overloading

The method overloading attributes apply only to methods.

cAbstract Marks a method as undefined for the current class, and therefore requiring definition in derived classes. By default, the generation interface assumes that all methods are fully defined.

cNonvirtual Indicates that a method may not be overloaded.

cVirtual Indicates that a method may be overloaded. This value is the default.

Automatic methods

The automatic method attributes apply only to class declarations.

cNoConstructorCalls Marks a class as not requiring automatic invocation of its constructor. This is the default value.

cConstructorCalls Marks a class as requiring automatic invocation of its constructor.

cNoDestructorCalls Marks a class as requiring automatic invocation of its destructor. This is the default value.

cDestructorCalls Marks a class as requiring automatic invocation of its destructor. Note that automatic constructor and destructor are indicated separately in order to support Java which does not have destructors.

cMainProcedure Marks the procedure as the main procedure in the program.

void **SetDeclarationAttribute** *(DeclAttributes da)* $\succ$ $\lfloor$Declaration d$\rfloor$ $\Rightarrow$ $\lfloor$Declaration$\rfloor$

This routine assigns attribute `da` to declaration `d`. For those attributes which do not require a value, use NoExpression.

2.5 Generating Statements

This section describes the routines available for representing statements.

2.5.1 Block Statements

Different programming languages have different rules about when a block opens a scope. The generation interface separates the issues of grouping statements from that of declaring scopes (see Section 2.1.4.1. Hence, none of the routines in this section imply the creation of a new scope. A block of statements may have an associated scope (*e.g.*, a procedure with local variables). Client code is responsible for associating a scope with a block of statements.

void **StmtBlockBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **StmtBlockEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor$Statement$\rfloor$

At the point that the **StmtBlockEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **StmtBlockBegin** call) should contain only Statement nodes.

2.5.2 Labeled Statements

This section describes routines for representing labeled statements.

void **StmtLabel** *()* $\succ \lfloor$LabelDecl, Statement$\rfloor \Rightarrow \lfloor$LabelStmt$\rfloor$

2.5.3 Conditional Statements

This section describes routines for representing decision statements. This section also defines routines for specifying conditions.

2.5.3.1 If statements

The routines in this section handle *if* statements. The interface does not provide direct support for *else-if* clauses, so these clauses will have to be transformed to nested *if* statements. The design of these routines require that for nested if statements, outer *if* statements remain on the stack until inner *if* statements have been processed.

void **StmtIfThenElse** *()* $\succ$ ⌊Expression e, Statement s1, Statement s2⌋ $\Rightarrow$ ⌊Statement⌋

The expression representing the test must be of boolean type. If the statement does not have an *else* clause, then a null statement should be used for s2.

void **StmtArithmeticIf** *(NameID lessLabel, NameID equalLabel, NameID moreLabel)* $\succ$ ⌊Expression e⌋ $\Rightarrow$ ⌊Statement⌋

This routine implements the semantics of Fortran 77's arithmetic if statement. The expression representing the test must be of boolean type.

2.5.3.2 Multi-way Branch Statements

These routines perform a single test and jump to one of potentially many points in the program.

void **LabelsBegin** *()* $\succ$ ⌊⌋ $\Rightarrow$ ⌊⌋

void **LabelsEnd** *()* $\succ$ ⌊⌋ $\Rightarrow$ ⌊Labels⌋

At the point that the **LabelsEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **LabelsBegin** call) should contain only LabelRef nodes.

void **StmtComputedGoto** *()* $\succ$ ⌊Labels l, Expression e⌋ $\Rightarrow$ ⌊Statement⌋

This routine implements the semantics of Fortran 77's *computed goto* construct. It is similar to a switch statement, except that the target labels are not limited to a single statement of code.

void **StmtAssignLabel** *(NameID value, NameID label)* $\succ$ ⌊⌋ $\Rightarrow$ ⌊Statement⌋

This routine implements the semantics of Fortran 77's *assign* construct.

Note that the semantics of this operation could be represented with a type conversion from integer to label. However, this would require creating variables of type label. Since labels are deprecated, this more direct representation has been chosen.

void **StmtAssignedGoto** *()* $\succ$ ⌊Expression e, Labels l⌋ $\Rightarrow$ ⌊Statement⌋

This routine implements the semantics of Fortran 77's *assigned goto* construct.

2.5.3.3 Case/Switch Statements

The generation interface supports two types of multiple test conditional statement. The first form is a well structured statement, as found in Modula-3. We refer to this form as a case statement. The body of a case statement consists of blocks of code, each of which when paired with their case keys is called a case alternatives. Each time a program passes through a case statement, it will execute zero or one of the case alternatives. A case alternative is executed when the value of the conditional expression matches one of its case keys. The other form matches the unstructured form of C/C++'s *switch* statement. This form still uses case keys, but does not separate the *switch* statement body into case alternatives.

The generation interface follows Modula-3's terminology of *case* statement. Each unique body of code within a *case* statement is termed a case alternative, and each case alternative may have more than one case key. A case key is the value which is compared against the expression at the top of the *case* statement.

void **CaseKeyBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **CaseKeyEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor \textsf{CaseKeys} \rfloor$

At the point that the **CaseKeyEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **CaseKeyBegin** call) should contain only Expression nodes.

void **CaseKey** *()* $\succ \lfloor \textsf{Expression e} \rfloor \Rightarrow \lfloor \textsf{CaseKeys} \rfloor$

The routines **CaseKeyBegin** and **CaseKeyEnd** build a list of case keys. The **CaseKey** routine is a short cut for building a list with one member.

void **CaseAlt** *()* $\succ \lfloor \textsf{CaseKeys ck, Statement s} \rfloor \Rightarrow \lfloor \textsf{CaseAlt} \rfloor$

void **CaseOthersAlt** *()* $\succ \lfloor \textsf{StatementList} \rfloor \Rightarrow \lfloor \textsf{CaseAlt} \rfloor$

This routine represents the default case alternative.

void **CaseBodyBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **CaseBodyEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor \textsf{CaseAlts} \rfloor$

The routines **CaseBodyBegin** and **CaseBodyEnd** build the body of a case statement.

At the point that the **CaseBodyEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **CaseBodyBegin** call) should contain only CaseAlt nodes.

void **StmtCase** *()* $\succ \lfloor \textsf{Expression e, CaseAlts} \rfloor \Rightarrow \lfloor \textsf{Statement} \rfloor$

This routine implements the semantics of Modula-3's *case* statement, in which the order of the case alternatives does not matter. The expression representing the test must be of boolean type.

void **StmtSwitch** *()* $\succ$ ⌊Expression e, Statement s⌋ $\Rightarrow$ ⌊Statement⌋

This routine implements the semantics of C/C++'s *switch* construct. The expression representing the test must be of boolean type.

2.5.3.4 Typecase

Modula-3's *typecase* statement allows a conditional expression based on an expressions type. The routines provided for handling a *typecase* mirror those provided for the *case* statement.

void **TypecaseKey** *(TypeID type, NameID variable)* $\succ$ ⌊⌋ $\Rightarrow$ ⌊TypecaseKey⌋

If `variable` is NoName, then no variable has been specified.

void **TypecaseKeyBegin** *()* $\succ$ ⌊⌋ $\Rightarrow$ ⌊⌋

void **TypecaseKeyEnd** *()* $\succ$ ⌊$*$⌋ $\Rightarrow$ ⌊TypecaseKeys⌋

At the point that the **TypecaseKeyEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **TypecaseKeyBegin** call) should contain only TypecaseKeys nodes.

void **TypecaseAlt** *()* $\succ$ ⌊TypecaseKeys tck, Statement s⌋ $\Rightarrow$ ⌊TypecaseAlt⌋

void **TypecaseBodyBegin** *()* $\succ$ ⌊⌋ $\Rightarrow$ ⌊⌋

void **TypecaseBodyEnd** *()* $\succ$ ⌊$*$⌋ $\Rightarrow$ ⌊TypecaseAlts⌋

At the point that the **TypecaseBodyEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **TypecaseBodyBegin** call) should contain only TypecaseAlt nodes.

void **StmtTypecase** *()* $\succ$ ⌊Expression e, TypeCaseAlts⌋ $\Rightarrow$ ⌊Statement⌋

This routine implements the semantics of Modula-3's *case* statement. In a *typecase* the order of the case alternatives does matter. The expression representing the test must be of boolean type.

2.5.4 Looping Statements

void **StmtWhileLoop** *()* $\succ$ $\lfloor$Expression e, Statement s$\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

The expression representing the test must be of boolean type.

void **StmtRepeatWhileLoop** *()* $\succ$ $\lfloor$Statement s, Expression e$\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

The expression representing the test must be of boolean type.

void **StmtRepeatUntilLoop** *()* $\succ$ $\lfloor$Statement s, Expression e$\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

The expression representing the test must be of boolean type.

void **StmtLoop** *()* $\succ$ $\lfloor$Statement s$\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

void **StmtDoLoop** *(NameID index)* $\succ$ $\lfloor$Expression first, Expression last, Expression step, Statement s$\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

This routine represents an iterating loop. When *step* is negative, the loop terminates when *index* becomes lower than `last`. Otherwise, the loop terminates when *index* exceeds *last*. The *first*, *last*, and *step* expressions are evaluated once, at entry to the loop.

Parameters:

`index` Variable (or other assignable name) being incremented.

`first` Initial value of `index`.

`last` Maximum value of `index`.

`step` Amount by which to increment `index` each time through the loop.

This routine follows Fortran 77 semantics, which requires that the index be updated. However, some source languages do not guarantee that the value of the index is meaningful after the loop finishes, which provides additional opportunities for optimizations. For such source languages, user code may create a local (*i.e.*, temporary) variable for the loop index. In this case, the index will have no uses beyond the loop, so client code may optimize more aggressively.

void **StmtForLoop** *()* $\succ$ $\lfloor$Expression d, Expression e, Expression e, Statement s$\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

The expression representing the test must be of boolean type.

2.5.5 Branch Statements

This section lists branching statements. The statements included here are simple branches, Fortran 77 has several elaborate branch statements which may be found in Section 2.5.

void **StmtBreak** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\mathsf{Statement}\rfloor$

void **StmtContinue** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\mathsf{Statement}\rfloor$

void **StmtGoto** *(NameID label)* $\succ \lfloor\rfloor \Rightarrow \lfloor\mathsf{Statement}\rfloor$

void **StmtReturn** *()* $\succ \lfloor\mathsf{Expression\ e}\rfloor \Rightarrow \lfloor\mathsf{Statement}\rfloor$

If the return statement does not have an expression, then NoExpression should be used.

void **StmtExit** *()* $\succ \lfloor\mathsf{Expression\ e}\rfloor \Rightarrow \lfloor\mathsf{Statement}\rfloor$

This routine terminates a program. If no expression is specified, NoExpression should be used.

void **StmtThrow** *(NameID value)* $\succ \lfloor\rfloor \Rightarrow \lfloor\mathsf{Statement}\rfloor$

This routine throws a C++ style exception and branches to an appropriate exception handler.

void **StmtRaise** *(NameID exception)* $\succ \lfloor\mathsf{Expression\ e}\rfloor \Rightarrow \lfloor\mathsf{Statement}\rfloor$

This routine raises a Modula-3 style exception and branches to an appropriate exception handler. If no expression is available, NoExpression should be used.

void **StmtAlternateReturn** *()* $\succ \lfloor\mathsf{Expression\ e}\rfloor \Rightarrow \lfloor\mathsf{Statement}\rfloor$

This routine implements the semantics of Fortran 77's alternate return construct. The expression indexes the special alternate return parameters.

2.5.6 Exception Handling Statements

Though exception handling is similar in Modula-3 and C++, The generation interface provides separate routines for these languages. The primary difference is that a Modula-3 exception passes an entity of type Exception, whereas a C++ exception passes an argument list for an exception handler. The syntax of their respective *try* blocks is also slightly different.

Modula-3 style exceptions

void **ExceptionKey** *(NameID e)* $\succ \lfloor\rfloor \Rightarrow \lfloor \text{ExceptionKey} \rfloor$

void **ExceptionKeyBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **ExceptionKeyEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor \text{ExceptionKeys} \rfloor$

At the point that the **ExceptionKeyEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **ExceptionKeyBegin** call) should contain only ExceptionKey nodes.

void **ExceptionHandler** *()* $\succ \lfloor \text{ExceptionKeys eks, Statement s} \rfloor \Rightarrow \lfloor \text{Handler} \rfloor$

NameID **ExceptionHandlerWithArgument** *(NameID exception, Identifier i)* $\succ \lfloor \text{Statement s} \rfloor \Rightarrow \lfloor \text{Handler} \rfloor$

This routine represents an exception with an argument. Generation interface implementations are responsible for generating the declaration for *i*.

void **ElseHandler** *()* $\succ \lfloor \text{Statement s} \rfloor \Rightarrow \lfloor \text{Handler} \rfloor$

This routine implements Modula-3's else handler, which is called for any exception that reaches it.

void **HandlerBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **HandlerEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor \text{Handlers} \rfloor$

Each try block has a list of elements of type Handler associated with it.

At the point that the **HandlerEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **HandlerBegin** call) should contain only Handler nodes.

void **StmtTryExcept** *()* $\succ$ ⌊Statement s, Handlers hs⌋ $\Rightarrow$ ⌊Statement⌋

This routine implements Modula-3's *tryExcept* construct.

void **StmtTryFinally** *()* $\succ$ ⌊Statement s1, Statement s2⌋ $\Rightarrow$ ⌊Statement⌋

This routine implements Modula-3's *tryFinally* construct.

C++ style exceptions

void **CatchException** *()* $\succ$ ⌊FormalDecl vd, Statement stmt⌋ $\Rightarrow$ ⌊Catch⌋

void **CatchAll** *()* $\succ$ ⌊Statement stmt⌋ $\Rightarrow$ ⌊Catch⌋

This routine represents a catch clause that can handle any exception. This routine represents C++'s "..." notation.

void **CatchBegin** *()* $\succ$ ⌊⌋ $\Rightarrow$ ⌊⌋

void **CatchEnd** *()* $\succ$ ⌊$*$⌋ $\Rightarrow$ ⌊Catchers⌋

Each try block has a list of elements of type Catch associated with it.

At the point that the **CatchEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **CatchBegin** call) should contain only Catch nodes.

void **StmtTry** *()* $\succ$ ⌊Statement s, Catchers cs⌋ $\Rightarrow$ ⌊Statement⌋

This routine implements C++'s *try* construct.

2.5.7 Miscellaneous Statements

void **StmtSpcfyUsing** *(NameID id)* $\succ$ ⌊⌋ $\Rightarrow$ ⌊Statement⌋

void **StmtUsingDirective** *(NameID namespace)* $\succ$ ⌊⌋ $\Rightarrow$ ⌊Statement⌋

NameID **StmtWithAlias** *(Identifier id)* $\succ$ $\lfloor$Expression e, Statement s$\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

void **StmtEval** *()* $\succ$ $\lfloor$Expression e$\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

This routine converts an expression to a statement. It executes an expression and then discards the result. Hence, the expression is being executed only for its side effect. This routine implements Modula-3's *eval* construct and is used to represent C/C++'s implicit conversion from an expression to a statement.

void **StmtDeclStmt** *()* $\succ$ $\lfloor$Declaration d$\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

This routine converts a declaration to a statement, which allows a statement block to consist of all statements.

void **StmtNull** *()* $\succ$ $\lfloor\rfloor$ $\Rightarrow$ $\lfloor$Statement$\rfloor$

This routine represents a null statement.

2.6 Generating Expressions

This section shows the interface routines used to specify expressions. Note that the interface implementation is responsible for identifying type specific versions of the operators.

Operators can represent user-defined functions in languages which allow operators to be overloaded (*e.g.*, C++). In the generation interface, operators represent only language defined operators, not overloaded operators. Users should translate overloaded operators to function calls.

Tables 2.3, 2.4, and 2.5 show the correspondences between the interface's routines and operators of a few languages. This table shows correspondences between operators which programmers may use and the routines of the generation interface. However, user code may use operators which the programmer cannot directly use (*e.g.*, C/C++ implicitly truncate real numbers but users of this interface must explicitly call a conversion routine).

Operator Correspondences– Part I			
Routine Name	**C++**	**Modula-3**	**Fortran 77**
Assignment Operators			
ExpAssignment	=	:=	=
Arithmetic Operators			
ExpPositive	+	+	*N/A*
ExpNegative	−	−	−
ExpAbsoluteValue	*N/A*	abs	abs, ?abs
ExpMinimum	*N/A*	min	min, ?min∗
ExpMaximum	*N/A*	max	max, ?max∗
ExpAddition	+	+	+
ExpSubtraction	−	−	−
ExpMultiplication	∗	∗	∗
ExpDivision	/	/	/
ExpModulus	*N/A*	*N/A*	*N/A*
ExpRemainder	%	mod	mod, ?mod
ExpExponentiation	*N/A*	*N/A*	∗∗
ExpPreDecrement	−−	*N/A*	*N/A*
ExpPreIncrement	++	*N/A*	*N/A*
ExpPostDecrement	−−	*N/A*	*N/A*
ExpPostIncrement	++	*N/A*	*N/A*
Relational Operators			
ExpEquality	==	=	.EQ.
ExpNotEqual	!=	#	.NE.
ExpGreater	>	>	.GT.
ExpGreaterEqual	>=	>=	.GE.
ExpLess	<	<	.LT.
ExpLessEqual	<	<=	.LE.
Bitwise Operators			
ExpBitComplement	∼	*N/A*	*N/A*
ExpBitAnd	&	*N/A*	*N/A*
ExpBitXor	∧	*N/A*	*N/A*
ExpBitOr	|	*N/A*	*N/A*
ExpBitShiftLeft	<<	*N/A*	*N/A*
ExpBitShiftRight	>>	*N/A*	*N/A*

Table 2.3: Correspondence between generation interface ExpRoutine names and language operators. Exp-Continued in Table 2.4.

Operator Correspondences– Part II			
Routine Name	**C++**	**Modula-3**	**Fortran 77**
Compound Assignment Operators			
ExpMultiplicationAssignment	$*=$	*N/A*	*N/A*
ExpDivisionAssignment	/=	*N/A*	*N/A*
ExpRemainderAssignment	%=	*N/A*	*N/A*
ExpAdditionAssignment	+=	*N/A*	*N/A*
ExpSubtractionAssignment	$-=$	*N/A*	*N/A*
ExpBitShiftRightAssignment	$>>=$	*N/A*	*N/A*
ExpBitShiftLeftAssignment	$<<=$	*N/A*	*N/A*
ExpBitAndAssignment	&=	*N/A*	*N/A*
ExpBitXorAssignment	$\wedge=$	*N/A*	*N/A*
ExpBitOrAssignment	$\|=$	*N/A*	*N/A*
Logical Operators			
ExpNot	!	not	.NOT.
ExpAnd	*N/A*	*N/A*	.AND.
ExpOr	*N/A*	*N/A*	.OR.
ExpAndConditional	&&	and	*N/A*
ExpOrConditional	$\|\|$	or	*N/A*
ExpArithmeticIf	?:	*N/A*	*N/A*
Set Operators			
ExpSetEquality	*N/A*	=	*N/A*
ExpUnion	*N/A*	+	*N/A*
ExpDifference	*N/A*	$-$	*N/A*
ExpIntersection	*N/A*	$*$	*N/A*
ExpSymmetricDifference	*N/A*	/	*N/A*
ExpSubset	*N/A*	<=	*N/A*
ExpSuperset	*N/A*	>=	*N/A*
ExpElement	*N/A*	in	*N/A*
Pointer Operators			
ExpAddress	&	adr	*N/A*
ExpDereference	$*$	$\sim$	*N/A*
ExpNil	*N/A*	*nil*	*N/A*
Aggregate Operators			
ExpThis	this	*N/A*	*N/A*
ExpSelect	.	.	*N/A*
ExpSelectIndirect	$->$	.	*N/A*
ExpSelectRelative	$.*$	*N/A*	*N/A*
ExpSelectRelativeIndirect	$->*$	*N/A*	*N/A*

Table 2.4: Correspondence between generation interface ExpRoutine names and language operators. Exp-Continued from Table 2.3. ExpContinued in Table 2.5.

Operator Correspondences– Part III			
Routine Name	**C++**	**Modula-3**	**Fortran 77**
Array Operators			
ExpSubscript	[]	[]	()
ExpArrayEquality	*N/A*	=	.EQ.
ExpArrayInequality	*N/A*	#	.NE.
ExpArrayGreater	*N/A*	*N/A*	.GT.
ExpArrayGreaterEqual	*N/A*	*N/A*	.GE.
ExpArrayLess	*N/A*	*N/A*	.LT.
ExpArrayLessEqual	*N/A*	*N/A*	.LE.
ExpSlice	*N/A*	subarray	:
ExpRemainingSlide	*N/A*	*N/A*	:
ExpConcatenation	*N/A*	*N/A*	//
Invocation Operators			
ExpCallProcedure	()	()	()
ExpCallMethod	()	()	*N/A*
Heap Operators			
ExpAllocate	new	new	*N/A*
ExpDelete	delete	dispose	*N/A*
ExpDeleteArray	delete[]	dispose	*N/A*
Type Operators			
ExpBytesizeVariable	sizeof	bytesize	*N/A*
ExpBytesizeType	sizeof	bytesize	*N/A*
ExpBitsizeVariable	*N/A*	bitsize	*N/A*
ExpBitsizeType	*N/A*	bitsize	*N/A*
ExpAdrsizeVariable	*N/A*	adrsize	*N/A*
ExpAdrsizeType	*N/A*	adrsize	*N/A*
ExpIstype	*N/A*	istype	*N/A*
ExpNarrow	*N/A*	narrow	*N/A*
ExpTypecode	*N/A*	typecode	*N/A*
ExpNumber	*N/A*	number	*N/A*
ExpFirst	*N/A*	first	*N/A*
ExpLast	*N/A*	last	*N/A*
Type Conversion Operators			
ExpTypeConversion	()	*N/A*	*N/A*
Miscellaneous Operators			
ExpSeries	,	*N/A*	*N/A*
ExpParentheses	()	()	()
ExpSelectScope	::	.	*N/A*
ExpAggregation pair	{}	{}	*N/A*

Table 2.5: Correspondence between generation interface routine names and language operators. Continued from Table 2.4.

Operators are first divided by the types of their arguments and then by their type of function. Calls to overloaded operators are represented as routine calls, since user-defined operators may not fit neatly in our categories.

The generation interface does not have any rules for type conversion. Therefore, user code is responsible for inserting explicit type conversions. Unless otherwise stated, binary operators require both operands to be of the same type, and operators return an entity of the same type as their operands.

2.6.1 Base Expressions

A base expression is an expression without subexpressions. Some base expressions are listed in following sections with related operators.

2.6.1.1 Identifier Reference

void **ExpIdReference** *(NameID entity)* $\succ \lfloor \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

This routine represents the use of a declared entity.

Scope operators

The scope operator allows programmers to access entities which are hidden by other uses of the same identifier. Resolving which declared entity is intended by a particular reference is part of semantic resolution, and therefore handled by the language parser. Hence, these routines are superfluous and only exist for providing additional information to client code. Note that ultimately user code must call the **ExpIdReference** routine to gain access the entity's value.

NameID **SelectScope** *(NameID entity1, NameID entity2)* $\succ \lfloor \rfloor \Rightarrow \lfloor \text{Reference} \rfloor$

This operator searches the scope named by `entity1` for `entity2`. Note that if `entity2` is a type, this returns a reference to the type's declaration, not the actual type.

NameID **SelectGlobalScope** *(NameID entity)* $\succ \lfloor \rfloor \Rightarrow \lfloor \text{Reference} \rfloor$

This operator searches the global scope named for identifier `entity`. Note that if `entity2` is a type, this returns a reference to the type's declaration, not the actual type.

2.6.1.2 Literals

This section describes routines for expressing literals.

void **ExpLiteral** *(TypeID type, String value)* $\succ \lfloor\rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

Literals are difficult to handle since we may be cross compiling. Hence, all literals are transfered across the interface as a String constant. Implementations of the interface must be able to convert numeric constants from strings to a numeric value. Equality of literals should be determined by comparing numeric values, rather than string values.

2.6.1.3 No Expression

This section describes the special value, NoExpression. This node represents that an optional expression has not been specified. This node may only replace a genuine expression node in those cases where it is explicitly permitted. This node does not correspond to any source language construct.

void **ExpNoexpression** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

This routine pushes a special expression node onto the stack. The implementation is capable of distinguishing this special expression node as not representing a valid expression.

2.6.2 Expression Ordering Operators

The operators represented by routines in this section serve only order and structure other expressions. The functions of these operators are independent of they type of their arguments.

void **ExpSeries** *()* $\succ \lfloor \text{Expression e1, Expression e2} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

This routine implementes C/C++'s comma operator.

void **ExpParentheses** *()* $\succ \lfloor \text{Expression e} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

This routine indicates that the programmer enclosed the associated expression in parentheses. The generation interface does not need parentheses to override precendence rules. However also specify the order in which computations are performed. This information may be useful to optimizations which affect expression ordering.

Aggregation

Some languages allow instances of aggregate and array types to be assigned to by aggrregate values. These routines allow the structuring of expressions into aggregate expressions. Every aggregate element has a position associated with it. The generation interface only supports positional specification of aggregate elements. User code must convert keyword specifications to positional specifications.

void **ExpPositionSingle** *()* ≻ ⌊Expression e⌋ ⇒ ⌊Position⌋

This routine builds a Position node that represents a single position.

void **ExpPositionRange** *()* ≻ ⌊Bounds b⌋ ⇒ ⌊Position⌋

This routine builds a Position node that represents a range of positions.

void **ExpPositionAny** *()* ≻ ⌊Bounds b⌋ ⇒ ⌊Position⌋

This routine builds a Position node that represents any position. This value corresponds to Ada's *others* construct in an aggregate. Any has lower priority than other positions; it fills in those positions that no other aggregation element does.

void **ExpAggregationElement** *()* ≻ ⌊Position p, Expression e⌋ ⇒ ⌊AggregationElement⌋

This routine builds an element of an aggregation. Each aggregation element must specify its position in the final data type, but its position may be a range or *any*.

void **ExpAggregationBegin** *()* ≻ ⌊Type t⌋ ⇒ ⌊Type t⌋

void **ExpAggregationEnd** *()* ≻ ⌊Type t, ∗⌋ ⇒ ⌊Expression⌋

At the point that the **ExpAggregationEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **ExpAggregationBegin** call) should contain only AggregationElement nodes. The resulting expression is of type *t*. Note that the type can be restricted to nodes of CompositeType type.

2.6.3 Assignment Operators

void **ExpAssignSimple** *()* ≻ ⌊Expression e1, Expression e2⌋ ⇒ ⌊Expression⌋

This routine works for all datatypes. It performs a simple bit copy of `e1` (rvalue) into `e2` (lvalue) for **bitsize**`(e2)` bits. This routine is best used for singleton values, and the other assignment routines for aggregates and arrays. Note that for all assignment operators, we push the rvalue before lvalue.

void **ExpAssignComponents** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

This routine does a component-wise copy for aggregates and arrays. Hence, this routine is sensitive to the types of its arguments. This routine requires array arguments to be of the same shape and size and will insert code to ensure run-time compliance, which is to say that it follows Modula-3 semantics.

void **ExpAssignFixedString** *(LengthFunction l)* $\succ$ $\lfloor$Expression e1, Expression e2, Expression padding$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

This routine provides string-like assignment for fixed sized arrays (*e.g.*, static and open arrays). This function allows assignment of arrays of unequal lengths. If the source expression is longer than the target, only enough elements are copied to fill the target. If the target expression is longer than the source expression, then the target expression is padded.

Parameters:

`l` Indicates how to determine the the dynamic length of a string. The value is an element of an enumerated type:

```
enum LengthFunction {cFixedLength, cTerminated};
```

cFixedLength Use the fixed length of `e1`. This option is for use with arrays with a fixed size at the point of assignment. For example, Modula-3 open arrays always have their length fixed before any assignments may be done to the array.

cTerminated The array size is determined by an embedded termination value, which is assumed to be zero. This value would be appropriate for C/C++, if they had this type of assignment.

`e1` Target of the assignment.

`e2` Source value for the assignment.

`padding` Value with which to pad `e1` if `e1` is longer than `e2`.

2.6.4 Numeric Operators

Operands for Numeric operators must be of type integer, float, or fixed point. To provide support for C/C++, the generation interface allows numeric operators to be applied to pointers as well, in which case their bit patterns are interpreted as integers.

Some routines which operate on numeric types behave differently depending on its arguments' types. The generation interface has several options in how to discriminate between these behaviors. One solution is to provide provide a separate version of the routine for each possible argument type, but this approach creates needless additional routines. Another solution is to give appropriate routines a type parameter, but this approach provides redundant information to the implementation, since it needs type information to handle expressions (*e.g.*, declare temporaries). A third solution is to require implementations to extract type information from an operator's arguments. Though this approach requires implementations to extract information user code already has, it is unlikely to create additional work and greatly simplifies the interface. Note, that the interface requires user code to do complete type conversion; therefore, unless otherwise stated both arguments must be of the same type.

Different languages have different semantics for overflow, underflow, and divide-by-zero error conditions. Both C/C++ and Modula-3 leave error handling for these conditions implementation dependent, but Ada requires them to be caught. The routines representing operators with possible error conditions accept an extra argument which indicates the desired semantics. Its value comes the following enumerated type:

```
        enum ErrorHandling {cImplementationDefined, cCatchError};
```

cImplementationDefined This value indicates that the implementation is free to do as it chooses.

cCatchError This value indicates that these errors should be caught.

2.6.4.1 Arithmetic Operators

void **ExpPositive** *()* ≻ ⌊Expression e⌋ ⇒ ⌊Expression⌋

void **ExpNegative** *()* ≻ ⌊Expression e⌋ ⇒ ⌊Expression⌋

void **ExpAbsoluteValue** *()* ≻ ⌊Expression e⌋ ⇒ ⌊Expression⌋

void **ExpMinimum** *()* ≻ ⌊Expression e1, Expression e2⌋ ⇒ ⌊Expression⌋

void **ExpMaximum** *()* ≻ ⌊Expression e1, Expression e2⌋ ⇒ ⌊Expression⌋

void **ExpAddition** *(ErrorHandling eh)* ≻ ⌊Expression e1, Expression e2⌋ ⇒ ⌊Expression⌋

void **ExpSubtraction** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpMultiplication** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpDivision** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpModulus** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpRemainder** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

The generation interface defines remainder and modulus as follows:

- Remainder truncates towards zero, and the sign of its result equals the sign of its right operand.
- Modulus truncates towards negative infinity, and the sign of its result equals the sign of its left operand.

C/C++ (%) and Modula-3's (*MOD*) modulus operators actually implement the interface's remainder function. The generation interface's modulus supports Ada semantics.

void **ExpExponentiation** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpPreDecrement** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpPreIncrement** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpPostDecrement** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpPostIncrement** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

2.6.4.2 Relational Operators

The operands to these operators are of Numeric type, but the resulting value is of Boolean type.

void **ExpEquality** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpNotEqual** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpGreater** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpGreaterEqual** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpLess** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpLessEqual** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

2.6.4.3 Bitwise Operators

Bit operators (see also Section 2.6.4.4) interpret their arguments as a bit pattern. Hence, no special handling exists for real types.

void **ExpBitComplement** *()* $\succ$ ⌊Expression e⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitAnd** *()* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitXor** *()* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitOr** *()* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitShiftLeft** *()* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitShiftRight** *()* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

2.6.4.4 Compound Assignment Operators

These operators perform two simpler operations in a single step. The second operation is a simple assignment.

void **ExpMultiplicationAssignment** *(ErrorHandling eh)* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpDivisionAssignment** *(ErrorHandling eh)* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpRemainderAssignment** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor \Rightarrow \lfloor$Expression$\rfloor$

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpAdditionAssignment** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor \Rightarrow \lfloor$Expression$\rfloor$

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpSubtractionAssignment** *(ErrorHandling eh)* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor \Rightarrow \lfloor$Expression$\rfloor$

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitShiftRightAssignment** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor \Rightarrow \lfloor$Expression$\rfloor$

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitShiftLeftAssignment** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor \Rightarrow \lfloor$Expression$\rfloor$

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitAndAssignment** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor \Rightarrow \lfloor$Expression$\rfloor$

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitXorAssignment** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor \Rightarrow \lfloor$Expression$\rfloor$

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpBitOrAssignment** *()* $\succ$ $\lfloor$Expression e1, Expression e2$\rfloor \Rightarrow \lfloor$Expression$\rfloor$

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

2.6.5 Boolean Operators

Operands for Boolean operators must be of type boolean. C/C++ specify that logical operators can accept arguments of integral type but still return arguments of logical type. Since the generation interface does not permit logical operators to have integral operands, user code must type convert integral operands to boolean type.

Nevertheless, boolean values are considered to be an integer subrange, so that Numeric **equality** and **inequality** apply to boolean types.

void **ExpTrue** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

void **ExpFalse** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

> **True** and false act as through they are an enumerated type with false equal to zero and true equal to one. Hence, the type conversion routines for enumerated types may be used on these values.

void **ExpNot** *()* $\succ \lfloor \text{Expression e} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

void **ExpAnd** *()* $\succ \lfloor \text{Expression e1, Expression e2} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

void **ExpOr** *()* $\succ \lfloor \text{Expression e1, Expression e2} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

void **ExpAndConditional** *()* $\succ \lfloor \text{Expression e1, Expression e2} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

void **ExpOrConditional** *()* $\succ \lfloor \text{Expression e1, Expression e2} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

> The ***Conditional** forms implement short circuit semantics. Hence, C and C++ should use these forms.

void **ExpExpressionIf** *()* $\succ \lfloor \text{Expression e1, Expression e2, Expression e3} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

> This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

2.6.6 Pointer Operators

Operands for Pointer operators must be of type pointer. For equality and inequality, pointers are treated as bit patterns (*i.e.*, a Numeric type).

void **ExpAddress** *()* $\succ \lfloor \text{Expression e} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

void **ExpDereference** *()* ≻ ⌊Expression e⌋ ⇒ ⌊Expression⌋

void **ExpNil** *()* ≻ ⌊⌋ ⇒ ⌊Expression⌋

This routine generates a *nil* pointer value. Modula-3 has an explicit nil value. C++ specifies that when the integer value zero is converted to a pointer it becomes the nil pointer value (regardless of bit representation).

2.6.7 Aggregate Operators

In each of the routines with parameters, the first argument is of aggregate type, and the second argument is of field or routine type.

void **ExpThis** *()* ≻ ⌊⌋ ⇒ ⌊Expression⌋

This routine returns a pointer to the current object. It is only valid inside of a method.

void **ExpSelect** *()* ≻ ⌊Expression e1, Expression e2⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpSelectIndirect** *()* ≻ ⌊Expression e1, Expression e2⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpSelectRelative** *()* ≻ ⌊Expression e1, Expression e2⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

void **ExpSelectRelativeIndirect** *()* ≻ ⌊Expression e1, Expression e2⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding C++ operator (see Table 2.3, 2.4, or 2.5).

2.6.8 Array Operators

This section describes the operators available for manipulating arrays. Some array operators (*e.g.*, **assignment** and **deleteArray**) may be found in other sections.

2.6.8.1 Subscripting

Some source languages supply multiple dimension subscripts, while others, such as C++, require repeated application of a one-dimensional subscript operator. The generation interface supports both approaches. Multiple dimension subscripts may be composed using the **index** routines. For single dimensional subscripts, either of the **subscript** routines may be used.

void **ExpIndexBegin** *()* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **ExpIndexEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor \text{Indicies} \rfloor$

At the point that the **ExpIndexEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **ExpIndexBegin** call) should contain only Expression nodes.

void **ExpSubscript** *(bool boundsChecking)* $\succ \lfloor \text{Expression e, Indicies i} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

This routine represents a subscript operation. Some languages require subscript bound checks while other languages do not. The `boundsChecking` parameter allows user code to select between these two choices.

void **ExpSubscript1d** *(bool boundsChecking)* $\succ \lfloor \text{Expression e1, Expression e2} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

This routine is a short-cut for when only one dimension is specified. Some languages require subscript bound checks while other languages do not. The `boundsChecking` parameter allows user code to select between these two choices.

2.6.8.2 String Array Operators

void **ExpArrayEquality** *()* $\succ \lfloor \text{Expression e1, Expression e2} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

This routine returns true if the two arrays (`e1` and `e2`) have the same size and their elements are equal.

void **ExpArrayInequality** *()* $\succ \lfloor \text{Expression e1, Expression e2} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

This routine is the complement of **arrayEquality**.

void **ExpArrayGreater** *()* $\succ \lfloor \text{Expression e1, Expression e2} \rfloor \Rightarrow \lfloor \text{Expression} \rfloor$

This routine implements the corresponding Fortran 77 operator (see Table 2.3).

void **ExpArrayGreaterEqual** *()* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Fortran 77 operator (see Table 2.3).

void **ExpArrayLess** *()* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Fortran 77 operator (see Table 2.3).

void **ExpArrayLessEqual** *()* $\succ$ ⌊Expression e1, Expression e2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Fortran 77 operator (see Table 2.3).

void **ExpSlice** *()* $\succ$ ⌊Expression s, Expression b, Expression e⌋ $\Rightarrow$ ⌊Expression⌋

This routine extracts a contiguous section (*i.e.*, substring) of a one dimensional array.

Parameters:

s The array to be operated on.

l The beginning position of the substring.

e The ending position of the substring.

void **ExpRemainingSlice** *()* $\succ$ ⌊Expression s, Expression l⌋ $\Rightarrow$ ⌊Expression⌋

This routine extracts the substring of elements from position `l` to the end of the one dimensional array.

Parameters:

s The array to be operated on.

l The beginning position of the substring.

void **ExpConcatenation** *()* $\succ$ ⌊Expression s1, Expression s2⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Fortran 77 operator (see Table 2.3). Concatentation is performed into a temporary variable.

2.6.9 Set Operators

void **ExpSetEquality** *()* ≻ ⌊Expression se⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpUnion** *()* ≻ ⌊Expression se1, Expression se2⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpDifference** *()* ≻ ⌊Expression se1, Expression se2⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpIntersection** *()* ≻ ⌊Expression se1, Expression se1⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpSymmetricDifference** *()* ≻ ⌊Expression se1, Expression se2⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpSubset** *()* ≻ ⌊Expression se1, Expression se2⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpSuperset** *()* ≻ ⌊Expression se1, Expression se2⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpElement** *()* ≻ ⌊Expression e, Expression se⌋ ⇒ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

2.6.10 Call Operators

Call operators represent calls to routines. These calls may be either function calls or method calls.

void **ExpPositionalArgument** *()* ≻ ⌊Expression e⌋ ⇒ ⌊Argument⌋

void **ExpNamedArgument** *(Identifier name)* ≻ ⌊Expression e⌋ ⇒ ⌊Argument⌋

Arguments may be either positional or named. C++ uses only positional arguments, but Modula-3 and Ada use both. Note that **positionalArgument** really only type converts from Expression to Argument.

void **ArgumentsBegin** *()* ≻ ⌊⌋ ⇒ ⌊⌋

void **ArgumentsEnd** *()* ≻ ⌊∗⌋ ⇒ ⌊ArgumentList⌋

At the point that the **ExpArgumentsEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **ExpArgumentsBegin** call) should contain only Argument nodes.

void **ExpCallFunction** *()* ≻ ⌊Expression function, ArgumentList al⌋ ⇒ ⌊Expression⌋

void **ExpCallMethod** *()* ≻ ⌊Expression object, Expression method, ArgumentList al⌋ ⇒ ⌊Expression⌋

This routine generates the representation for a method invocation. The `object` parameter represents the object associate with the method. The `method` formal is the name of the method being called (It should be an IdReference node).

2.6.11 Heap Operators

The behavior of heap operators are affected by several compilation unit attributes (see Section 2.7). The allocation operators call constructors, if appropriate. The delete operators call destructors, if appropriate. In addition, implementations of the interface must be aware of whether or not garbage collection is being used.

void **ExpAllocate** *(TypeID type)* ≻ ⌊⌋ ⇒ ⌊Expression⌋

void **ExpAllocateDefault** *(TypeID type)* ≻ ⌊Expression e⌋ ⇒ ⌊Expression⌋

This routine is used when the allocation has a default value. For a class object with an initializer, the expression `e` is a function call to the class' initializer.

void **ExpAllocatePlacement** *(TypeID type)* ≻ ⌊Expression e, Expression p⌋ ⇒ ⌊Expression⌋

This routine allows both a default and a placement parameter (for C++). The argument `p` is a placement list parameter.

void **ExpAllocateOpenArray** *(TypeID type)* ≻ ⌊Indicies i⌋ ⇒ ⌊Expression⌋

This routine allocates an open array (as found in Modula-3), and sets the size of the array in each dimension.

void **ExpAllocateSettingFields** *(TypeID type)* ≻ ⌊ArgumentList al⌋ ⇒ ⌊Expression⌋

This routine allocates an aggregate entity and then uses the positional arguments specified in `al` to initialize the entity.

void **ExpDelete** *()* ≻ ⌊⌋ ⇒ ⌊Expression e⌋

void **ExpDeleteArray** *()* ≻ ⌊⌋ ⇒ ⌊Expression e⌋

This routine implements the semantics of C++'s delete array operator. Note that we do not support C++'s archaic number of elements parameter to the delete array operator.

2.6.12 Type Operators

2.6.12.1 Type Query Operators

void **ExpBytesizeVariable** *()* ≻ ⌊Expression variable⌋ ⇒ ⌊Expression⌋

This routine implements the Modula-3 *bytesize* operator as well as the C/C++ *sizeof* operator.

void **ExpBytesizeType** *(TypeID type)* ≻ ⌊⌋ ⇒ ⌊Expression⌋

This routine implements the Modula-3 *bytesize* operator as well as the C/C++ *sizeof* operator.

void **ExpBitsizeVariable** *()* $\succ$ ⌊Expression variable⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpBitsizeType** *(TypeID type)* $\succ$ ⌊⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpAdrsizeVariable** *()* $\succ$ ⌊Expression variable⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpAdrsizeType** *(TypeID type)* $\succ$ ⌊⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpIstype** *(TypeID type)* $\succ$ ⌊Type t⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpNarrow** *(TypeID type)* $\succ$ ⌊Type t⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpTypecode** *()* $\succ$ ⌊Expression e⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpNumber** *()* $\succ$ ⌊Expression e⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpFirst** *()* $\succ$ ⌊Expression e⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

void **ExpLast** *()* $\succ$ ⌊Expression e⌋ $\Rightarrow$ ⌊Expression⌋

This routine implements the corresponding Modula-3 operator (see Table 2.3).

2.6.12.2 Type Conversion Operators

Type conversion rules vary substantially between languages. Hence, user code is responsible for ensuring that all implicit conversions are made explicit.

Some languages provide a fixed set of type conversions, but C++ allows users to define conversion routines for classes. Hence in the generation interface, a type conversion is a triple consisting of the expression to be converted, the type to which it is to be converted, and a routine for performing the conversion. For language defined conversions, the generation interface provides an enumerated list of recognized conversions. The elements of the following enumeration follow Modula-3 semantics:

```
enum ConversionRoutines {cReal, cFloor, cCeiling, cRound, cTruncate,
     cOrdinal, cEnumerationValue, cLoophole, cCast, cComplex};
```

void **ExpTypeConversion** *(TypeID type, NameID routine)* $\succ$ $\lfloor$Expression e$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

void **ExpTypeConversion** *(TypeID type, ConversionRoutines cr)* $\succ$ $\lfloor$Expression e$\rfloor$ $\Rightarrow$ $\lfloor$Expression$\rfloor$

2.7 Compilation Units

All compilations begin with a compilation unit. In terms of an abstract syntax tree, a compilation unit is the root of the tree. Compilation units collect together information from outside the source program, which generally means source language information.

The generation interface requires that user code specify the source language of each compilation unit. Implementations may use this information to define language specific constants, types, routines, etc. Moreover, the source language information is used to preset attributes of the compilation unit. Programmers use the following enumeration to specify source languages:

```
enum LanguageId {cLangC, cLangCxx, cLangModula3, cLangFortran77};
```

void **CompilationUnitBegin** *(LanguageId l)* $\succ \lfloor\rfloor \Rightarrow \lfloor$CompilationUnit$\rfloor$

void **CompilationUnitEnd** *()* $\succ \lfloor * \rfloor \Rightarrow \lfloor$CompilationUnit$\rfloor$

At the point that the **CompilationUnitEnd** routine is called, the top of the stack (*i.e.*, the portion added since the **CompilationUnitBegin** call) should contain only UnitDecl nodes. Note that both begin and end routines return a CompilationUnit node, and this node is the same. The begin routine returns the partially complete node so that attributes may be associated with the node.

void **SetIdentifierCase** *(IdentifierCase ic)* $\succ \lfloor$CompilationUnit cu$\rfloor \Rightarrow \lfloor$CompilationUnit$\rfloor$

Implemenations of the generation interface are required to handle both case sensitive and insensitive identifiers. The default is case sensitive. We could have required the front end to homogenize case for case insensitive languages; however, this information may prove vital to debuggers.

```
enum IdentifierCase {cSensitive, cInsensitive};
```

void **SetMemoryManagement** *(MemoryManagement mm)* $\succ \lfloor$CompilationUnit cu$\rfloor \Rightarrow \lfloor$CompilationUnit$\rfloor$

This routine specifies how dynamic memory is managed by the source language.

```
enum MemoryManagement {cUserManaged, cGarbageCollected};
```

void **SetRecordFieldOrderRule** *(bool orderMatters)* $\succ \lfloor$CompilationUnit cu$\rfloor \Rightarrow \lfloor$CompilationUnit$\rfloor$

This routine specifies whether or not the source language uses the order of fields to distinguish record types. The default value is true, order does matter.

void **SetClassFieldOrderRule** *(bool orderMatters)* $\succ \lfloor$CompilationUnit cu$\rfloor \Rightarrow \lfloor$CompilationUnit$\rfloor$

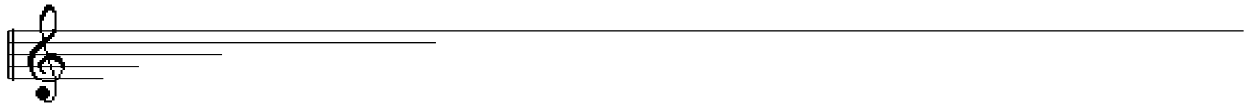
This routine specifies whether or not the source language uses the order of fields to distinguish class types. The default value is true, order does matter.

void **SetMethodsRule** *(bool methodsMatter)* $\succ \lfloor$CompilationUnit cu$\rfloor \Rightarrow \lfloor$CompilationUnit$\rfloor$

This routine specifies whether or not methods are used to distinguish class types. The default value is true, methods do matter.

2.8 Annotations

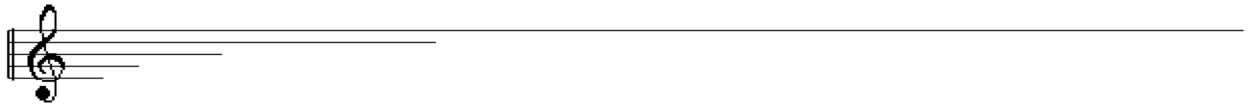
The generation interface allows user code to pass information that does not directly correspond to the structure of the program. We refer to this information as annotations. The interface currently provides two kinds of annotations: point annotations and range annotations. A *point annotation* applies to a specific piece of the program (*e.g.*, an expression or function). A *range annotation* may apply to several pieces of the program and is marked with a begin/end pair.



2.8.1 Point Annotations

A point annotation applies to the top element of the stack. A point annotation may modify the top element of the stack but does not (permanently) remove it.

void **SetSourceLine** *(int line)* $\succ \lfloor\rfloor \Rightarrow \lfloor \text{Node} \rfloor$



2.8.2 Range Annotations

A range annotation applies to all elements pushed onto the stack after the begin routine and before the corresponding end routine.

void **SourceFileBegin** *(String filename)* $\succ \lfloor\rfloor \Rightarrow \lfloor\rfloor$

void **SourceFileEnd** *(String filename)* $\succ \lfloor * \rfloor \Rightarrow \lfloor\rfloor$

Strictly speaking, the file name would not have to be specified at both the beginning and end, but doing so may help in debugging.

Bibliography

[1] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *The Modula-3 Language Definition*. Digital Equipment Corporation, Maynard, MA, 1995.

[2] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, Reading, MA, 1990.

[3] G. Weaver, K.S. McKinley, and C. Weems. Score: A compiler representation for heterogeneous systems. In *Heterogeneous Computing Workshop*, Honolulu, HI, April 1996.

Index