How To Write a GEN++ Specification

Prem Devanbu
Artificial Intelligence Principles Research Department
prem@research.att.com

L E Eaves
Object Oriented and Artificial Intelligence Technologies Group,
laurae@mozart.att.com

D-R-A-F-T
release 1.1, June, 1994

Contents

1 Introduction

In this tutorial-style manual document, we describe the use of GEN++, an application generator for creating code analyzers for C++ programs. GEN++ is based on a language independent specification language and generation system called GENOA [1] . This document is structured as follows: we begin with an illustrative example to explain the general structure of a language tool, and the basic notion of the *abstract semantic graph* representation of an input program. We then explain the basic fundamentals of GENOA, the underlying portable system that forms the basis of GEN++, and the important language independent notion of the *gtree*.

With that, we focus on the C++ implementation of GENOA, GEN++. First, we consider a simple example application for GEN++ that can find where variables are modified, and accessed in the body of a C++ program. We show the short GEN++ specification that implements the tool, and present the mechanics of building a running executable from this specification. We then continue by briefly describing the various constructs in GEN++, and then present a final tutorial examples. *If you're reading this, you are probably a friendly user; we are grateful for your interest in this software, and solicit your input. Feel free to contact* `mozart!laurae` *or* `research!genoa` *for help etc.*

2 HOW TO READ THIS MANUAL

Don't be put off by the length of this manual ! Almost half this manual is in the appendix, which is a description of how GEN++ views the C++ language. You can look up the appendix (starting on page 41) on a lazy (as-needed) basis. If you're a novice, and want to get a quick feel for the GEN++ language and write some interesting tools, try the following: First read § 3, 4, 5 Then skim through Section 6 to get a feeling for the various features of the GEN++ specification language. Then read the first page of Appendix A to understand how to read the tables there. Then, read through the tutorial in section 8; you should then be able to tackle the suggested exercises in § 8.5, and perhaps implement your own tools ! The rest of the manual deals with topics such as error messages, release notes, some idiosyncrasies of the parse tree built by CFRONT, etc. These can be skipped in the first reading.

3 Structure of a Typical Language Tool

A typical language tool processes source code, and produces some sort of output. The output might be some abstracted information (e.g., CIA [4] [3]), or a list of messages indicating violations of good coding practices (such as with LINT) or code in a different language (like CFRONT, which translates C++ programs to C).

The internal design of language tools, in most cases, is very similar; figure 1 illustrates the common elements found in most tools.

First, the input source file is converted into *lexical tokens* by a lexer. Tokens are strings that form the basic constituents of the source language, such as identifiers, integer constants, operators, etc. Next, this stream of tokens is processed by a parser, which groups tokens into higher level structures such as expressions, statements, declarations etc, and constructs an *abstract syntax tree* of the program. Next, a semantics processor modifies some parts of

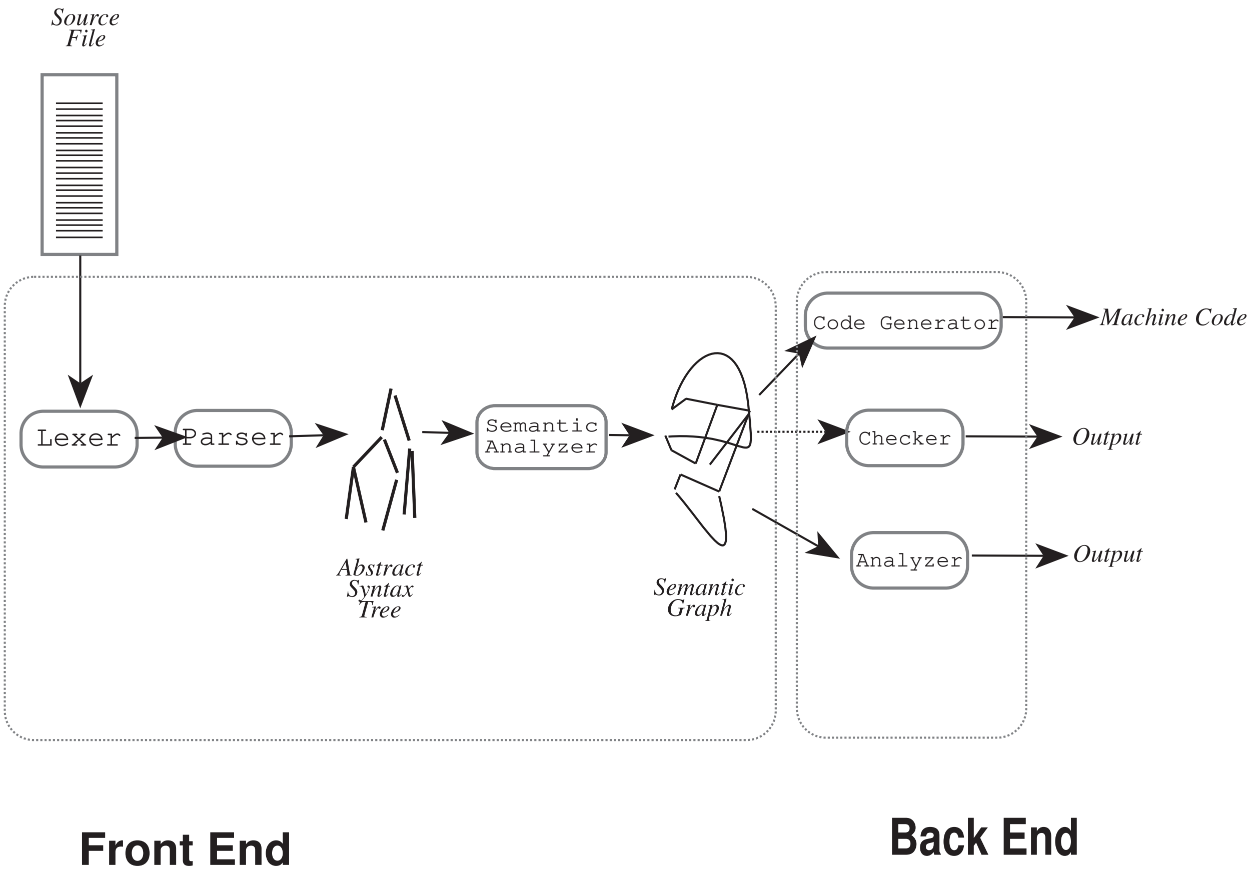


Figure 1: Internal structure of language tools

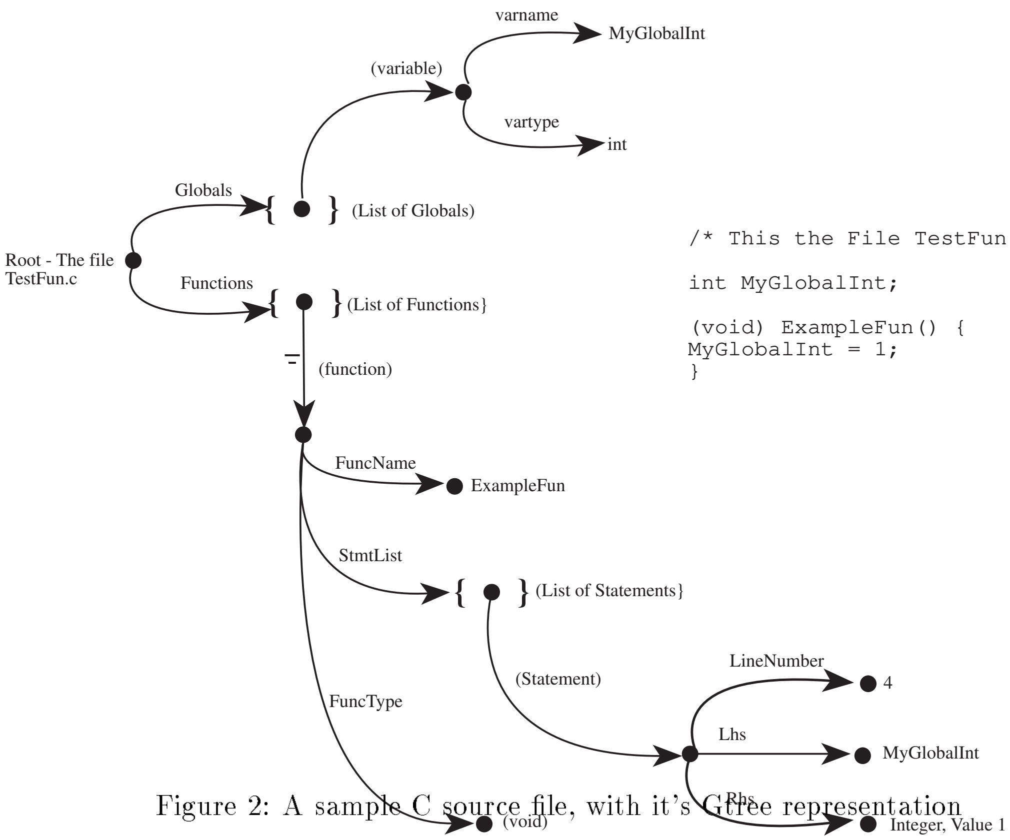
the tree by doing symbol resolution, type inference, etc, and produces an *abstract semantics graph* (ASG). This ASG forms the basis of most language tools; the particular tool's task, be it abstraction, checking or code generation, is performed on the abstract semantic graph representation.

The fact that most language tools (independent of language and intended task) work with some intermediate representation of the source code is the basic concept underlying GENOA.

4 Basic Concepts: gnodes, gtrees and traversals

GENOA is a tool for generating source code analyzers; it is designed to work with a range of different programming languages. This is made possible by adopting a language-independent view of source code, the *abstract semantics graph*. We refer to ASG's as *gtrees*[1], which are actually graphs made up of *gnodes* which represent the various program elements. Each node, has an associated type; the determines its structure.

[1]Admittedly a bit of a misnomer.




```
/* This the File TestFun.c */

int MyGlobalInt;

(void) ExampleFun() {
MyGlobalInt = 1;
}
```

Figure 2: A sample C source file, with it's Gtree representation

As an example, consider figure 2, which has a sample source file in C, and the corresponding gtree. Notice that the gtree is composed of gnodes, which are labeled with *types*. These types correspond to the nonterminals of the source language. For example, a gtree for C program can have nodes with types such as *function*, *assignment-statement* etc. Each node (based on it's type), has one or more *slots*, whose fillers are nodes of particular types, as determined by the syntax and semantics of the source language. For example, in C, a node of type *assignment-statement* has two slots, *lhs* and *rhs*, both

of which are filled by nodes of type *expr*. Slots may have just one node as a filler, or a list of nodes; for example, the *parameters* slot of a function is filled by a list of nodes of type *variable*. In the sequel, we refer to all the fillers of all the slots of a node collectively as *children*; *descendants* refer to all the nodes in the transitive closure of *children*.

Just as some non-terminals in a grammar can have several productions that can generate them, some types of gnodes can have several *variants*. For example, C, the node type *statment* has (among others) the variants *if-stmt, while-stmt, compound-stmt, goto-stmt*, etc. Contrariwise, we say that *statement* is the *base* type of *if-stmt, while-stmt, compound-stmt, goto-stmt*, etc. For any language, there is a set of node types, variants, slots and fillers; we invite reader at this juncture to glance at Appendix A, on page 49, where the list of node types, slots and fillers relevant to GEN++, the C++ implementation of GENOA are presented (if you have access to Mosaic, you may prefer to look at the files `typesummary.html` and/or `typedocs.html`). In addition to the node types corresponding to the terminal and non-terminals of a language, there are also nodetypes in GENOA corresponding to the "primitive" C datatypes `int`, `float`, `char *` and `FILE *` that are built in.

A source analyzer, as conceived by GENOA, runs over the gtree representation of a source program, looking for nodes of certain types, inspecting subtrees, gathering information, perhaps then proceeding to other specific parts of the tree, possibly looking for certain relationships, and of course, finally generating output. To implement such analyzers, GENOA provides a range of traversal, iteration, test, expression, and output constructs, which we describe in detail in section 6. In a sense, the GENOA specification language is a query language for parse trees; indeed, GENOA specifications are sometimes called "queries".

So far the discussion about analysis tools has been at a general, language-independent level; but from now on, we will focus specifically on GEN++ First, we present an example of a simple C++ tool, and illustrate the ease of constructing it with GEN++.

5 Building Tools With GEN++

In this section we illustrate the use of GEN++ by building an analyzer from a small sample query.

C++ programming tools are available which extract information about the organization of a program for example, class hierarchies, cross reference lists, or call graphs. At times, however, it would be useful to construct tools to perform more specific tasks.

The following GEN++ query locates name references and identifies whether they are used or assigned to. [2]

```
/* For each reference to a variable, print whether
it is used, or assigned to.
You'll find it in the file defuse.query in the shipped package */

ROOTPROC VarUseDef

PROC VarUseDef
ROOT CFile;
{
  [(?NameRef
     (IF (AND (TYPEOF $parent Assignment) (EQUAL $slot ``be_lhs''))
         (THEN (PRINT stdout ``At %s assignment to %s '' $location $token))
         (ELSE (PRINT stdout ``At %s access of %s '' $location $token))
     )
  )]

}
```

The full meaning of the query will become apparent after reading section 6. Basically, a search is made over all the nodes below the root node (of type `CFile`) of the g-tree representation of the program; when a node of type `NameRef` is found (a reference to a

[2]This example does not identify updates via increment operators `++` and . These require slightly more overhead. In addition, (theoretically) the exact result of pointer or array references cannot be decided by static analysis.

name), a check is made to see if it is the left hand side of an assignment. If so, the name reference is considered to be an "assignment to"; otherwise it's an "access". This query illustrates some of the compact, powerful features of the specification language; the toolsmith is able to specify tools in terms of high-level operations on the parse tree.

To build and run this tool using the enclosed software, simply place the specification in a file (say, `VarUseDef.query`) and run the following:

```
mk VarUseDef
```

This will create an executable tool, in a file named "`analyzer`" that performs the analysis specified.

This executable is actually based on a modified version of CFRONT (more precisely, the C++ Translator Release 3.0.1) which partially translates a program. A customized routine is then called that performs the specified operations upon the semantic graph. Since the analyzer accepts the same command line as CFRONT, it can be used in place of CFRONT in the CC shell:

`cfrontC=`*full_path_of_analyzer* `CC -F ...`

thus simplifying its use with existing makefiles. The release packages contain sample queries and a README file that describes the build process in more detail. Very often a custom tool can be constructed by simply adapting one of examples provided with the package.

6 The GEN++ Language

In this section, we present a more detailed exposition of the features of the GEN++ specification language. This specification language is not intended to be a general purpose programming language; it is designed to be used to specify tree operations. As such, the structure of the language, and the constructs provided, are "tree like". Specifically GEN++ adopts a lisp-like view of trees, where nodes and subnodes are represented a list.

A GEN++ program is usually a set of procedures. Each procedure is defined to work with a specific nodetype (*i.e.*, it starts its work on a node of a particular type), and perform a certain operation.

Some procedures are distinguished as "root procedures", using the `ROOTPROC` declaration. These are invoked, one after the other, on the Root Node of the parse tree. Each procedure is a list of constructs.

Constructs are typically lists enclosed by braces. There are different kinds of braces, (depending on the type of operation performed by the construct) which we describe below (§ 6.2.1). All constructs in GEN++ have an implicit notion of a *current node*, and "do their thing" with the current node. There are several kinds of *declaration* statements in GEN++, that serve to set up global namespaces, set up certain kinds of options for runtime, etc. We first describe these, then move on to describing the rest of the language.

6.1 GEN++ Global Declarations

In this section, we describe the global declaration statements in GEN++, and their function. As of the second release of GEN++, global declarations, except for PROC and FUNCTION declarations, **DO** end in semicolons. All the examples shown in the manual in this section do end in semicolons. However, for compatibility with previous release, the semicolon is optional.

6.1.1 FILE Declaration

The *file* declaration can be used to set up *output* files, and associate them with logical names; these files are all opened before a GEN++ analyzer begins to execute. The syntax of the file declaration is illustrated below:

`FILE` *file-id* "*File/Full/Pathname/in/Quotes*" ;

In the body of the analyzer code, all `PRINT` statements (See § 6.2.5) with an associated file-id FILE-ID will generate output to the given file. Optionally, a file may be specified to be usable for input; in this case, the file-id can be used with a `READ` expression (See § 6.3.2) to read values in. The syntax for an input file declaration:

`FILE INPUT` *file-id* "*File/Full/Pathname/in/Quotes*" ;

This command can also be used to simply declare a file-id, without specifically opening a particular file for input/output. This can be done later using the open command (§ 6.2.5). This syntax is as follows:

`FILE` *file-id* ;

or, for input files:

`FILE INPUT` *file-id* ;

The effect of a `FILE` command in (any form) can be duplicated by using an appropriate combination of `GLOBAL, ARG` or `LOCAL` declaration, in conjunction with an `OPEN/CLOSE` command pair, (see § 6.2.5, page 13) as illustrated below:

```
GLOBAL GNODE OutFile;
...
...within a procedure body
(OPEN OutFile ''/tmp/outfile'')
...
...
(CLOSE OutFile)
...
...
```

6.1.2 Include Declarations

GEN++ specifications are translated into C++ (also compatible with K&R/ANSI C) programs; it is possible to call C++ procedures as foreign (external) function calls from within the specifications (see § 6.4) for more details).

For this purpose, the user may specify header files to be included in the generated files. The syntax for doing this is:

```
INCLUDE "headerfilespec" ;
```

N.B. Note that the `headerfilespec` above is the usual header file specification, *as a quoted string*. The following examples should clarify:

```
INCLUDE "<fstab.h>" ;
```

will include the header file `<fstab.h>`. To include a file with a full path name, the quotes used must be escaped, as follows:

```
INCLUDE "\"/home/prem/myproj/myhdr.h\" " ;
```

N.B In the files generated by GEN++ (from any specification), the following header files are included by default: `stdio.h`, `strings.h`, and `stdlib.h`.

User defined objects or libraries to be included during linkage should be specified in the `$OTHERLIBS` variable in the `makefile` that is included with the delivered package.

6.1.3 Procedure and Function Declaration

A GEN++ analyzer can have several procedures and/or functions. Functions return values (GNODEs) and procedures do not. They can both be recursive. Such a declaration declares a procedure (or function), a *root node type*, an associated set of formal *arguments*, a set of *local* variables, and presents the *body* of the procedure, which is a list of constructs. Procedures can be separately promoted to be a *Root Procedure* which are invoked implicitly (functions cannot be promoted to be Root). Otherwise they are invoked as they are actually *call*ed (the `call` construct is defined below).

The *root* node type of a procedure merits some explanation. As we talked about earlier, every node in a `gnode` tree has an associated type (in C, they have types such as `if-stmt`, `Switch-stmt` etc.). As we shall see below, every operation is GEN++ is performed on a *current node*; often, the operations have to assume that the current node is of a certain type. The root node type declaration within the procedure declaration specifies the current node that the procedure can expect when it is invoked. This node is always the (implicit) first argument for any call to this procedure (See § 6.2.5). The typename in the declaration can be any of the node types declared in Appendix A (page 39). The root node type of a Root Procedure, in particular, is the node type `Cfile`. In addition, it is occasionally useful to write a procedure that performs the same function on a node, of any type (for example, printing it out in some form, or getting the location); in this case, the type is specified as "`G_AnyType`"

The syntax is as follows:

`PROC Procname`
argument declarations
`ROOT` *RootNodeType* ;
{
local declarations
constructs
}

The precise form of the argument and local variable declarations are described below in the variable declaration section. Constructs are described in the traversal and test constructs section. The top level constructs in the procedure have the same current node that is passed to the procedure. The syntax for a function declaration is identical, except that the prolog begins with "`FUNCTION Procname`". There is no need to specify a return type in the declaration, since it always a GNODE. Additionally, the `RETURN` statement can be used anywhere within a function or procedure. Both functions and procedures are invoked using the `CALL` (§ 6.2.5) command. However, it is important here to distinguish between the `CALL` to a procedure, which doesn't return a value, and is considered a command in GEN++, and a CALL to a function, which does return a GNODE, and can be used in an GEN++ expression (§ 6.3).

6.1.4 Variable Declarations

Variables are usually of three scopes: *global, local* and *argument*. These are specified using the keywords `GLOBAL`, `LOCAL` and `ARG` in the declarations. `GLOBAL` declarations occur at the beginning of the GEN++ specification file; `LOCAL` declarations occur in the body of `PROC` (see § 6.1.3, page 7) definition; the `ARG` occurs in the header of a procedure definition. Global variables have scope over the entire GEN++ program, independent of the procedures. Local variables are active only within the procedures within which they are declared. Arguments are place holders within a procedure body for parameters which are used in a procedure call. All variables used in GEN++ code should normally be declared "`GNODE`"; it is the only datatype used in GEN++. However, it is also possible to declare variables of the 3 main primitive datatypes, using `INTEGER`, `STRING` and `FLOAT` keywords; these variables can be used in C "escape" code that can be included into GEN++ specifications.

In addition, one can pass information in to GEN++-generated tools from the environment, as well as through command line options. There two other scope indicators: the *command option* scope and the *environment variable* scope. *The command line option scope is currently not functional; we're hoping to make this available soon. In the meantime, we suggest you use environment variables or the READ command (§ 6.3.2) to get information other than the standard* CFRONT *options into tools*

The environment variables need a type specification to make sure that they are converted properly from the ascii The type specification can be one of STRING, INTEGER, or FLOAT. The resulting variables are always of type GNODE, and can be used freely within the GEN++ code.

The *disposition* of a variable, `COMPULSORY` or `OPTIONAL`, is again only applicable to environment variables and command-options. Essentially, if a command option (or environment variable) specified `COMPULSORY`, is not present, the generated tool halts with an error message. Additionally, in environment variables and command line options, the name of the command line option or environment variable is also specified. The syntax of the declaration is illustrated below:

```
LOCAL GNODE Name1;
ARG GNODE FunLocate;
GLOBAL GNODE ArgStr;
GLOBAL Thieor; LOCAL NextPos; ARG LfunNm; COMMAND-OPTION ct INTEGER OPTIONAL IntVal;
ENVIRONMENT-VARIABLE vn STRING COMPULSORY VariableName;
```

The first four examples are trivial; the fifth one indicates that there is a command line option, which would be indicated by "-ct" which is of type integer; the "OPTIONAL" is a flag that indicates that this argument is optional (no error would be generated when the tool is invoked without a "-ct" argument); and the value of this argument would be stored in the global variable "IntVal". Likewise, the final declaration indicates that value of the environment variable "vn" is a string, which would be made available in the global variable "VariableName"; in this case, the "COMPULSORY" indicates that this environment variable is NOT optional–the tool will signal an error if this variable is not set.

6.1.5 Root Procedure Declaration

This is a simple declaration that indicates that a procedure is to be elevated to a root procedure; it will be invoked on the base node (or list of nodes) returned by the front end. The full details of the inter-operation between GEN++ and arbitrary front ends are described in the GENII documentation; we just describe it in short here.

A GEN++ generated analyzer works with a front end as follows: before any analysis routines are called, GEN++ invokes a CFRONT routine which parses and type checks the source file being analyzed. When the front end has built and annotated all it's data structures, (usually at the point where it would perform code generation) it returns to the GEN++ analyzer a pointer to the "root" of all the data structures it has built.

When the initialization routine has returned, the pointer(s) that it returns are built into a (list of) GNODE(s). At this point, the procedures that are declared to the root procedures are invoked in the order they are nominated to be root procedures.

The syntax of the root procedure declaration is as follows:

`ROOTPROC` *procedurename*

6.2 GEN++ Constructs

There are four main types of constructs in GEN++; *traversal* constructs, *test* constructs, *imperative* constructs, and *expression* constructs. We describe each of them in turn. We emphasize again that all GEN++ constructs work with an implicit current node.

6.2.1 Traversal Constructs

Traversal constructs in GEN++ move the current node to a new node or repeatedly move the current node over a set of nodes; they also specify a list of operations to be performed on the current node. These could be other traversal operations, output operations, expressions with side effects, etc. Here are the traversal constructs:

$\langle$*slotname ... list of operations ...*$\rangle$
{*typename ... list of operations ...*}
[... *list of operations* ...]
(*TRAVERSE typename expression operation*)

The first construct, the angular bracket, moves from the current node to the filler(s) of a specific slot. The slot name indicates a movement to the filler of the specified slot on the current node; the rest are a list of operations to be executed on that node. For example, consider the operation below:

```
<line ( PRINT ''Line Number %s'', $token) >
```

This construct moves to the line number slot of the current node, and makes that the current node; the print command then prints the line number out (the print command is described further below).
N. B: This construct works differently if the slot is a *list_of_properties* slot. For example, a class in C++ have a list of properties slot named `attrs`, whose fillers are class properties of type `ClassAttr`, such as `HasCtor`, `HasDtor`, etc (see Appendix A; node types are listed in alphabetic order). In each of these cases, the particular node may have slots that are applicable (only) in this case. So, if the `attr` property of a node has filler `CA_HasDtor`, then the node will have a filler for the `dtor` slot (which would be the declaration of a function that actually is the destructor. We may then have a query fragment such as:

```
<attrs
      (?CA_HasDtor
                 <dtor
```

The second construct, the curly braces, is a list iterator. This construct is used when the current node is actually a list of nodes; it executes all the constructs within it on every node in the list. The first identifier in the construct is the type of the node in the list. For example, consider the following operation executed when the current node is a function call node:

```
<callargs
         {Expression
                    (PRINT fileout ''Function argument %s'', $token)}>
```

The first angular bracket construct moves the current node to the arguments of the function; this node is a list node. Then we have a list construct which walks over members of this list (which are all of type `Expression`), and prints them out. The `BREAK` command (§ 6.2.5) can be used to exit from a list iteration.

The third and final traversal construct, the square bracket construct is an "all descendants" iteration construct; it executes all the operations contained within it on *every* node in the tree *below* the current node. For example the operation:

```
[ (ASSIGN count (+ 1 count)) ]
```

(assuming count is properly initialized) counts all the nodes in the parse tree. More realistic examples of the use of this construct are presented later on As in the case of the list iteration, the `BREAK` command (§ 6.2.5) can be used to exit from the square bracket "all descendants" iteration. For example the fragment:

```
[ (?Return (BREAK)) ...]
```

can be used to process all the nodes under the current node, exiting when a node of type `Return` is found.

N. B. Some times the slot on a nodetype is construct is *optional*, i.e., there does not have to be a filler for this slot. Examples include the `else` part of an `If` statement (the `ifFbranch` slot in GEN++) and `default` case of an `Switch` statement (the `default_s` slot). If the `default` case for a `Switch` does not exist, for example, the filler for the `default_s` slot is `NULL`. In such cases, an angular bracket construct that moves to the filler will not be executed at all. Thus, in following GEN++ specification fragment:

```
1 [ (?If
2     <ifFbranch (PRINT ... ) >
3 )]
```

when an `If` statement without an `else` branch is found, the construct on line 2, with it's embedded `PRINT`, will simply not be executed.
Now we describe the **TRAVERSE** command. This command used to be the *SEARCH construct*. For backward compatibility, the *SEARCH* keyword is still supported. A common idiom in a GEN++ query is to find interesting nodes in the tree, perhaps gather them in a list (see Page 17 for list operations such as `CONS, APPEND` etc) and then come back later and process them. Normally, the traversals are performed by moving the cursor around the parse tree built by the compiler. If it is desired to perform traversals over particular expressions, then the `TRAVERSE` construct is used. The syntax is:

`(TRAVERSE` *node-type expression traversal*`)`

This construct causes the *traversal* to be performed, using the value of *expression* as the current node. It is assumed that the type of the *expression* is of *node-type*. For example, suppose the GEN++ variable `statlist` has a list of nodes of type `Statement` (perhaps a list of statements gathered during a first pass, using some of the list operations on page 17). Now, to traverse each of these statements and print out only the locations of the `If statements`, we use the following construct:

```
1 (TRAVERSE
2          Statement statlist
3          {Statement (?If
4                         (PRINT stdout ``If Stmt at %s'' $location ))})
```

The `TRAVERSE` on line 1 causes the list iteration on line 3 to performed over the list of statements in the variable `statlist`

6.2.2 Test Constructs

There are two kinds of test constructs; the node type testing construct, and the conditional constructs.

6.2.3 Node Type Testing Construct

The node type testing construct is shown below:

(*?testnodetype* ...)

This construct is used to test the current node to see which of several different productions was used to generate it. The first identifier in the construct is the type which is being tested for; the rest are a list of constructs to be executed if the current node is of the specified type. For example if the current node is of type `expression`, the following operation might be executed on it;

```
(?FunCall
        <callname
                (PRINT fileout ''Function %s is being called'' $token)>)
```

If the current node is of type `FunCall`, this operation moves to the `callname` slot (the filler of this slot is the name of the function being called) and then prints out the function name. The combination of the square bracket search and the node type testing construct is a common and useful idiom: often, a search is made for a particular node type, whereupon further operations can be performed:

`[(?`*nodetype* ...*operations* ...`)]`

The variable definition/use example in Section 5 is based on just this idiom.

6.2.4 Conditional Constructs

The second kind of test construct includes the `COND` , `IF`, and `WHILE` constructs. The `COND` construct, is shown below:

`(COND ((test-cond) construct*)* )`

This construct is used to execute a series of test expressions (we describe the different kinds of test expression below, when we describe expressions) and then conditionally execute the list of operations associated with the first test expression evaluates to a true value. Note that the cond construct itself doesn't move the current node; however, one of the constructs contained within it might do so. This operation is illustrated below:

```
(COND ( (> StyleErrorCount 10)
       (PRINT fileout ''Your program has too many stylistic errors'')
       (CALL abort)))
```

This construct is checking a count of some kind of error, and prints out an error message, and aborts the entire analysis. Another kind of conditional construct is the `IF` statement. It's syntax:

```
(IF test-cond
    (THEN construct* )
    (ELSE construct* )
```

When the conditional expression returns true, the constructs in the THEN part are executed; if not, the ones in the ELSE part. The ELSE part is optional; however, because of the syntax used here, there is no ambiguity, so the IF constructors may be nested freely without confusion. The final type of condition command is the WHILE command.

```
(WHILE test-cond
       construct*
)
```

This command semantically equivalent to the C/C++ `while` command. The list of constructs `construct*` are executed while the condition `test-cond` is true. The BREAK command (See § 6.2.5) can be used inside the WHILE command.

6.2.5 Imperative Commands

There are several different kinds of imperative commands.

OPEN/CLOSE Command File-id's can be associated with actual files using the OPEN command. The syntax is as follows:

(OPEN *file-id Expression-of-type*-C_String)

The file-id used here must have been declared in an initial FILE command. The C_String must be an expression (See § 6.3) that evaluates a GNODE of type C_String. See the tutorial example on page 31 (§ 8.3) for an illustration of how to use this command to base output file names on input file names (*e.g.*, generating a file of name `frimble.out` from an input file named `frimble.c`). The OPEN command can also be used to open a file for input, thus:

(OPEN INPUT *file-id Expression-of-type*-mailC_String)

PRINT Command The PRINT command is a dead ringer for the C `fprintf` statement. The syntax is:

(PRINT *file-id "format string" arg1 arg2 ...*)

It works exactly like the C printf command, with one crucial difference; all format specifications for output are "%s". Regardless of node type, everything you print out in GEN++ is automatically de-parsed into a string at run time, using a default unparsing algorithm. This simplifies code generation, avoids type confusions and errors. The *file-id* argument may be `stdout`, `stderr` or a variable that was declared in a FILE declaration (See § 6.1.1).

As mentioned, PRINT displays gtrees by using default unparsing algorithms specified for each node type. Some nodes also have alternate print formats stored in special slots called "print strings". These slots have no fillers and can only be used to specify formatting information for printing. The syntax is shown below. If `fname` and `$token` are expressions of node type NameRef, and if NameRef has a print string `verbose`, then

```
(PRINT stdout "at %s:  %s references %s
n" $location
              <fname verbose> <$token verbose>)
```

would print the values using `verbose` instead of the default unparsing algorithm.

DUMP Command The `dump` command dumps out the g-tree representation into a file in a readable ASCII format. The syntax is:

`(DUMP` *file-id arg tabcount*`)`

The argument *arg* is an expression that is the root node of the gtree to be dumped; the argument tabcount is the number of tab indentations after which to start the dump; normally, this should be zero.

ASSIGN Command The syntax of this command is

`(ASSIGN variable expression)`

This command evaluates the second operand and assigns it to the first operand, which must be a declared variable. If it isn't, it's a syntax error; if the types of the declared variable and the expression don't match at run time (*e.g.,* if the variable is declared to be `INTEGER`, and the expression evaluates to a `STRING`, a run-time type error will be generated.

RETURN Command The syntax of this command is

`(RETURN expression)`

This command evaluates the operand and returns it as the value of a function. It can also be used without an argument for `PROC`s (which don't return values).

CALL Command The syntax of this command is

`(CALL functionname expression*)`

This command evaluates the operands, and calls the function *with the current node as first argument*, and the rest of the expressions in `expression*` as the rest of the arguments. Thus, there will always be one more argument than those of arguments listed explicitly in the call; the first argument passed is always the current node, and it will be root node of the procedure being called (See explanation in § 6.1.3)

This same syntax can also be used as a function call *expression*; This usage is allowed when the function named in the call is actually declared as a `FUNCTION` in the declaration. In case the `CALL` is to an external function declared with the `EXTERNAL` specification (See § 6.4), GEN++ will generate code to convert all the arguments into standard C++ data as specified in the `EXTERNAL` declaration; the returned value will be converted back into the appropriate GNODE, for use in GEN++ expressions. In the case of external procedure calls, the "current node is implicit first argument" convention does not apply.

BREAK Command The syntax of this command is

`(BREAK)`

This command immediately ends the execution of the `WHILE` loop within which it is contained, and continues at the statement following the loop.

EXEC Command This command an imperative statement to execute a raw C code fragment The syntax is shown below. If any value is returned, it is not used. (If it is necessary to return a value, an expression can be used in the CEXPR expression, with necessary type declaration to ensure proper conversion of GNODE data to GEN++ format. See § 6.3.1.)

`(EXEC` *"valid-c-statment"*`)`

The C statement that is given in quotes will simply be executed at this point. To pass a GEN++ value into the C code, any variables declared in the GEN++ specification (subject to scope constraints) can be used. However, to ensure proper type conversion from the GEN++ world into the C world, we provide 4 type conversion macros. They should only be used in C code fragments within quotes. Here they are:

Type Conversion Macro	From	To
`getCstring`	GNODE type `C_String`	`(char *)`
`getCint`	GNODE type `C_Integer`	`int`
`getCfloat`	GNODE type `C_Float`	`float`
`getCfile`	GNODE type `C_File`	`(FILE *)`
`getCvoid`	Any GNODE type	`void*`

Note the different conversions for the different built-in GNODE types. `C_File` nodes, which are for input output, also need to be converted. A common idiom where this is used is for flushing output files:

```
(PRINT fileout "..." ...)
(EXEC "fflush(getCfile(fileout));")
```

will force output from the `PRINT` to be written out immediately. The identifier `fileout`, a GNODE of type `C_File`, is converted to the `FILE *` type expected by `fflush`.

6.3 GEN++ Expressions

There are two kinds of expressions; conditional expressions, which are used only within `COND` and `IF` statements, and normal expressions, which can be used in assignment statements, print statements, etc.

Expressions are escapes, `READ` operations, `CALL`s literals, primitive or composed with expression constructs.

6.3.1 Escape Expressions

An escape is of the form

```
(CEXPR <node-type> ''C-fragment'')
```

The `node-type` is any valid GEN++ node type; (see Appendix A, page 39 for a list of node types) The `CEXPR` expressions can be used (with care) within any type of expression (conditional, composite or arithmetic); the GEN++ programmer should ensure that it is an expression returning the appropriate *type* of value. To use GEN++ values in the C fragment, the proper conversion macros should be used to ensure type conversion (see § 6.2.5).

6.3.2 READ Operations

The `READ` operation is a dead ringer for the C `fscanf` statement. The syntax is:

`(READ` *file-id "format string" arg1 arg2 ...*`)`

It works exactly like the C `fscanf` command, with one crucial difference; The *file-id* argument is one that was declared in a `FILE` declaration (See § 6.1.1, and it must be declared with the "`INPUT`" option. Any format declaration, such as `%d`, `%s` etc are allowed. The `arg1`, `arg2`, etc must be variable names; they will be set by the read command to be GNODEs of the type as determined by the corresponding formate specification (`C_Integer` for `%d`, `C_String` for `%s`, etc.)

6.3.3 Literals

Integers and string literals are permitted: `0, 234, ''foobar''` etc. Only integers should be used in arithmetic expressions (see below).

6.3.4 CALLs

A `CALL` can be used an expression to invoke a procedure, if the procedure is declared as a `FUNCTION`. If a procedure declared as a `PROC` is invoked from a `CALL` expression, type-incorrect C code will be generated by GEN++, and a tool cannot be built. (See § 6.2.5 for more details)

6.3.5 Primitive Expressions

There several kinds of primitive expressions in GEN++. They are usually of the form $*primitivetokenname*.

1. `$token` This primitive returns the gnode representation of the current node.
2. `$slot` This primitive returns the name of the slot whose filler the current node is. (the returned value is a GNODE of type `C_String`)
3. `$location` returns a node identifying the location of the current node in the source file. (In this case, locations are of of type `Location` and contain the file name and approximate line number where the current node occurs in the source file.) GEN++ processes `$location` constructs by searching in the current node and successive parent nodes until a node of type `the Location` is found. (See also pages 17, 20 and 40.) The location of an arbitrary node can be obtained by using the `LOCATION` construct. (See also pages 17, 20 and 40.)

4. `$nodetype` returns a GNODE of type `C_String` with a string representation of the type of the current node. (See also NODETYPE, page 17.) This can be saved and compared to the type of other nodes, or it can be printed.

5. `$parent` returns the parent of the current node.

6.3.6 Composite Expressions in GEN++

Composite expressions are constructs with arguments that return values.

`(SLOT slotname <expr> )` This expression returns the filler of the slot `slotname` of `<expr>`. If the slot `slotname` is not applicable to `<expr>`, then the expression returns an error.

`(LOCATION <expr> )` This expression returns the location (*i.e.,* a GNODE of type `Location`) of the node returned by `expr`. The location of the current node can also be obtained using `$location`. (See also pages 16, 20 and 40.)
`(NODETYPE <expr> )` This expression returns a GNODE of type `C_String`, corresponding to the node type of `<expr>`. Thus if `<expr>` is a while-statement, the above expression would return a GNODE of type `C_String` of value `"While"`. The primitive $nodetype (See § 4, page 17) is the same as `(NODETYPE $token)`.
`(SLOT-OF <expr> )` This expression returns a GNODE of type `C_String`, corresponding to the slot of `<expr>`. Thus if `<expr>` is a then-branch of an if-statement, the above expression would return a GNODE of type `C_String` of value `"ifTbranch"`. The primitive $slot (See § 2, page 16) is the same as `(SLOT-OF $token)`.
`(PARENT-OF <expr> )` This expression returns a GNODE corresponding to the parent of `<expr>`. Thus if `<expr>` is a expression on the LHS of an assignment, above expression would return a GNODE of type `Assignment` corresponding to the parent statement. The primitive $parent (See § 5, page 17) is the same as `(PARENT-OF $token)`.

GEN++ provides a list data structure. Lists in GEN++ are assumed to be of uniform type, *i.e.,* all members are the same type. The following are list expressions:

- `(NIL nodetype)` This expression returns an empty list of type "nodetype".
- `APPEND` This binary construct returns the concatenation of its two arguments; `(append list1 list2)` returns the concatenation of `list1` and `list2`.
- `NEXT` This unary construct that is *always* applied to a member of a list returns the next member thereof, unless the argument is the last member, in which case NULL is returned.
- `PREV` This unary construct that is *always* applied to a member of a list returns the previous member thereof, unless the argument is the first member, in which case NULL is returned. If `PREV` and `NEXT` are applied to an operand which is not a list element, an error will be generated.
- `LENGTH` returns the length of the list (a `GNODE` of type `C_Integer`).

- `CONS` This binary construct adds a node to the head of a list. Thus If variable `A` is the list of type `String` with two elements `(''func2'' ''func3'')` and the variable `B` is of type String with the value `''func1''`, then the expression `(CONS B A)` has the value `(''func1'' ''func2'' ''func3'')`. *N.B* `CONS` is to add a node to a list; to concatenate two lists, use the `APPEND` operation.

- `(LIST-OF nodetype arg1 arg2 ...)` returns a list of nodes of type nodetype containing elements arg1 ...

- `CAR` and `CDR` are analogous to the LISP operations of the same name. `(CAR list)` returns the first element of a list; `(CDR list)` returns the list with the first element removed.

- Last but not least, (`LAST` returns the last element of a list. If the list is empty, this command returns `NULL`.

Note also the conditional expression (`EMPTY` ¡expr¿) (which is listed in the next section.

6.3.7 Conditional Expression

These expressions are used to specify test conditions in `IF` and `COND` statements.

`(EQUAL expr1 expr2)` Returns true if the two expressions are equal. They can be single nodes or lists. GEN++ first tests if they are the two are of the same type; if so, then they are tested to see if the represent the same value in the front end.

`(NULL expr)` Returns true if the expression `expr` is NULL. For example, `(NULL (SLOT dtor CLASSDECL))` would return "true" if the variable `CLASSDECL` was a gnode that represented a class that had a no destructor.

`(SAMETYPE expr1 expr2)` returns true if the two expressions are of the same type. *e.g.* if one expression is of type `assignment`, and the other is of type `goto`, this returns false.

`(TYPEOF expr nodetype)` returns true if the expression of the specified type. Another way to test the type of a node is `(EQUAL $nodetype ''typename'')`

`(MEMBER-EQUAL` *thing list*`)` returns true if "thing" is a member of the list "list".
GEN++ also provides a way of testing whether a list is empty

`(EMPTY` *expr1*`)`

N.B The `EMPTY` command is different from the `NULL` command. The latter tests to see if a value is `NULL` (*e.g.,* the filler of a slot). The former is used to test if a list is empty.

`(OR` *condition1 condition2*`)` returns true if one of the subexpressions does.

`(AND` *condition1 condition2*`)` returns true if both subexpressions return true.

`(NOT` *condition1*`)` returns true if the subexpression returns false, and false otherwise.

GEN++ also provides arithmetic conditionals. They are listed below, without comment; they have the standard meanings. They should be only applied on expressions of type

Integer; (*e.g.* lengths of lists, or the line numbers of files etc.) all other uses will result in an error. Naturally, that the equality construct listed above will work on integers as well.

(> *expr1 expr2*)

(< *expr1 expr2*)

6.3.8 Arithmetic Expression

GEN++ also provides an entire range of arithmetic constructs. They are listed below, without comment; they have the standard meanings. They should be only applied on expressions of type Integer; (*e.g.* lengths of lists, or the line numbers of files etc.) all other uses will result in an error.

(+ *expr1 expr2*)

(- *expr1 expr2*)

(* *expr1 expr2*)

(/ *expr1 expr2*)

6.4 Calling External Functions

GEN++ was deliberately designed to be a very simple language, with a limited number of data types, built in expressions and compact, simple control structures. It is not meant to be a general purpose programming language. It is therefore expected that a user may want to interface to other languages to to take advantage of richer type structure and other features. A simple way to do this is to send output from GEN++ to other tools.

Another possibility is to send data off to auxiliary functions, by directly invoking them from a GEN++ query. To be able to do this in a consistent fashion, there is a facility to provide foreign function linkage information at the beginning of a GEN++ query file, along with the `FILE` and `ROOTPROC` declarations. As an example, consider a function `MyStrConcat`, with an ANSI signature of `char* MyStrConcat(char *,char *)`; it would be declared in this fashion:

```
EXTERNAL PROC MyStrConcat (STRING STRING) RETURNS STRING
```

If no value is returned, say for example with a function `MyProc`, of signature `(void ) MyProc(int, char *, void *)`, we declared it this form (the `RETURNS` phrase is omitted):

```
EXTERNAL PROC MyProc ( INTEGER STRING ANY)
```

With these declarations, foreign functions may be invoked with the usual `CALL` construct. Of course, if a value is returned, they can be `CALL`ed from within an expression. In order to complete the linkage, the object (or library) file containing the definitions of these must be inserted into the `$OTHERLIBS` variable in the makefile, to ensure that all the names are resolved.

Normally, the GEN++ system generates the C++ language `external` declaration for foreign functions. Sometimes, an external function (or procedure) is actually declared in a header file that is included (see § 6.1.2 for information about `INCLUDE` commands) In this case, the GEN++ `EXTERNAL` declaration can specify a `DECLARED` option to turn off the generation of the C++ `external` declaration, thus:

```
EXTERNAL DECLARED PROC strlen ( STRING) RETURNS INTEGER
```

It is important that the `EXTERNAL` declaration be included, *even for functions declared in header files*. This information is used by GEN++ at run time to ensure that the types of the data being passed are truly matched up with what the external procedure expects.

7 What's In a Parse Tree?

Before writing queries, it is necessary to have some understanding of the "parse trees" being queried. The GENII specification defines the general structure of the trees (See the appendix, or the Mosaic files `typedocs.html` and `typesummary.html`). However, this does not provide complete information about the transformations done by the front end during semantic processing. This is because the parse tree is actually in mid translation and therefore may not match the input program syntactically. The gtree represents an "edited snapshot" of information in front end data structures after type checking and some transformations have been done. The contents of gtrees, and the interface definition may vary with different implementations.

A GEN++ gtree (fortunately) does not reflect all the details of CFRONT internals, much of which has been hidden in the interface definition. The GENOA DUMP command can be used to display gtrees for specific inputs:
`(DUMP` *file-id arg tabcount*`)`
where *tabcount* is the number of units to indent subitems in the display. Output is an indented prefix display of the gtree represented by *arg*.

The following sections contain a smattering of general information about the gtrees in this implementation.

7.1 Locations

(See also pages 16, 17 and 40.) The special id `$location` in a GENOA specification refers to the approximate location (in this case, file + line number) of the current node. The current location is determined by looking in the current node and successive parent nodes for the nearest location specifier. In GEN++, locations have type `Location`. For example, if the current node is an expression node, then the GEN++ construct

```
(PRINT stderr "here I am!:  %s" $location)
```

would print the location of the nearest enclosing statement, since expressions don't contain locations in this specification. Statements and declarations contain location slots. The location slot for a file is the location of EOF.

The Appendix also contains additional notes on the use of `$location`.

7.2 Some Transformations

The GENII specification in the appendix describes the node types and "slots" or members that define the gtree processed in GEN++ queries. This gtree resembles the input program in structure with some modifications. This section lists some of the transformations made on the original parse tree during semantic processing. NOTE: This list may be incomplete or subject to change. The interface specification in GEN++ is not a comprehensive representation of C++, but is designed as a convenient interface for accessing commonly needed information.

- Linkage specifications such as

  ```
  extern "C" {
      // declarations
  }
  ```

  have already been processed and incorporated into the declared function types. (See `FunctionType`, slot `linkage`.) Therefore the syntax is not represented in the tree.

- Implicit casts and conversions are inserted in expressions. This includes insertion of C-style casts as well as implicit calls to constructors and conversion operators.

  Note that there are various flavors of casts in a gtree. C-style casts are represented as `Ccast` nodes. A `Ccast` may be either a `UserCast` (explicit int he source code) or a `GeneratedCast`. These are different from conversion operators and constructors, which are represented as various flavors of function calls. This should be obvious from the specification. (See the appendix.)

- Overloaded operators are invoked as "generated function calls" (`GeneratedCall`) and the identities of overloaded function names are resolved.

- Constructors have been invoked in initializations. Initialization expressions can be found in slot `init` in applicable node types, such as `ObjectDef`. Destructor calls have not been processed, however. See below.

- Default arguments have been inserted in function calls. For example, if function `f` is defined as

  ```
  void f( int, char* = "fizzle", int = x );
  ```

  then the call

  ```
  f(3);
  ```

  will be represented by

  ```
  f(3,"fizzle",x);
  ```

in the gtree. There is currently no indication of which arguments were explicit in the source and which have been inserted. This must be determined by comparison with the source program.

It should also be noted that default arguments accumulate over declarations. For example, if a subsequent declaration of "f" appeared with no default arguments, the gtree would still contain the default arguments as if they were supplied explicitly:

```
void f( int, char*, int ){} // "fizzle" and ::x implicit
```

Thus, they will be found in global searches.

Note that the insertion of default argument expressions in function calls can result in unexpected output from some queries. For example, the implicit reference to name "x" above would be treated as an explicit reference in the source program.

- Implicit blocks may be inserted to preserve C-style block structure when a declaration appears after executable statements. For example,

```
{ int i = 3; ++i; int j = 4; }
←
{ int i = 3; ++i; { int j = 4; } }
```

- Different types of function calls currently have different forms in the gtree, reflecting idiosyncrasies of cfront's implementation. For example, the gtree for a virtual call may look somewhat different from the tree for a non-virtual call. (Note that virtual calls have not been transformed into low-level expressions using virtual tables, however, the type slots and other aspects of the gtree vary for different cases.) Function overloading also affects the information in gtrees representing function calls. An attempt has been made to preserve commonly-used information (such as function name) in a prominent place in the `Call` expression so it can be accessed easily and predictably. The DUMP command can help to identify other subtle differences between call expressions that may affect certain queries.

- Some complex rewrites require generation of temporaries, which may change the shape of some expressions. While most all temp generation is suppressed in gfront (most temporaries being produced later in the compilation), some may still occur, particularly in expressions containing implicit constructor calls.

This can affect the results of queries what make detailed assumptions about the structure of certain subexpressions. Attempts have been made to embed as much complexity as possible in the implementation of the GENII interface, and to "turn off" transformations on the parse tree that cause information to be lost or obscured. (For example, see below.)
NOTE: If you encounter an unexpected rewrite that causes a query to fail, please report it. Either the rewrite can be modified for GENOA, or details of working around them can be hidden in the GENII interface.

The following operations have NOT been performed when the GENOA queries are run. These transformations were

- inline expansion,
- most, if not all, automatic destructor invocations,
- virtual function call rewrites and low-level arithmetic for pointer-to-member access. (Note: All function calls preserve the function name so that references can be identified in queries.)
- inserting executable code for initializations. For example, assignments or constructor calls for local variables are not inserted in blocks, as is done during normal translation.

```
{ int i = 3; X obj(27); ...}
```

becomes

```
/* condensed DUMP output */
Block
     <locals>{
             ObjectDef ``i''
                   <init> 3
             ObjectDef ``obj''
                       <init> obj.X(27)
     }
     <body>{
           ...
     }
```

The assignment

```
i = 3;
```

and constructor call

```
obj.X(27);
```

are not in the body of the block. To access initializers, it is necessary to walk through the local declarations and check the `init` slot of each `ObjectDef`. If a constructor is invoked for a declaration, the initializer (or `init` slot) will be rewritten to incorporate it. This is also true for global variables. I.e., initializers can be obtained from the declaration; it is unnecessary to walk through the `__()` function.

- implicit destructors are not invoked.

7.3 Parameterized Types

Parameterized types and functions introduce templates and instantiations into a program's source tree.

A template is an untypechecked, generic declaration of a class, function or data member. Formal parameters are placeholders for types and values to be supplied when the template is instantiated.

```
// class template
template<class T, int i>
class C {
    T* t;
public:
    C() { t = new T[i]; }
};
```

A template declaration cannot be typechecked until it is instantiated (provided with actual parameters), since the types of references to these parameters is unknown. Thus only limited information can be inferred.[34]

```
// instantiation of class template C
C<int,3> obj;
```

Instantiated templates are fully typechecked. If an instantiation has an explicit, user-supplied definition, it is called *specialized*:

```
// specialized definition
class C<int,3> {
    C() { funny_optimization(); }
};
```

Otherwise the definition is inferred from the template declaration, and is implicit.

[4]The current implementation supplies only the name of the template and its formal parameters. The body of the template is not currently represented int he GENII interface spec.

7.3.1 What's In the Gtree?

All information available in CFRONT's symbol table is accessible in the derived gtree, even entities for which no code would be generated in a compilation.

Note: The instantiation mechanism in a given compilation system is implementation dependent. In the C++ translator environment, if a given instantiation (`C<int>` is referenced in multiple source files but not defined explicitly in the program, the linker attempts to instantiate a single copy by recompiling parts of the source to generate an implicit definition, which is then linked with the program. However, the complete definition for `C<int>` is available in cfront's symbol table for each source file that references it, since `C<int>` must be instantiated in order to compile the file. Therefore, the the corresponding definitions are also available in the gtree.

Explicit declarations (template and specialized templates) will appear on the list of user declarations (for example, `CFile||globals`, `MemberDeclarations`, or locals). The following node types are defined for templates:

```
Template /* a template declaration */
    DataTemplate
    FunctionTemplate
    ClassTemplate
Parameter /* formal parameter in a template declaration */
    TypeParameter
    ValueParameter
Actual /* actual parameter in an instantiation */
    TypeActual
    ExprActual
```

Instantiations (whether specialized or implicit) are treated like other declarations, except that they are marked as instantiated in their attributes. For example, instantiated classes have attribute `CA_Instantiated`. The list of actual parameters can be obtained via

```
CA_Instantiated||actuals
```

For functions, the actual parameters can be obtained through

```
FA_Instantiation||actuals
```

Example:

```
class<class T> struct S { T t; };
S<int> si;
S<char*> sp;
```

results in three separate class definitions in the global symbol table: the class template S, and two instantiations `S<int>` and `S<char*>`, all at the same location. `si` and `sp` are also defined with types `S<int>` and `S<char*>`, respectively.

7.3.2 Accessing Parameterized Types Through GEN++

The current specification supports access to class, function and data templates, formal parameters, and actual parameters in instantiations. The declaration of an instantiated template does not appear explicitly in lists of `Declarations` for example,

```
CFile||globals
Block||locals
AggrDef||members
```

This is because it is not explicitly defined by the user (unless it is a specialization). However, the implicit `Declaration` can be accessed indirectly through the type:

```
AggrType||refname||def
```
[5]

for classes, or through

```
NameRef||def
```

for functions and data. Actual parameters can be accessed through `ClassAttr`

```
CA_Instantiated||actuals
```

or FunctionAttr

```
FA_Instantiation||actuals
```

Data templates are always class template members. Therefore the actual parameters can be accessed through the qualifier:

```
C<actuals>::member
```

In the parse tree, this will appear as a `NameRef` 'member' with a qualifier '`C<actuals>`', which is also a `NameRef`.

[5] The GENOA operator `||` is analogous to `->` in C++.

7.4 Known Bugs

- Templates are partially supported in the specification, but not all information in a template definition is accessible from the GENII spec. This is because the template definition has not been type checked, and the GENII specification does not currently support untypechecked expressions, which have a slightly different representation in the front end.)

- The unparsing specifications in GENII now uses slightly modified versions of CFRONT's pretty-printing routines used in error diagnostics. Output is accurate, but may not always be legal c++. Therefore, unparsing should not be used to generate code. Later versions may correct this.

- Since source code has been partially translated, some types and attributes may not match those in the source program. For example, return types for constructors and other special functions may reflect the actual type in the generated code:

```
struct X {
    X(); // const X* X::X()
    X̃(); // void X::X̃()
}
```

  Other translations, such as inserting implementation dependent arguments, have not been done. In essence, the information included in a gtree is sometimes ad hoc and inconsistent where compiler-generated information constructs are concerned. But enough useful information is included to implement most common queries effectively.

  CAVEAT: The GENII specification in GEN++ is not meant be a complete, consistent, well-defined representation of C++. For this, see systems such as Alf/Grail [2] .

7.5 What GEN++ Doesn't Do (yet?)

The following are known deficiencies of GEN++, which may or may not be remedied in the future.

- GEN++ currently doesn't support queries on preprocessor information. This will require merging the front end with a preprocessor. Extensions would also be necessary in the GENII spec to refer to preprocessor constructs. (Prototype exists...)

- GEN++ runs in batch mode only. A GEN++ interpreter with a friendly user interface has been considered to allow queries to be run interactively. This would expand the target population to include sophisticated end users.

- Due to internal incompatibilities introduced in cfront to support GEN++, GEN++ does not generate code. Code generation would allow queries to be run simultaneously with compilation. It could also be used to support tools which instrument or transform code.

- The current implementation of GEN++ is based on the C++ translator. Thus does not provide persistence. It would also be possible (and in fact, more natural) to implement a GENOA-based tool generator on top of Alf. The alternative would be to add persistence to cfront.

- GENOA supports simple I/O and a handful of utilities to interface to external functions and data. However, its own data structures are limited; the intention is that external routines (written in C) can be called to do other more complex operations. GENOA itself is primarily designed to be a query language for parse trees.

8 A Quick Tutorial

8.1 Tutorial Example 1: A Validation Tool

We now present a quick step-by-step description of how one goes about writing a specification for an analysis tool. As a first illustration, let us consider the following coding guideline:

In this body of code, switch statements **must** *always contain a default case.*

This can be implemented by a simple GEN++ query. First, one must set up a root procedure, and identify it as such, and declare an output file; the root procedure definition is begun, and it's root node declared to be of type `CFile`:

```
FILE badsw "Switch"
ROOTPROC badswfind

PROC badswfind
ROOT CFile;
{
...
}
```

Next, we first have to look in the table in Appendix A, and identify what the exact name of the switch statement type is; in fact, it is `Switch`. It's evident that we have to conduct a global search for occurrences of Switch:

```
FILE badsw "Switch"
ROOTPROC badswfind

PROC badswfind
ROOT CFile;
{
  [
    (?Switch
    ...
```

```
    )
  ]
}
```

We have to look at the default case of the switch, and make sure there is one; by inspection of the `Switch` type, we see that there is a `default_s` slot. If a filler for this slot exists, then we have a default for a `Switch` statement. We first declare a local variable, `hasdefault`, which we initialize to `0` whenever we find a switch statement. Then, we use the angular bracket construct to move to the filler of the `default_s` slot. If this slot doesn't exist, then the constructs inside the `< default_s ...>` will not be executed; so the value of `hasdefault` will remain at `0`. We can test for this to see if there is no default:

```
FILE badsw "Switch"
ROOTPROC badswfind

PROC badswfind
ROOT CFile;
{
  LOCAL GNODE hasdefault;
  [
    (?Switch
      (ASSIGN hasdefault 0)
      <default_s
       (ASSIGN hasdefault $token)>
      (IF (EQUAL hasdefault 0)
          (THEN (PRINT badsw "Switch at %s doesn't have a default case" $location)))
    )
  ]
}
```

8.2 Tutorial Example 2: A Metrics Tool

Let us now look at a simple metrics type example.
For each function in this body of code, print out the number of references to global variables
For the purposes of this query, let's assume that all function definitions appear at the top level (*i.e.,* function definitions aren't contained inside other function definitions).

```
ROOTPROC GlobVarRef
PROC GlobVarRef
ROOT CFile;
{
  LOCAL GNODE fname;
  LOCAL GNODE gcnt;
  <globals
   {Declaration
    (?FunctionDef
     (ASSIGN fname (SLOT id $token))
     (ASSIGN gcnt 0)
     [(?IdName
       <scope
        (?Scope_Extern (ASSIGN gcnt (+ 1 gcnt)))
        (?Scope_Static (ASSIGN gcnt (+ 1 gcnt)))
       >
     )]
    (PRINT stdout "Func %s, %s global references" fname gcnt)
    )
   }
  >
}
```

As before, the query starts at the root node, of type `CFile`. From here, we use the angular bracket construct to move to the `globals`. This is a list of nodes of type `Declaration`. We use a list iteration: ({`Declaration ...`}) to iterate over these declarations. For each of these, we test if it's a function definition, using the `(?FunctionDef ...)` construct. When we find a function definition, we save the filler of the `defname` slot, using the `SLOT` expression (see § 6.3.6, page 17). in the local variable fname. Also, we now initialize a variable `gcnt` (to count the number of external name references) to zero. Now, we start a global search of all the nodes below the function definition node, looking for name references. When we find name references, we have to determine their scope. When we look up the node type `IdName` in the appendix we find that it contains no scope information, however, it is derived from node type `NameRef`, which contains a slot `scope`. Therefore, slot `scope` is actually inherited from `NameRef` and can be used as if it were a slot in `IdName`. Now, the filler of slot `scope` is of type `Scope`, which includes `Scope_Extern` and `Scope_Static`. These are the node types we need for testing if the scope of a name reference is global. So, in the query, we move to filler of the `scope` slot of the node of type `IdName`. We test to see if this of type `Scope_Extern` or `Scope_Static`. If it is, we increment the counter for global name references from the function whose name we stored away in `fname`. When the global search over the descendants of the function definition node is complete, we print out a global name reference count for that function.

8.3 Tutorial Example 3: Naming Output Files

In all the examples shown so far, the queries have generated outputs in a file of the same name, irrespective of what the input source file name was. In the following query, we illustrate how to generate an output file name from the input file name. For this, we `OPEN` command (§ 6.2.5, page 13) in conjunction with the `CEXPR` (§ 6.3.1, page 15), expression. For this purpose, we only show a fragment of the query that immediately applies to this particular task. For an actual example, see the query `hier.query` in the examples shipped with this package. In this case, the output file name will have the same name as the input file, except, the extension will be "`.out`" (instead of "`.c`"). Thus, an input file named "`frimble.c`" will produce an output file named "`frimble.out`".

```
ROOTPROC NamedExt
FILE fout;

PROC NamedExt
ROOT CFile;
{
  LOCAL GNODE fname;
  <filename (ASSIGN fname $token)>
  (OPEN fout (CEXPR C_String "GenFixExt(getCstring(fname),\"out\")"))
  ...
  ...
  ...
}
```

As described in § 6.2.5, the `OPEN` command takes a string argument. In the above example, we save the name of the file (which is of type `C_String`) in the variable `fname`. Then, we use the built-in function getCString (See § 6.2.5, page 15) to convert the GNODE into a `char *` pointer, and pass it to a function `GenFixExt`, with another string argument. The function `GenFixExt` is a built-in GEN++ library function (with signature `char *GenFixExt(char *, char *)`) which returns a string consisting of the first argument, except with the last character replaced by the second argument (*e.g.,* `GenFixExt("tumbler","down")` would return the string "`tumbledown`"). In this case, `GenFixExt` retuns the file name with the "`.c`" replaced by "`.out`". This value is converted into a GNODE of type `C_String` by the `CEXPR` command, and passed to the OPEN command, which opens the file of the appropriate name (in this case, for output) and assigns it the file-id `fout`.

8.4 Tutorial Example 4: Backwards and Forwards Iteration

Sometimes it is necessary to iterate backwards or forwards from a particular position in a list (*e.g.,* a list of statements in a block). Here are some GEN++ fragments that illustrate how to do this. First, we take the first `If` statement in a block, and iterate forth (and then back) from it:

```
...
...
(?Block
 <blockbody
  {Statement (?If (ASSIGN this $token) (BREAK) )}
  (ASSIGN forth this)
  (WHILE (NOT (NULL forth))
          (PRINT ‘‘Statement at %s’’ (LOCATION forth))
          (IF (TYPEOF forth Return) (THEN (BREAK)))
          (ASSIGN forth (NEXT forth)))
...
...
  (ASSIGN back this)
  (WHILE (NOT (NULL back))
          (PRINT ‘‘Statement at %s’’ (LOCATION back))
          (ASSIGN back (PREV back)))
...
... > )
```

First, above code fragment searches for `Blocks`, and when it finds one, it searches for the first occurring `If` statement in the body of the block; as soon as it finds one, it saves in the variable `this` and breaks out of the list iteration over the statements in the body of the block. Then, we first iterate forward starting at this statement, and then backward. In the forward iteration, we illustrate how the loop could be conditionally terminated early.

8.5 Suggested Exercises

- Look at the query file `hier.query` in the examples directory. Explain how it works. (*Note:* This tool a graph of the inheritance hierarchy of a C++ program).

- Look at the query file `metrics.query` in the examples directory. Explain how it works. (*Note:* This query computes some connectivity metrics for C++ classes.)
- Can you think of a simple modification of Example 2 that will count the number of statements in a function. How about the number of `If` statements ? Or the number of looping (whiles, dos etc) (*Hint:* Use an `OR` of `TYPEOF` conditions).
- Create a tool to search for function definitions, and make sure that the functions are not are not both inline and virtual. This is a coding standard in some projects.

9 Errors and Warnings

There are 2 types of messages that relate to GEN++: generation time (when an analyzer is generated) and run time (when the analyzer is run).

9.1 Generation Time Errors

While generating an analyzer form a specification, GEN++ may generate several warnings and or errors. We list these below, along with possible remedial actions. Typically, errors will abort analyzer generation; warnings will not.

1. A very common error is of the form:

   identifier `is not a valid type name.`

   This is caused when an invalid typename is used any otherwise valid GEN++ construction that needs one, such as the `TYPEOF` conditional expression, or the `(?`*typename* `...)` construct, or the `ROOTTYPE` *typename* declaration (for procedures) etc. *Remedy:* check the name you used against the list in the appendix (which is in alphabetic order) for spelling/capitalization errors.

2. Another error message is

   `Procedure name,` *identifier* `is a type name, choose another.`

   or,

   `Variable name,` *identifier* `is a type name, choose another.`

   Type names are reserved words in GEN++: in the generated C source, the typenames are part of an enumerated type. Since any other use of these names would be offensive to the C compiler that compiles the generated C source, such usage causes a fatal error message.

3. Improper use of slots in an expression such as `(SLOT` *slotname expression*`)` or a construct such as `<`*slotname* `...>` results in the following error:

   slotname `is not a valid slot on node of type` *nodetype*

   For example, a query such as

   ```
   (?BinaryExpr
           <globals ...>)
   ```

would cause the above message; the node type `BinaryExpr` doesn't have a slot named `globals`.

There is a related warning, (not error) which still allows code generation to go through, but could cause run time errors:

slotname `is not a slot on node type` *typename*`, but on a derived class`

This error happens when a GEN++ spec uses a slot which occurs not with the node type of the current node, but only a derived type. Thus, the specification may work correctly, but could sometimes cause a run time error. For example, consider the following specification:

```
1 (?Cfile
2     <globals
3         {Declaration
4             <returntype ...> } >)
```

On line 2, we move the current node to the fillers of `globals` slot; we can see (from the description of `Cfile`) in the appendix; that the fillers are of type `Declaration`. Line 3 is an iteration over this list of `Declarations`, and line 4 moves to the filler of `returntype` slot of each `Declaration`. From an inspection the `Declaration` node type and its variants in the appendix, it's clear that the `returntype` is available only the `FunctionSpec` variant of `Declaration` and not on `Declaration` itself. This spec would give rise to the abovementioned warning. If the generated analyzer were run on a file that had nothing but function specifications, there would be no run time errors. If other types of declarations such as global variable declarations, typedef etc where found, a run time error would be generated, since none of these have a `returntype` slot . *Remedy:* Using the node type tables in the appendix, ensure that the node types and slots are being used consistently. If possible, embed the offending construct in a `(?`*nodetype* `...)`, or by using a `IF` construct with a `TYPEOF` condition to ensure correct nodetype is being used.

4. While calling external functions that return values (See § 6.4) only the basic types `INTEGER`, `STRING` and `FLOAT` can be used. The of the `ANY` type causes the following error:

   `The return type can be only one of INTEGER, STRING, or FLOAT, sorry !`

9.2 Run Time Errors

In this section we, describe errors that may result during the running of a GEN++ generated tool.

9.2.1 Initialization Errors

These errors are generated if some aspect of the initialization of a GEN++ analyzer fails. The error depends on the stage of the initialization. Errors/Warning could happen during the compilation of the source by C++; then, during processing (by the analyzer) of environment variables or command line options, or during the opening (by the analyzer) of files for input/output.

1. During the processing by CFRONT, of the input C++ source code, any errors generated by CFRONT are printed out.

2. During the processing of environment variables or command line parameters, the following messages may be generated:

   `*** Error *** The Environment variable` *varname* `is not set`

   Remedy: make sure the variable is properly defined and exported.

   Command line options are currently not functional. This will be remedied in a subsequent release

3. Files declared in the `FILE` declaration will be opened prior to executing the analyzer, and after processing the input C++ source. These errors may result:

   `FATAL ERROR **********Cannot open (or create, if output) File` *filename*

   Remedy: The file should be readable if for input, and writable, if for output. (Note that output files are created anew for each run.)

After initialization is complete, the analyzer begins to process GTREEs as specified by the specification. Here again, several *execution* errors may be generated.

9.2.2 Execution Time Errors

Should something catastrophic, like a core dump, occur, GEN++ will print out a message like:

`Fatal Error at Line` *nn* `while executing construct beginning` *constr* ...

to help the user localize the error. For further information, the user can use a symbolic debugger to inspect the execution stack, local variables, arguments etc of the various procedures in the users query.
The following is a list of other run time errors.

1. One category is node type errors.

   (a) *Arithmetic Expressions* expect integer operands. The following errors report violations:
   `"***ERROR !!!  Arguments to arith expr not integer !  at` *lineno*
   Remedy: check the specification to make sure that the argument is an expression that is guaranteed to be of type `C_Integer`.

   (b) Procedures expect the node they are passed (as an implicit argument) to match the `ROOTTYPE` declaration. If not, the following error is generated.
   `Procedure` *procname* `with node of type` *typename*`, not of type` *typename*
   With this error, the `CALL` to the procedure is aborted. Processing continues with the construct following the `CALL`. *Remedy:* check the specification to make sure that the current node when the procedure is being invoked is an argument of the

right type. If, for example the current node at the point is a `Statement`, and only `Switches` are acceptable, try bracketing the `CALL` inside a `(?Switch ...)` construct, or by using a `IF` construct with a `TYPEOF` condition.

(c) Foreign function calls (See § 6.4) expect arguments to be as declared in the `EXTERNAL PROC` declaration. If the arguments don't conform to the prescribe types, the errors are generated. If the argument is one of `STRING`, `FLOAT` or `INTEGER`, the following error will occur:

`Warning !!!! Type error on line` *lineno*

`In External Call Expression for arg` *argno* `is of type` *actual_type*`, not` *declared_type*

If the argument is declared to be type `ANY`, and it turns out to be a primitive datatype such as `STRING`, `FLOAT` or `INTEGER` instead, the following error will occur:

`Warning !!!! Type error on line` *lineno*

`In External Call Expression for arg` *argno* `is a primitive data value of type` *Prim_type*

2. Another category is errors relating to lists.

(a) The list iteration operator : { *typename* ...} expects the current node to be a list node (like the list of `Statements` in a `Block`). If the current node is not a list, the following error is generated:

`***WARNING** List iteration starting on linenumber` *lineno* `- Parent NOT a LIST !`

For example,

```
(?BinaryExpr
        <be_lhs
           {typename ...} >)
```

would result in the above error (left hand side of a binary expression is not a list). Likewise, if the a construct such as { *typename* ...} is used, and if any of the the members are not of type *typename*, this error is generated:

`**WARNING** Members of list in iteration on linenumber` *lineno* `not of type` *typename*

Remedy: check the the specification, and the node type declaration in the appendix to make sure that the current node for the { *typename* ...} is a list of the appropriate type.

(b) Lists in GEN++ are expected to be *uniform* ... a list of `Statements`, `Expressions`, etc. If the only lists used in the query are the ones occurring "naturally" in the GTREE, this error shouldn't arise. Non uniform lists (which can only be created by explicit `CONS` commands) generate the following error:

`*WARNING** Members of list in iteration on linenumber` *lineno* `is` *typename* `not type` *typename*

Remedy: check how the list is constructed, to make sure type incompatible elements are not introduced into the list.

3. Another category of errors occurs during internal GTREE operations. Sometimes, they can represent internal problems in GEN++, the generated code, or the GTREE libraries.

   (a) If an attempt to find a filler of a slot on a node that has no such slot, this error would occur:

   `***Error*** - Parent of type` *typename* `has no slot` *slotname*"

   This error could result from the use of a `SLOT` expression, or from the use of the < ... > construct (see the discussion on generation-time errors above) *Remedy:* Using the slot name and node type given, inspect the code to identify where the error is being generated; using the node type tables in the appendix, ensure that the node types and slots are being used consistently. If possible, embed the offending construct in a `(?`*nodetype* `...)`, or by using a `IF` construct with a `TYPEOF` condition to ensure correct nodetype is being used.

4. Another category is due to memory management failures. There are several possible errors:

   (a) GEN++ uses a stack of "dope vectors" to point into CFRONT data structures. These are called GNODEs. Normally, during the exploration of a tree, the maximum number of GNODEs is the depth of the tree. If, for some reason, more than a `#define`d (now 10 million) GNODEs are allocated, the following error results:

   `*****FATAL ERROR !!! USED UP MORE THAN` *MAXNUM* `gnodes !!`

   Remedy: Just a small matter of solving the halting problem. Most likely, there is an infinite loop somewhere. Check for infinite `WHILE` constructs and/or runaway recursive function calls.

   (b) GEN++ uses `malloc(3)` to allocate memory in sizes of `CHUNK` (`#define`d to 10,000) GNODEs. If for some reason, the malloc fails, This error will result:

   `Insufficient space - failure in mymalloc`

   Remedy: Try running the analyzer on a machine with greater memory capacity, or one that is less loaded.

10 Conclusion & Acknowledgements

GEN++ is a meta-tool that can be used to generate a variety of C++ analysis tools. This document has described GEN++ for friendly users.
Atts.
References
Appendixes A,B and C

References

[1] Devanbu, P., A Language and Front-end Independent Code Analyzer, *Proceedings, International Conference On Software Engineering*, Melbourne, Australia, May 1992.

[2] Murray, R.B., "Statically Typed Abstract Representation for C++ Programs", Second USENIX C++ Conference, 1992.

[3] Grass J., and Chen, Y-H., "The C++ Information Abstractor", Proceedings, *The Second USENIX C++ Conference*, San Francisco, USA, 1992.

[4] Chen, Y. F. and Ramamoorthy, C. V., The C Information Abstractor, *Proceedings of the Tenth International Computer Software and Applications Conference (COMPSAC)*, October 1986, Chicago, IL.

[5] Cleaveland, J. C., and Kintala, C., Tools for building Applications Generators, *AT&T Technical Journal*, July-Aug 1988. AT&T Bell Labs, North Andover, Mass.

[6] METATOOL Specification Driven Tool, System Overview, AT&T Bell Laboratories, June 1990. *Document Number: 193010-211* (Available from R. Tatem, AT&T Bell Labs, North Andover, Mass.)

[7] J.L. Steffen, Interactive examination of a C program with Cscope, Proc. USENIX Assoc. Winter Conference, Jan, 1985.

APPENDIX A

List of Nodetypes in C++

How to read the node type descriptions

In order to write a GEN++ query, it is necessary to be familiar with the "parse trees" of C++. The following list of descriptions defines the node type hierarchy and slots (or members) for each node type. For example, the following describes node type **BinaryExpr**, with its slots, base type and variants:

BinaryExpr	
base type is **Expression**	
variants	
Plus Minus Mult Div Mod Assignment NotEq EqEq LessEq Less GreaterEq Greater Lshift Rshift Comma BitOr BitAnd Xor LogOr LogAnd	
be_rhs	*a* **Expression**
be_lhs	*a* **Expression**

BinaryExpr has one base type, **Expression**, and 21 variants (all the different types of binary expressions). It has two slots, be_lhs, and be_rhs, both of which have as filler a node of type **Expression**. Occasionally, a slot name will appear with a preceding "!", as in the following description of node type `Switch`:

Switch	
base type is **Statement**	
!default_s	*a* **Statement**
!cases	*list of* **Statement**
switchbody	*a* **Statement**
switchcondition	*a* **Expression**

The "!" before a slot indicates that this slot will *not* be expanded while conducting a global search (using the "`[ ]`" construct) in a query. This prevents repeated traversals of shared subtrees, thus avoiding infinite recursions or unnecessary processing. In this example, the fillers of slot `cases` are also embedded in the `switchbody`. The `cases` slot is provided only for convenience and is not expanded automatically during a global search, since these statements would already be searched by a traversal of the body of the switch (filler of slot "switchbody"). "Expansion suppression" is used in a number of places to prevent runaway traversals. It can be circumvented by explicitly traversing a suppressed slot. For example, node type `Expression` contains a slot `!type` which is suppressed. To examine the type of an given expression, simply traverse to that slot explicitly:

```
PROC ScanTypes
ROOT Statement;
{
```

```
  [(?Expression
     <type ...>
  )]
}
```

As another example, a tool might need to examine the declaration of a type name that is referenced in a cast. To do this the following construct might be used:

```
[(?Ccast
   <refname
        <def ...>
   >
)]
```

Note that `refname` is of type `NameRef`, which contains a slot `!def` of type `Declaration`, which is suppressed.

A warning about locations is in order here. (See also pages 16, 17 and 20.) When traversing through suppressed slots, it is possible to encounter a slot which has a different location than the initial subtree being examined. This is the case in the `Ccast` example above. Before the traversal to slot `def`, the "current location" is the location of the cast or `NameRef`; after the traversal, `$location` will be at the declaration of the name. Thus if the original location is needed while processing `def`, it must be saved in a local variable before the traversal.

We are now ready to proceed to the specification. The next section contains a summary of the node type hierarchy. This is followed by an alphabetical list of nodetype specifications.

NOTE: *If you have received the normal shipment of* GEN++, *you will find therein two files,* `typedocs.html` *and* `typesummary.html` *with exactly the same information as given in the following pages; if you have access to Mosaic, (or a similar WWW browser) you will find that these files provide a far better way to look up this information.*

Node Type Hierarchy

```
# type.summary
Catch # Catch clause for try statement


T_Attr # Attributes of built-in types
  TA_Long
  TA_Unsigned
  TA_Signed
  TA_Numeric
  TA_Integral
  TA_Volatile
  TA_Const


FunctionTypeAttr # Attributes of function types
  FTA_Ellipsis
  FTA_Static
  FTA_Const


CV_Spec # Const/Volatile specifier
  Volatile
  Const


Initializer
  StaticInitializer # Initializing code for static initialization
  Expression
    PrimaryExpr
      Literal # Literal constant
        FloatConst
        CharConst
        IntConst
        StringConst
      NameRef # Reference to a user-defined name
        AccessedName # Name declared in an access declaration (see VisibilitySpec)
        LabelName # Name of a label
        TypedName # Name of object, function or type
          TypeName # User-defined type name
          FunctionName # Name of function
          IdName # Name of object
      PMFValue # Value of pointer to member function
```

```
      AggrValue # Aggregate initializer
    Delete # delete [size] ptr
    New # new (placements) type (inits)
    Mdot # Low level ref to member of virtual base or similar construct
    P2MemberRef # .* or ->*
      P2MData # p->*memdata
      P2MFunction # p->*memfunc
    MemberRef # .  or ->
      Arrow # x->y
      Dot # x.y
    Question # ?:
    Ccast # Simple C-style cast to basic type
      GeneratedCast # Compiler-generated cast
      UserCast # Explicit (user-specified) cast
    Sizeof_T # sizeof(type)
    Sizeof_E # sizeof(expr)
    Index # Subscripted expression x[y]
    FunCall # Generic function call
      GeneratedCall # Compiler-generated function call
        GConversion # Generated call of conversion operator
        GDestruction # Generated destructor call
        GConstruction # Generated constructor call
        GOperatorCall # Generated call of overloaded operator
      UserCall # User-specified function call
    UnaryExpr # Unary expression
      Deref # *x
      AddrOf #
&x
      Uplus # +x
      Uminus # -x
      UnaryCount # ++/-- operators
        PostInc # x++
        PostDec # x--
        PreDec # --x
        PreInc # ++x
      Not # !
Complement # ~
    BinaryExpr # Binary expression
      Plus # +
      Minus # -
      Mult # *
      Div # /
      Mod #
%
      Assignment
        Assign # =
        RshiftAssign # >>=
```

```
        LshiftAssign # <<=
        MinusAssign # -=
        PlusAssign # +=
        ModAssign #
%=
        MultAssign # *=
        DivAssign # /=
        OrAssign # |=
        AndAssign #
&=
        XorAssign #
^=
      NotEq # !=
      EqEq # ==
      LessEq # <=
      Less # <
      GreaterEq # >=
      Greater # >
      Lshift # <<
      Rshift # >>
      Comma # ,
      BitOr # |
      BitAnd #
&
      Xor #
^     LogOr # ||
      LogAnd #
&
&
    NullExpr # Null or missing expression


MemInit # Base/Member initializer in constructor definition


FunctionAttr # Attributes of function declarations
  FA_IncompleteInstantiation
  FA_Instantiation
  FA_Template
  FA_Ellipsis
  FA_Static
  FA_Const
  FA_Virtual
  FA_Inline
```

```
FunctionLinkage # Function linkage specifier
  Linkage_Cplusplus
  Linkage_C


BaseAllocation # Base class allocation attribute
  BA_Promoted
  BA_Allocated
  BA_Virtual
  BA_Plain


BaseAccess # Base access identification
  BA_Private
  BA_Protected
  BA_Public


MemberSpec # Member, friend or access declaration
  NestedDeclaration # Member or friend declaration
    FriendDeclaration # Friend declaration
      FriendFunction # Friend function declaration
      FriendClass # Friend class declaration
    MemberDeclaration # Member declaration
  VisibilitySpec # Base member access declaration
  AccessSpec # public:  / protected:  / private:


BaseSpec # Base class specification


ClassAttr # Attributes of class/union declarations
  CA_HasOpDelete
  CA_HasOpNew
  CA_HasVirtuals
  CA_HasOpAssign
  CA_HasNullCtor
  CA_HasCopyCtor
  CA_HasDtor
  CA_HasCtor
  CA_Specialized
  CA_Instantiated
  CA_Abstract


Actual # Template actual parameter (for instantiations)
```

```
  ExprActual # Actual value parameter
  TypeActual # Actual type parameter


Parameter # Template formal parameter
  ValueParameter # Template value parameter
  TypeParameter # Template type parameter


Statement # Generic statement
  Try # Try statement (not implemented in 3.0)
  Goto
  LabelStmt # Generic statement label
    Default # Switch default
    Case # Case label
    Label
  Continue
  Break
  Switch
  Do
  For
  While
  If
  Block # Explicit or compiler generated block
  Return
  NullStmt # Null or missing statement
  ExprStmt # expr;
  AsmStmt # Asm statement


NameValue # Integer value of name or full initializer if not constant


StorageClass
  SC_Static
  SC_Member # Non-static member
  SC_Auto
  SC_Register
  SC_Enum # Identifies an enumerator


Type # Type specifier
  BuiltinType # Built-in type
    T_Void
    T_LDouble
    T_Double
```

```
    T_Float
    T_Long
    T_Int
    T_Short
    T_Char
    T_Zero
    T_Any
  PtrType
  RefType
  ArrayType
  NamedType # Types with names
    Typeref # Reference to a typedef name
    AggrType # Class/Union type
      ClassType
      AnonType # Included for completeness
      UnionType
    EnumType
  FunctionType
  Overload # Identifies all instances of a function
  FieldType


Scope # Scope identification
  Scope_Extern
  Scope_Static # File or local static
  Scope_Label # Scope of labels -- function-wide
  Scope_Local # Scope of local declarations
  Scope_Argument # Scope of function argument
  Scope_Member # Scope of member
    Scope_Private
    Scope_Protected
    Scope_Public
  Scope_Weird # Unknown value (should not occur)


NameKind # Kind of name
  NK_Type
  NK_Enumerator
  NK_Label
  NK_This
  NK_Simple
  NK_Constructor
  NK_Destructor
  NK_Operator
```

```
Table # Access to internal cfront symbol table


Declaration # Generic declaration
  ObjectSpec # Object declaration or definition
    ObjectDecl # Object declaration (not allocated)
    ObjectDef # Object definition
    EnumeratorSpec # Declaration of enumerator
  FunctionSpec # Function declaration
    FunctionDef # Function definition
    FunctionDecl # Function declaration (no body)
  AsmSpec # Asm declaration
  EnumSpec # Enum declaration
  AggrSpec # Declaration of aggregate (class/struct/union)
    AggrDecl # Forward declaration of aggregate
      UnionDecl # Forward declaration of union
      ClassDecl # Forward declaration of class or struct
    AggrDef # Definition of aggregate (class/struct/union)
      AnonUnion # Anonymous union declaration
      UnionDef # Union definition
      ClassDef # Class definition
  Typedef # Typedef declaration
  Template # Declaration of a template
    ClassTemplate # Class template declaration
    FunctionTemplate # Function template declaration
    DataTemplate # Data template declaration
  LabelSpec # For consistency with other declarations
  BogusDecl # Implies internal error


FileSpec # File info


Location # Location in file


CFile # C++ source file


C_String


C_File


C_Float
```

`C_Integer`

Node Type Specifications

AccessSpec	public: / protected: / private:
base type is **MemberSpec**	

AccessedName	Name declared in an access declaration (see VisibilitySpec)
base type is **NameRef**	
verbose	*Print String*
name	*a* **NameRef**
!enclosing_class	*a* **NameRef**

Actual	Template actual parameter (for instantiations)
variants	
ExprActual TypeActual	

AddrOf	&x
base type is **UnaryExpr**	
ue_arg	*a* **Expression**

AggrDecl	Forward declaration of aggregate
base type is **AggrSpec**	
variants	
UnionDecl ClassDecl	

AggrDef	Definition of aggregate (class/struct/union)
base type is **AggrSpec**	
variants	
AnonUnion UnionDef ClassDef	
!friends	*iteration of* **Declaration**
members	*iteration of* **MemberSpec**
bases	*list of* **BaseSpec**
!size	*a* **C_Integer**
attrs	*list of properties of type* **ClassAttr**

AggrSpec	Declaration of aggregate (class/struct/union)
base type is **Declaration**	
variants	
AggrDecl AggrDef	
flavor	*a* **C_String**

AggrType	Class/Union type
base type is **NamedType**	
variants	
ClassType AnonType UnionType	

AggrValue	Aggregate initializer
base type is **PrimaryExpr**	
elements	*iteration of* **Initializer**

AndAssign	& =
base type is **Assignment**	

AnonType	Included for completeness
base type is **AggrType**	

AnonUnion	Anonymous union declaration
base type is **AggrDef**	

ArrayType	
base type is **Type**	
element_type	*a* **Type**
dimension	*a* **Expression**
!size	*a* **C_Integer**

Arrow	x−>y
base type is **MemberRef**	

AsmSpec	Asm declaration
base type is **Declaration**	
asm_string	*a* **C_String**

AsmStmt	Asm statement
base type is **Statement**	
asm_string	*a* **C_String**

Assign	=
base type is **Assignment**	

Assignment	
base type is **BinaryExpr**	
variants	
Assign RshiftAssign LshiftAssign MinusAssign PlusAssign ModAssign MultAssign DivAssign OrAssign AndAssign XorAssign	

BA_Allocated	
base type is **BaseAllocation**	

BA_Plain	
base type is **BaseAllocation**	

BA_Private	
base type is **BaseAccess**	

BA_Promoted	
base type is **BaseAllocation**	

BA_Protected	
base type is **BaseAccess**	

BA_Public	
base type is **BaseAccess**	

BA_Virtual	
base type is **BaseAllocation**	

BaseAccess	Base access identification
variants	
BA_Private BA_Protected BA_Public	

BaseAllocation	Base class allocation attribute
variants	
BA_Promoted BA_Allocated BA_Virtual BA_Plain	

BaseSpec	Base class specification
refname	*a* **NameRef**
allocation	*list of properties of type* **BaseAllocation**
access	*a* **BaseAccess**

BinaryExpr	Binary expression
base type is **Expression**	
variants	
Plus Minus Mult Div Mod Assignment NotEq EqEq LessEq Less GreaterEq Greater Lshift Rshift Comma BitOr BitAnd Xor LogOr LogAnd	
be_rhs	*a* **Expression**
be_lhs	*a* **Expression**

BitAnd	&
base type is **BinaryExpr**	

BitOr	
base type is **BinaryExpr**	

Block	Explicit or compiler generated block
	base type is **Statement**
!blockend	*a* **Location**
blockbody	*list of* **Statement**
!ltable	*a* **Table**
locals	*iteration of* **Declaration**

BogusDecl	Implies internal error
base type is **Declaration**	

Break	
base type is **Statement**	
!enclosing	*a* **Statement**

BuiltinType	Built–in type
base type is **Type**	
variants	
T_Void T_LDouble T_Double T_Float T_Long T_Int T_Short T_Char T_Zero T_Any	

CA_Abstract	
base type is **ClassAttr**	

CA_HasCopyCtor	
base type is **ClassAttr**	

CA_HasCtor	
base type is **ClassAttr**	

CA_HasDtor	
base type is **ClassAttr**	
!dtor	*a* **Declaration**

CA_HasNullCtor	
base type is **ClassAttr**	

CA_HasOpAssign	
base type is **ClassAttr**	

CA_HasOpDelete	
base type is **ClassAttr**	

CA_HasOpNew	
base type is **ClassAttr**	

CA_HasVirtuals	
base type is **ClassAttr**	

CA_Instantiated	
base type is **ClassAttr**	
actuals	*iteration of* **Actual**

CA_Specialized	
base type is **ClassAttr**	

CFile	C++ source file
!gtable	*a* **Table**
globals	*iteration of* **Declaration**
!files	*iteration of* **FileSpec**
!where	*a* **Location**
!filename	*a* **C_String**

CV_Spec	Const/Volatile specifier
variants	
Volatile Const	

C_File

C_Float

C_Integer

C_String

Case	Case label
base type is **LabelStmt**	
caselabel	*a* **Expression**
!casevalue	*a* **C_Integer**

Catch	Catch clause for try statement
catchbody	*a* **Block**
catcharg	*a* **Declaration**

Ccast	Simple C–style cast to basic type
base type is **Expression**	
variants	
GeneratedCast UserCast	
fromexpr	*a* **Expression**
!fromtype	*a* **Type**

CharConst	
base type is **Literal**	
chvalue	*a* **C_Integer**

ClassAttr	Attributes of class/union declarations
variants	
CA_HasOpDelete CA_HasOpNew CA_HasVirtuals CA_HasOpAssign CA_HasNullCtor CA_HasCopyCtor CA_HasDtor CA_HasCtor CA_Specialized CA_Instantiated CA_Abstract	

ClassDecl	Forward declaration of class or struct
base type is **AggrDecl**	

ClassDef	Class definition
base type is **AggrDef**	

ClassTemplate	Class template declaration
base type is **Template**	

ClassType	
base type is **AggrType**	

Comma	,
base type is **BinaryExpr**	

Complement	˜
base type is **UnaryExpr**	
ue_arg	*a* **Expression**

Const	
base type is **CV_Spec**	

Continue	
base type is **Statement**	
!enclosing	*a* **Statement**

DataTemplate	Data template declaration
base type is **Template**	

Declaration	Generic declaration
variants	
ObjectSpec FunctionSpec AsmSpec EnumSpec AggrSpec Typedef Template LabelSpec BogusDecl	
!defname	*a* **NameRef**
!offset	*a* **C_Integer**
storage	*a* **StorageClass**
scope	*a* **Scope**
kind	*a* **NameKind**
qualifier	*a* **NameRef**
type	*a* **Type**
!where	*a* **Location**
id	*a* **C_String**

Default	Switch default
base type is **LabelStmt**	

Delete	delete [size] ptr
base type is **Expression**	
del_size	*a* **Expression**
del_ptr	*a* **Expression**

Deref	*x
base type is **UnaryExpr**	
ue_arg	*a* **Expression**

Div	/
base type is **BinaryExpr**	

DivAssign	/=
base type is **Assignment**	

Do	
base type is **Statement**	
!doend	*a* **Location**
dobody	*a* **Statement**
docondition	*a* **Expression**

Dot	x.y
base type is **MemberRef**	

EnumSpec	Enum declaration
base type is **Declaration**	
enumerators	*iteration of* **EnumeratorSpec**

EnumType	
base type is **NamedType**	

EnumeratorSpec	Declaration of enumerator
base type is **ObjectSpec**	
init	*a* **Expression**
value	*a* **C_Integer**

EqEq	==
base type is **BinaryExpr**	

ExprActual	Actual value parameter
base type is **Actual**	
expr	*a* **Expression**

ExprStmt	expr;
base type is **Statement**	
expr	*a* **Expression**

Expression	
base type is **Initializer**	
variants	
PrimaryExpr Delete New Mdot P2MemberRef MemberRef Question Ccast Sizeof_T Sizeof_E Index FunCall UnaryExpr BinaryExpr NullExpr	
!type	*a* **Type**

FA_Const	
base type is **FunctionAttr**	

FA_Ellipsis	
base type is **FunctionAttr**	

FA_IncompleteInstantiation	
base type is **FunctionAttr**	
actuals	*iteration of* **Actual**

FA_Inline	
base type is **FunctionAttr**	

FA_Instantiation	
base type is **FunctionAttr**	
actuals	*iteration of* **Actual**

FA_Static	
base type is **FunctionAttr**	

FA_Template	
base type is **FunctionAttr**	

FA_Virtual	
base type is **FunctionAttr**	

FTA_Const	
base type is **FunctionTypeAttr**	

FTA_Ellipsis	
base type is **FunctionTypeAttr**	

FTA_Static	
base type is **FunctionTypeAttr**	

FieldType	
base type is **Type**	
f_expr	*a* **Expression**
!f_width	*a* **C_Integer**
f_type	*a* **Type**

FileSpec	File info
size	*a* **C_Integer**
name	*a* **C_String**
filid	*a* **C_Integer**

FloatConst	
base type is **Literal**	

For	
base type is **Statement**	
forbody	*a* **Statement**
forincrement	*a* **Expression**
forcondition	*a* **Expression**
forinit	*a* **Statement**

FriendClass	Friend class declaration
base type is **FriendDeclaration**	

FriendDeclaration	Friend declaration
base type is **NestedDeclaration**	
variants	
FriendFunction FriendClass	

FriendFunction	Friend function declaration
base type is **FriendDeclaration**	

FunCall	Generic function call
base type is **Expression**	
variants	
GeneratedCall UserCall	
callargs	*iteration of* **Expression**
callexpr	*a* **Expression**
!callname	*a* **NameRef**

FunctionAttr	Attributes of function declarations
variants	
FA_IncompleteInstantiation **FA_Instantiation** **FA_Template FA_Ellipsis** **FA_Static FA_Const** **FA_Virtual FA_Inline**	

FunctionDecl	Function declaration (no body)
base type is **FunctionSpec**	

FunctionDef	Function definition
base type is **FunctionSpec**	
body	*a* **Block**
meminit	*list of* **MemInit**

FunctionLinkage	Function linkage specifier
variants	
Linkage_Cplusplus **Linkage_C**	

FunctionName	Name of function
base type is **TypedName**	
verbose	*Print String*
!storage	*a* **StorageClass**

FunctionSpec	Function declaration
base type is **Declaration**	
variants	
FunctionDef FunctionDecl	

header	*Print String*
signature	*Print String*
sigstring	*a* **C_String**
attrs	*list of properties of type* **FunctionAttr**
linkage	*a* **FunctionLinkage**
args	*iteration of* **Declaration**
returntype	*a* **Type**
!type	*a* **Type**

FunctionTemplate	Function template declaration
base type is **Template**	

FunctionType	
base type is **Type**	
argtypes	*iteration of* **Type**
returntype	*a* **Type**
member_of	*a* **NameRef**
attrs	*list of properties of type* **FunctionTypeAttr**

FunctionTypeAttr	Attributes of function types
variants	
FTA_Ellipsis FTA_Static FTA_Const	

GConstruction	Generated constructor call
base type is **GeneratedCall**	

GConversion	Generated call of conversion operator
base type is **GeneratedCall**	

GDestruction	Generated destructor call
base type is **GeneratedCall**	

GOperatorCall	Generated call of overloaded operator
base type is **GeneratedCall**	

GeneratedCall	Compiler–generated function call
base type is **FunCall**	
variants	
GConversion GDestruction GConstruction GOperatorCall	

GeneratedCast	Compiler–generated cast
base type is **Ccast**	
!totype	*a* **Type**

Goto	
base type is **Statement**	
gotoname	*a* **NameRef**

Greater	>
base type is **BinaryExpr**	

GreaterEq	>=
base type is **BinaryExpr**	

IdName	Name of object
base type is **TypedName**	
verbose	*Print String*
!value	*a* **NameValue**
!storage	*a* **StorageClass**

If	
base type is **Statement**	
ifFbranch	*a* **Statement**
ifTbranch	*a* **Statement**
ifcondition	*a* **Expression**

Index	Subscripted expression x[y]
base type is **Expression**	
ix_subscript	*a* **Expression**
ix_left	*a* **Expression**

Initializer	
variants	
StaticInitializer	**Expression**

IntConst	
base type is **Literal**	
civalue	*a* **C_Integer**

Label	
base type is **LabelStmt**	
labelname	*a* **NameRef**

LabelName	Name of a label
base type is **NameRef**	
!stmt	*a* **Statement**

LabelSpec	For consistency with other declarations
base type is **Declaration**	
!stmt	*a* **Statement**

LabelStmt	Generic statement label
base type is **Statement**	
variants	
Default Case Label	

Less	<
base type is **BinaryExpr**	

LessEq	<=
base type is **BinaryExpr**	

Linkage_C	
base type is **FunctionLinkage**	

Linkage_Cplusplus	
base type is **FunctionLinkage**	

Literal	Literal constant
base type is **PrimaryExpr**	
variants	
FloatConst CharConst IntConst StringConst	
strep	*a* **C_String**

Location	Location in file
line	*a* **C_Integer**
file	*a* **C_String**
filid	*a* **C_Integer**

LogAnd	&&
base type is **BinaryExpr**	

LogOr	||
base type is **BinaryExpr**	

Lshift	<<
base type is **BinaryExpr**	

LshiftAssign	<<=
base type is **Assignment**	

Mdot	Low level ref to member of virtual base or similar construct
base type is **Expression**	
mem	*a* **Expression**
bname	*a* **C_String**

MemInit	Base/Member initializer in constructor definition
init	*iteration of* **Expression**
!bmdef	*a* **Declaration**

MemberDeclaration	Member declaration
base type is **NestedDeclaration**	

MemberRef	. or ->
base type is **Expression**	
variants	
Arrow Dot	
qualifier	*a* **NameRef**
member	*a* **NameRef**
left	*a* **Expression**
!aggrtype	*a* **AggrType**

MemberSpec	Member, friend or access declaration
variants	
NestedDeclaration **VisibilitySpec AccessSpec**	
!where	*a* **Location**
scope	*a* **Scope**

Minus	-
base type is **BinaryExpr**	

MinusAssign	-=
base type is **Assignment**	

Mod	%
base type is **BinaryExpr**	

ModAssign	%=
base type is **Assignment**	

Mult	*
base type is **BinaryExpr**	

MultAssign	*=
base type is **Assignment**	

NK_Constructor
base type is **NameKind**

NK_Destructor
base type is **NameKind**

NK_Enumerator
base type is **NameKind**

NK_Label
base type is **NameKind**

NK_Operator
base type is **NameKind**

NK_Simple
base type is **NameKind**

NK_This	
base type is **NameKind**	
!thisclass	*a* **AggrDef**

NK_Type
base type is **NameKind**

NameKind	Kind of name
variants	
NK_Type NK_Enumerator NK_Label NK_This NK_Simple NK_Constructor NK_Destructor NK_Operator	

NameRef	Reference to a user–defined name
base type is **PrimaryExpr**	
variants	
AccessedName LabelName TypedName	
verbose	*Print String*
mangled	*Print String*
!def	*a* **Declaration**
!scope	*a* **Scope**
!kind	*a* **NameKind**
!id	*a* **C_String**

NameValue	Integer value of name or full initializer if not constant
fullvalue	*a* **Expression**
intvalue	*a* **C_Integer**

NamedType	Types with names
base type is **Type**	
variants	
Typeref AggrType EnumType	
refname	*a* **NameRef**

NestedDeclaration	Member or friend declaration
base type is **MemberSpec**	
variants	
FriendDeclaration MemberDeclaration	
def	*a* **Declaration**

New	new (placements) type (inits)
base type is **Expression**	
new_function	*a* **NameRef**
new_placement	*iteration of* **Expression**
new_inits	*iteration of* **Expression**
new_type	*a* **Type**

Not	!
base type is **UnaryExpr**	
ue_arg	*a* **Expression**

NotEq	!=
base type is **BinaryExpr**	

NullExpr	Null or missing expression
base type is **Expression**	

NullStmt	Null or missing statement
base type is **Statement**	

ObjectDecl	Object declaration (not allocated)
base type is **ObjectSpec**	

ObjectDef	Object definition
base type is **ObjectSpec**	
init	*a* **Initializer**

ObjectSpec	Object declaration or definition
base type is **Declaration**	
variants	
ObjectDecl ObjectDef EnumeratorSpec	

OrAssign	|=
base type is **Assignment**	

Overload	Identifies all instances of a function
base type is **Type**	
!instances	*iteration of* **Type**

P2MData	p−>*memdata
base type is **P2MemberRef**	

P2MFunction	p−>*memfunc
base type is **P2MemberRef**	

P2MemberRef	.* or −>*
base type is **Expression**	
variants	
P2MData P2MFunction	
member	*a* **Expression**
left	*a* **Expression**
!aggrtype	*a* **AggrType**

PMFValue	Value of pointer to member function
base type is **PrimaryExpr**	
f	*a* **Expression**
i	*a* **Expression**
d	*a* **Expression**

Parameter	Template formal parameter
variants	
ValueParameter TypeParameter	
type	*a* **Type**
id	*a* **C_String**

Plus	+
base type is **BinaryExpr**	

PlusAssign	+=
base type is **Assignment**	

PostDec	x−−
base type is **UnaryCount**	
ue_arg	*a* **Expression**

PostInc	x++
base type is **UnaryCount**	
ue_arg	*a* **Expression**

PreDec	−−x
base type is **UnaryCount**	
ue_arg	*a* **Expression**

PreInc	++x
base type is **UnaryCount**	
ue_arg	*a* **Expression**

PrimaryExpr	
base type is **Expression**	
variants	
Literal NameRef PMFValue AggrValue	

PtrType	
base type is **Type**	
points_to	*a* **Type**
member_of	*a* **NameRef**
attrs	*list of properties of type* **CV_Spec**

Question	?:
base type is **Expression**	
q_false	*a* **Expression**
q_true	*a* **Expression**
q_test	*a* **Expression**

RefType	
base type is **Type**	
refers_to	*a* **Type**
member_of	*a* **NameRef**
attrs	*list of properties of type* **CV_Spec**

Return	
base type is **Statement**	
value	*a* **Expression**

Rshift	>>
base type is **BinaryExpr**	

RshiftAssign	>>=
base type is **Assignment**	

SC_Auto	
base type is **StorageClass**	

SC_Enum	Identifies an enumerator
base type is **StorageClass**	

SC_Member	Non–static member
base type is **StorageClass**	

SC_Register	
base type is **StorageClass**	

SC_Static	
base type is **StorageClass**	

Scope	Scope identification
variants	
Scope_Extern Scope_Static Scope_Label Scope_Local Scope_Argument Scope_Member Scope_Weird	

Scope_Argument	Scope of function argument
base type is **Scope**	

Scope_Extern	
base type is **Scope**	

Scope_Label	Scope of labels –– function–wide
base type is **Scope**	

Scope_Local	Scope of local declarations
base type is **Scope**	

Scope_Member	Scope of member
base type is **Scope**	
variants	
Scope_Private Scope_Protected Scope_Public	

Scope_Private	
base type is **Scope_Member**	

Scope_Protected	
base type is **Scope_Member**	

Scope_Public	
base type is **Scope_Member**	

Scope_Static	File or local static
base type is **Scope**	

Scope_Weird	Unknown value (should not occur)
base type is **Scope**	

Sizeof_E	sizeof(expr)
base type is **Expression**	
szexpr	*a* **Expression**

Sizeof_T	sizeof(type)
base type is **Expression**	
sztype	*a* **Type**

Statement	Generic statement
variants	
Try Goto LabelStmt Continue Break Switch Do For While If Block Return NullStmt ExprStmt AsmStmt	
!successor	*a* **Statement**
!where	*a* **Location**

StaticInitializer	Initializing code for static initialization
base type is **Initializer**	
expr	*a* **Expression**

StorageClass	
variants	
SC_Static SC_Member SC_Auto SC_Register SC_Enum	

StringConst	
base type is **Literal**	

Switch	
base type is **Statement**	
!default_s	*a* **Statement**
!cases	*list of* **Statement**
switchbody	*a* **Statement**
switchcondition	*a* **Expression**

TA_Const	
base type is **T_Attr**	

TA_Integral	
base type is **T_Attr**	

TA_Long	
base type is **T_Attr**	

TA_Numeric	
base type is **T_Attr**	

TA_Signed	
base type is **T_Attr**	

TA_Unsigned	
base type is **T_Attr**	

TA_Volatile	
base type is **T_Attr**	

T_Any	
base type is **BuiltinType**	

T_Attr	Attributes of built–in types
variants	
TA_Long TA_Unsigned TA_Signed TA_Numeric TA_Integral TA_Volatile TA_Const	

T_Char	
base type is **BuiltinType**	

T_Double	
base type is **BuiltinType**	

T_Float	
base type is **BuiltinType**	

T_Int	
base type is **BuiltinType**	

T_LDouble	
base type is **BuiltinType**	

T_Long	
base type is **BuiltinType**	

T_Short	
base type is **BuiltinType**	

T_Void	
base type is **BuiltinType**	

T_Zero	
base type is **BuiltinType**	

Table	Access to internal cfront symbol table
members	*iteration of* **NameRef**
scope	*a* **C_String**

Template	Declaration of a template
base type is **Declaration**	
variants	
ClassTemplate FunctionTemplate DataTemplate	
formals	*iteration of* **Parameter**

Try	Try statement (not implemented in 3.0)
base type is **Statement**	
tryhandlers	*list of* **Catch**
tryblock	*a* **Block**

Type	Type specifier
variants	
BuiltinType PtrType RefType ArrayType NamedType FunctionType Overload FieldType	
t_attrs	*list of properties of type* **T_Attr**

TypeActual	Actual type parameter
base type is **Actual**	
type	*a* **Type**

TypeName	User–defined type name
base type is **TypedName**	
verbose	*Print String*

TypeParameter	Template type parameter
base type is **Parameter**	

TypedName	Name of object, function or type
base type is **NameRef**	
variants	
TypeName FunctionName IdName	
!qualifier	*a* **NameRef**
!type	*a* **Type**

Typedef	Typedef declaration
base type is **Declaration**	

Typeref	Reference to a typedef name
base type is **NamedType**	
!type	*a* **Type**

Uminus	–x
base type is **UnaryExpr**	
ue_arg	*a* **Expression**

UnaryCount	++/–– operators
base type is **UnaryExpr**	
variants	
PostInc PostDec PreDec PreInc	
ue_arg	*a* **Expression**

UnaryExpr	Unary expression
base type is **Expression**	
variants	
Deref AddrOf Uplus Uminus UnaryCount Complement Not	

UnionDecl	Forward declaration of union
base type is **AggrDecl**	

UnionDef	Union definition
base type is **AggrDef**	

UnionType	
base type is **AggrType**	

Uplus	+x
base type is **UnaryExpr**	
ue_arg	*a* **Expression**

UserCall	User–specified function call
base type is **FunCall**	

UserCast	Explicit (user–specified) cast
base type is **Ccast**	
totype	*a* **Type**

ValueParameter	Template value parameter
base type is **Parameter**	

VisibilitySpec	Base member access declaration
base type is **MemberSpec**	
verbose	*Print String*
name	*a* **NameRef**
!enclosing_class	*a* **NameRef**

Volatile	
base type is **CV_Spec**	

While	
base type is **Statement**	
whilebody	*a* **Statement**
whilecondition	*a* **Expression**

Xor	ˆ
base type is **BinaryExpr**	

XorAssign	ˆ=
base type is **Assignment**	

Index