BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-89-M9

"A Query Processor for an Object Oriented Database"

by
Wayne Dexter Wong

A Query Processor For An Object Oriented Database

by

Wayne Dexter Wong

Research Project

Submitted in partial fulfillment of the requirements for the

Degree of Master of Science in the Department of Computer Science

at Brown University.

October 1989

This project by Wayne Dexter Wong
is accepted in its present form by the Department of
Computer Science in partial fulfillment of the
requirements for the degree of Master of Science.

Date 10/30/89

Stanley B. Zdonik

Stanley B. Zdonik, Advisor

A Query Processor For An Object-Oriented Database

Wayne Dexter Wong

October 10, 1989

1 Introduction

Since the late 1960's, computer science has seen a steady growth of interest in databases and database management systems. It is only within the last decade, however, that much of this interest has focused on a particular new perspective on the topic – namely, the object-oriented model. This model, with its notion of grouping associated pieces of information into "objects" and allowing access to said information only via operations dedicated to the containing objects, is considered a remarkable tool for representing applications at a more "conceptual" level as well as promoting software reusability and maintainability.

However, attempts to provide a suitable framework for *queries* on the object-oriented model have met with mixed notices. It appears that the amorphous nature of a generalized entity, or "object", containing both properties and algorithms – which are themselves objects – does not lend itself easily to a straightforward and yet comprehensive syntax for accessing and manipulating such objects. This is often the main criticism of proponents of the relational model, whose tabular formats are easily translated into familiar-looking 2-dimensional array accesses.

This document describes the design and implementation of a "processor" (indeed, a "compiler" of sorts) for queries on an object-oriented database. Specifically, the query format is that proposed in [Sha89] and the database in question is based on the ENCORE data model described in [Zdo86]. The query processor translates

queries expressed in a high-level algebra into invocations of lower-level database methods on objects within the type system (described in a subsequent section). The query processor itself is part of a special preprocessor allowing queries to be embedded within programs written in a database programming language.

Section 2 gives a brief outline of similar efforts in this particular area, followed in section 3 by an informal description of the ENCORE data model. (The author assumes, however, that the reader is already familiar with standard object-oriented concepts.) Section 4 then describes the query algebra forming the input for the query processor (henceforth QP), after which section 5 discusses the phases of the processor in a manner similar to the principle stages of a conventional programming-language compiler. Section 6 demonstrates these stages via analysis of a small example; and Section 7 places the QP within the context of the ENCORE front-end as a whole. Finally, Section 8 mentions some as yet unresolved implementation issues.

2 Related Endeavors

In his paper on the INGRES relational database system, [Sto76] presents the first major instance of what has, for better or worse, become the most widely accepted methodology for translation of database queries. In this technique, a special preprocessor is used to accept a database programming language with embedded queries, which are denoted by a special enclosing syntax. Queries thus recognized are passed to a query parser (usually based on *yacc*) which, after syntax and type checking, generates parse trees containing enough information about their origin for a query optimizer to restructure said trees according to a given set of optimization criteria. Finally, compilable code is generated which makes calls on existing database library routines to perform the query.

An interesting variation on this approach was given by [Ban87], in which a query interpreter allows the user to interactively construct, run, and peruse the intermediate results of complex queries. This is accomplished by allowing interactive access to the unique identifier associated with a given object.

The "code generation" theme was taken a major step farther in [Gra87]. Although the initial query parser ([Car86]) is much in the standard vein of such translators, the query optimizer is one that has been *automatically generated* from a predefined input. This input consists of a description of the query operators and their corresponding database functions (with a cost formula for each) and a set of optimization rules. The same authors have also developed a QUEL Interface (using the Cornell Program Synthesizer Generator [Rep84]) which actually guides the user in building queries.

3 The ENCORE Data Model

This section, based on material in [Zdo86], is not a full description of the ENCORE model but rather one which provides the basic understanding necessary to comprehend subsequent sections.

The ENCORE model is commonly (and appropriately) referred to as a *type system* – that is, all objects are typed and all types are objects. In fact, the supertype of all objects is type *Object*. Objects may be comprised of several *attributes* which are themselves objects. These attributes may be *properties*, analogous to tuple-attributes in the relational model, or *operations*, which are, in effect, *types* which have "invocation images" as instances. These operations define the object-oriented *methods* which are an application's only means of access to the information within the object. Objects also may be grouped into *sets*, which may in turn be grouped with other sets into sets. Syntactically speaking, if an object is of type T (which in turn is of type *Type*), all objects of this type may be included in a *collection* of type $Set[T]$. (Note that there may be many instances of this particular set type.)

In addition, a group of objects of different types may be combined into a single object of type *Tuple*. This type – a nod in the direction of the relational model – is represented as $Tuple[< (A_1, T_1), ..., (A_n, T_n) >]$, where A_m is the name of the tuple *attribute* that contains the object of type T_m.

Regarding the actual representation of objects, each object is known by a unique identifier, henceforth called a *UID*. If two objects have the same UID, they are the same object. This particular form

of object equality, which is the form used in the QP, is known as *0-equality*. A complete treatment of the various means by which two objects can be construed as "equal" is given in [Kho86] and [Sha89].

4 The Query Algebra

The following subsections, describing the basic queries that can be input to the processor, are extracted largely from [Sha89].

4.1 *Select*

The *Select* operation creates a collection of database objects which satisfy a selection *predicate*:

$$Select(S, p) = \{s | (s\ in\ S) \wedge p(s)\}$$

where S is a collection and p is a predicate defined over the type of the members of S. For example:

$$Select(Cats, \$c\ c@name == "Phydeaux");$$

returns a collection of objects of type Cat whose *name* attribute contains the string-object corresponding to the string "Phydeaux". the $\$c$ notation indicates that c is a *lambda variable* which ranges over all values in *Cats*.

4.2 *Image*

The *Image* operation returns components of objects in a collection:

$$Image(S, f : T) = \{f(s) | s\ in\ S\}$$

where S is collection and f, when invoked upon an object in S, returns an object of type T. For example:

$$Image(Employees, \$e\ e@age);$$

returns a set of objects containing the ages of the members of the collection *Employees*.

4.3 *Project*

The *Project* operation is an extension of *Image* which returns a collection of *tuples*, one for each member of the collection queried upon. Each tuple contains a list of components of a member:

$$Project(S, \{(A_1, f_1), ..., (A_n, f_n)\}) = \{< A_1 : f_1(s), ..., A_n : f_n(s) > | s \; in \; S\}$$

where S is a collection and A_i is the attribute name for the tuple field containing $f_i(s)$, where s is a member of S. For example:

$$Project(Cars, \$c \; \{(Who, c@Owner), (What, c@Make), (HowMuch, c@Price)\})$$

returns a collection of tuples (triples, in this case) giving the owner, make, and price of each car in the *Cars* collection.

4.4 *Ojoin*

The *Ojoin* operation, like *Project*, returns a collection of tuples. However, *Ojoin* operates over *two* collections, and each of the tuples in the resultant collection has two attributes containing a pair of objects – one from each of the collections being queried on – which are related via a specified predicate:

$$Ojoin(S, R, A_1, A_2, p) = \{< A_1 : s, A_2 : r > | s \; in \; S \wedge r \; in \; R \wedge p(s, r) = TRUE\}$$

where S and R are collections, A_1 and A_2 are the names to be given to the two attributes of the result, and p is a predicate defined over objects from S and R. For example:

$$Ojoin(Soldiers, Civilians, S, C, \$s \; \$c \; s@age == c@age)$$

returns a list of pairs consisting of a soldier and a civilian who are of the same age.

4.5 others

The remaining queries in [Sha89] are discussed mainly for completeness here, since the QP need only translate them directly into their corresponding ENCORE query methods.

Flatten(S) takes S, which is a set of sets, and returns a set of non-set objects – that is, it eliminates the nesting of sets by bringing all member objects to the same level.

Nest(C, A_i) takes C, a collection of tuples, and A_i, an attribute of the tuple type, and collapses into a single tuple all those tuples which have matching values in all their attributes except (possibly) A_i. In this case, A_i becomes a *set* of all the values it held when the other attribute values were "held constant".

UnNest(C, A_i) takes C and transforms any tuple with an attribute containing a set of objects into a collection of tuples, one for each member of the set contained in the original attribute. In effect, this is a *Flatten* operation for tuples.

DupEliminate(S, i) takes collection S and invokes the test for *i-equality* (as discussed in [Kho86] and [Sha89]) on its members. It then replaces all members which are i-equal with a single such member.

Coalesce(S, A_k, i) takes S, which is a collection of tuples, invokes the i-equality test on the values of attribute A_k in each tuple. Then all i-equal values of A_k are replaced by one of the values.

5 Design Overview

As was alluded to in the Introduction, the QP resembles, in function and form, a compiler for a small programming language; its design and surroundings are similar to those of the INGRES QP [Sto76]. Statements in the source language (query algebra) are in turn embedded (as in EQUEL [All76]) in programs written in a "database programming language"(DPL), which is C extended with object-oriented types and functions via additional syntactic analysis. The QP is a part of the larger ENCORE DPL preprocessor and is called whenever a query is encountered in the source code.

The QP translates algebraic statements into method invocations on *collection* objects; one such method exists for each query operator in the algebra. Part of this process involves creating ENCORE operation objects which capture the algebraic expressions appearing in the query.

The query translation process follows a fairly conventional path.

Query statements are tokenized and then parsed, via a set of predefined linguistic rules. As the parsing proceeds, type checking (and the "derivation" of types for certain objects) is performed and, gradually, operations in the ENCORE DPL are generated which call other pre-existing ENCORE methods. Finally, a code fragment which invokes these operations is generated, and replaces the original query embedded in the source code. The DPL preprocessor then sends this newly generated code, along with the DPL code in which the original query was embedded, to the standard C preprocessor and compiler. When the code is executed, the aforementioned ENCORE methods are executed over the ENCORE type system, which retrieves desired objects from the object-oriented database server OBSERVER [Ska86].

5.1 Syntactic Analysis

The QP receives, as input, strings (i.e. *char **) containing complete queries. It then tokenizes these strings (using a lexical analyzer produced by the standard generation tool *lex*) and parses them (using a parser generated by the equally standard generation tool *yacc*). It is in the semantic actions associated with each grammar rule that the computing and checking of types, and the generation of actual C code to perform queries, occurs. The complete grammar and actions are included at the end of this document, but a few major grammar rules bear explaining:

- *query* is a query statement (i.e. a string commencing with SELECT, IMAGE, PROJECT, OJOIN, FLATTEN, NEST, UNNEST, COALESCE, or DUPELIM), including its arguments (some of which may themselves be queries).
- *variable* is a lambda variable followed by 0 or more properties, each of which is to be retrieved from the object resulting from the previous properties being retrieved from the lambda variable. Properties are separated from the lambda variable and from each other by "@", as in "j@address@city". This refers to the *city* property of the *address* property of *j*. A number of variables (and possibly queries) combined via arithmetic operator-methods forms an *expression.*

- *pred* is a *predicate*, comprised of a boolean-valued expression or combination thereof. The standard logical operators (*and*, *or*, *not*, etc.) are permitted, plus the operators *MemberOf* (testing for membership of an object in a set) and *SubsetOf* (testing for containment of a set within another set).

5.2 Type "Derivation" and Type Checking

In order to type check a query and generate properly typed operations, the QP must deduce and keep track of the types of any queries, collections, variables, or lambda variables. In order to accomplish this, the types of certain query arguments must be *derived*, since these arguments may not be declared symbols but variables composed of expressions involving symbols. For example, deriving the type of *j@NumCats + j@NumDogs* involves deriving the types of the *NumDogs* and *NumCats* properties of *j* and then the type of the result when the two values are added. The matter is further complicated since some arguments may themselves be queries. Eventually, it eventually becomes necessary to access the symbol table created by the preprocessor in order to terminate this "recursion". A discussion of the motivations behind determining the types of variables is given in [Nix87].

Only in the case of collections directly specified by name, as in "Dogs", is the preprocessor symbol table accessed. Once the entry for the collection has been located and the string variable describing its type has been retrieved from the entry, the operation *GetTypeObject* (called with the type-string) returns the corresponding type object. Retrieving the *memType* property of this type object yields the type of any lambda variable associated with that collection – that is, it represents the type of members of that collection.

For queries, types are derived by first retrieving the *OperationType* object (for the query) associated with the collection type on which the query is to operate. We then invoke the *TypeCheck* method on that OperationType object along with a list of the types of the arguments to the query method.

For variables, type derivation means (recursively speaking) starting by deriving the type of a variable, then deriving the type of a

property of the variable. The "basis" in this case is represented by a lambda variable, whose type is derived by first retrieving the type-string of the variable from the lambda variable stack (to be discussed later), then passing it as an argument to the ENCORE function *GetTypeObject*. This function returns the type object associated with the type-string.

The type of a property of a variable is then derived by first retrieving the PropertyType object associated with the property in question. Then the *valueClass* property of the PropertyType object is retrieved, which gives us the result type of the property of the variable. If there are further properties used in the variable, then this result type assumes the role of the variable, and so on. Note how the *yacc* input structure lends itself to this sort of approach.

Regarding the actual *checking* of types, the primary responsibility for this lies with the preprocessor containing calls to the QP. However, there are instances where it is advantageous for type checking to be performed at the query-translation level. This also is treated in [Nix87].

As defined by [Sha89], all queries operate on *sets* (which may themselves contain sets). In the ENCORE model, these groupings are known as *collections*. Hence having first parsed a query into its highest-level components (i.e. its name and its arguments), the QP checks the type of the query's first argument, which must be of a collection type. In addition, variables (as in *j@address@city*) may be checked upon retrieval of each successive property – if a retrieval produces a null object, then either the property or its "target" object is of an inappropriate type.

In addition, before generating a call to an arithmetic or logical operator-method, the QP may examine the type of the would-be parameters to said operator. For instance, the second argument to the *MemberOf* operator must be of a collection type, as must both the arguments to the *SubsetOf* operator.

The preceding discussion has concerned itself with type checking at *compile-time*. This, of course, assumes that the type objects from which objects are generated already exist in the type system. Although this is predominantly the case, there may be occasions where *new* types, and objects of these types, are created at *run-*

time and queries executed which involve them. In these instances, a certain amount of type-checking may be executed at run-time.

Determining if a variable's type exists at compile-time is relatively straightforward. If a type does not currently exist within the type system, a call to *GetTypeObject* will return a type object reserved especially for as yet undefined types. This sets a flag instructing the QP to allow type-checking to be performed at run-time.

As regards type derivation at run-time, the QP's current recourse is to assign all expressions to be of type *ENObject*, ENCORE's "ultimate default". Another possible approach would be to actually derive types *at* run-time (by which time all necessary types would exist), hence dividing the code-generation duties between run-time and compile-time. This method, however, appeared to offer few benefits in return for the required effort and was rejected.

5.3 Code Generation

As mentioned previously, code generation is performed concurrently with query parsing. Hence code is generated in a "bottom-up" fashion, with source for the lowest-level constructs (in this case, variables) being generated first, then used in the generation of code for higher-level forms (such as predicates), and so forth all the way up to the final "main program", a call to which replaces the original query statement embedded in the host code.

This approach ,discussed in [Ant77] as "the translation method", differs from other efforts ([Sto76], [Gra87]) in that the output of the query parser is in the form of actual program text rather than trees of database operators. The reasons for choosing this methodology are twofold. First, the nature of the algebra as described in [Sha89] does not lend itself to deep nesting of queries from a user standpoint. Since the usual purpose of generating "execution trees" from queries is optimization, which is effective and worthwhile primarily in cases of nested queries, it was decided that generating trees from which code is then generated would not recoup its investment. The second reason is somewhat more pragmatic: the focus of the author's work is query processing rather than optimization, and time constraints did not allow for the implementation of what would essentially be another, separate "compilation". However, this is not to imply that

query optimization is a closed matter as far as implementation goes.

For expressions, once the component variables have been transformed into C variables (which is usually a direct translation), their relationship to one another is represented in code by generating invocations of ENCORE's binary-operator methods. Thus $a+b$ becomes $Add(a, b)$. Naturally, a and b may themselves be comprised of such method calls.

Code for predicates, which are comprised of expressions and/or other predicates, is also generated as method invocations. Hence $a < b$ becomes $LessThan(a, b)$. This code in turn becomes the return-value of a boolean operation which is generated.

This brings us to the issue of *parameters* for generated operations. More specifically, how are operation arguments and local variables generated, or even determined? To address this, the QP uses two stacks; one for lambda variables and one for collections. (Two stacks are maintained because references to lambda variables and to collections are needed at different points in the code generation process.) For the lambda variable stack, entries are grouped into *frames* corresponding to the level of scoping at which entries appear in the query. Variables occurring at the current level are generated as arguments to the operation being generated; those from lower levels become local variables to the operation, and are initialized by being extracted from an "arglist" passed in by the operation. (This list also contains all collections used in the query.) Hence when the code at a given scoping level has been generated, the stack entries for the corresponding frame are popped. The QP detects a new scoping level whenever a new set of lambda variables is specified, at which point a special "first-in-frame" marker is passed along with the first variable of the set to be pushed. The collection stack, however, does not use frames since collections are actually defined completely outside the query (in the surrounding C code) and are referenced throughout same.

Given this, we can generate code for complete queries. Each query is "compiled" into a typed operation which invokes the ENCORE method for the query on the "target collection" and returns the result of the query. Methods for queries involving predicates (such as *Select* and *Ojoin*) have, as arguments, the collection(s)

upon which the query is to operate, the boolean operation object that checks the predicate, and a list *args*. This list contains all the arguments to the predicate-operation except the first, which is assumed by the operation to be a member of the collection argument. In ENCORE's parlance, such an initial argument is known as *self* – a method is said to operate on *self*. The actual ENCORE method for *Select* invokes the predicate operation on successive elements (*selves*) of the target collection and inserts those producing TRUE into the resultant collection. The *Ojoin* method is similar, except that it submits every possible pairing of an element from the first collection and one from the second to the predicate, and combines "successful combinations" into tuples which are inserted into the return value.

Translation of *Image* queries is also similar to that of *Selects*. However, in this case instead of a boolean operation being generated, a typed operation is constructed which computes the operation designated by the original query to act upon successive members of the target collection. As expected, each of these members takes a turn at being *self* for an invocation of the operation, and each return-value of the operation is inserted into the resultant collection.

Processing of *Project* queries requires a somewhat different approach. Typed operations are generated for each of the operations appearing in the tuple argument of the query. These operations are then assigned, as an identifier, the name of the attribute which they will compute, and inserted into a list which is passed to the *Project* method. The remaining method arguments are, as previously, the target collection and *args*. The generated operations will extract any necessary arguments from *args*.

6 Anatomy Of A Query (an example)

To better illustrate the query translation process, we present a step-by-step treatment of one of the previous examples (assuming, for the sake of simplicity, that the types involved are predefined):

$$Select(Cats, \$c\ c@name == "Phydeaux");$$

The first part of the query subject to semantic processing by the QP is *Cats*. By its position within the query, it should be a collection; in fact, it is type-checked (if possible) to see that it *is* a collection and, if not, an error is generated. If it is indeed a collection, the QP locates it in the symbol table (maintained by the preprocessor) and retrieves its type, which it stores within the *yacc* rule for this collection argument.

Upon encountering the lambda variable *c* the QP pushes it onto the lambda variable stack (along with a flag indicating a new scope has begun), and also derives its type by retrieving the *memType* property of the type of *Cats*.

Having just processed a lambda variable declaration and knowing that the current query is a *Select*, the QP expects a predicate to be upcoming. As part of processing the predicate, the left side of the predicate (*c@name*) is parsed and its type derived by first getting the type of c from the lambda variable stack, then applying the *GetPropertyType* operator to the type with the property *name* (and type-checking to ensure the validity of *name* for *c*), then getting the type of the retrieved property object. In addition, the text for this expression is transformed into an ENCORE operation invocation, *GET_PROP_VALUE(c,"name",(long)0)*. (The final argument is a flag which is not used by generated code.)

The right side of the predicate ("Phydeaux") is easily handled, being recognized as a string by its enclosing double-quotes. No type-derivation is needed, and the text generated is *ENFromString("Phydeaux")*, which takes the string argument and produces a corresponding string *object*.

With code generated for both halves of the predicate, the QP now generates a call to the ENCORE boolean operation for equality, with the two expressions as arguments:

> !EnUidCmp(GET_PROP_VALUE(c,"name",(long)0), EN-FromString("Phydeaux"))

This predicate-code comprises part of a boolean operation which is then generated. This operation consists primarily of an if-clause, with the predicate-code as the clause, which returns EN_TRUE or EN_FALSE, the ENCORE boolean value-objects.

Arguments for this operation (which is given a unique name) consist of all symbols on the lambda variable stack corresponding to the current scope (the "selves" of the routine), and a variable *args*, which contains any pre-existing collections (like *Cats*) involved in the query. *Args* also contains any lambda variables from scopes previous to the current one, as in the case of nested queries. These values are extracted from *args* via the *ENGetArg* operation, and are assigned to local variables in the generated code.

The only other local variables in the generated operation are those corresponding to any other generated operation. The code for declaration and initialization of these operation objects is maintained by generating a declaration for each operation as it is generated, and also an initialization consisting of a call to the function *GetObjFromUID*. The argument to this function is the UID assigned to the operation object for the operation when it was generated. (In this particular example, no other operations have been generated.)

Finally, code is generated which adds the "selves" of the generated operation to *args* in the event that they are needed in subsequent scopes. Returning to our example, this results in:

```
ENObject
Cat_PredOp113527590(c, args)
ENObject c;
ENObject args;
{
   ENObject Cats;
   Cats = ENGetArg(args, "Cats");
   ENAddArgs(&args, 1, "c", &c);
   if(!EnUidCmp(GET_PROP_VALUE(c,"name",(long)0),(ENFromString("Phydeaux
   {
      return(EN_TRUE);
   }
   else
   {
      return(EN_FALSE);
   }
}
```

(Notice how the name of the operation is prefixed with the name of the type of the *self* argument.)

This operation, once completely generated, is then automatically compiled, made into an object of type *OperationType*, and installed in the database. This assigns the operation a UID, which can be used to retrieve and re-use the operation object subsequently.

The final step taken by the QP is to generate a "main function" that invokes the query method:

```
ENObject
Query37741575(Cats)
ENObject Cats;
{
    ENObject args;
    ENObject PredOp113527590;
    args = ENBuildArgList(1,"Cats," &Cats);
    PredOp113527590 = GetObjFromUID(1685,2963);
    return(INVOKE(Cats, "Select", 2L, 0L, (long)0, (long)0, Pre-
dOp113527590, args));
}
```

A string containing this function definition is passed back to the preprocessor.

7 Interaction With Pre-Processor

As discussed in the Design Overview, the QP receives its input from a specialized preprocessor which accepts programs written in C "extended" by support of ENCORE types and functions. Queries may be embedded in such programs by encasing the query text within %{ and %}, similar to the enclosing ##'s used by INGRES [Sto76]. This enables the preprocessor to recognize the beginning and end of a "query string", which is passed in its entirety to the QP.

Upon receiving the query-function definition from the QP as described in the previous section, the preprocessor can then simply place this function text in the code after the function in which the

query is embedded. The "header line" of the function is then used to replace the original query call. This is done in order to preserve line numbering within the source.

This special preprocessor's primary duties (in addition to invoking the QP) are to convert object-oriented "casts" of variables to calls on ENCORE casting functions, and to do type-checking of expressions when possible. The converted code is then compiled and executed in typical fashion.

8 Selected Open Problems

8.1 query optimization

As stated in the Design Overview, although the QP's current implementation does not include query optimization, for reasons of expediency and efficiency. However, the implementation of a query optimizer based on models discussed in [Zdo89] would be of considerable use in examining the feasibility of such models.

8.2 improved error checking

Currently, the QP handles errors in query syntax, and well as unresolved references, duplicate definitions and type inconsistencies. Obviously there exists room and need for further and more comprehensive error checking – for instance, collections involved together in *Ojoin* operations could be examined for possible incompatibility.

8.3 removal of extraneous variables

Because of the nature of the lambda variable and collection stacks, the QP will, in some cases, generate definition and initialization code for local variables that are unused in the function currently being constructed. Although this has no effect on the output of the query, it would be desirable to generate only those symbols referenced by the current function or those at deeper levels of scoping.

8.4 duplicate collection references

Currently, due to the nature of the ENCORE methods for iterating through collection members, the QP does not allow a collection to be referenced more than once in a query.

9 References

[All76] E. Allman et al., "Embedding A Data Manipulation Language In A General Purpose Programming Language", Proc. 1976 SIGPLAN-SIGMOD Conf. On Data Abstraction, Salt Lake City, UT, March 1976.

[Ant77] F. Antonacci et al., "Structure And Implementation of A Relational Query Language For The Problem Solver", Proc. VLDB Conf., Tokyo, Japan, 1977.

[Ban87] F. Bancilhon, "FAD, A Powerful And Simple Database Language", Proc. 13th VLDB Conf., Brighton, 1987.

[Car86] M. Carey et al., "The Architecture Of The EXODUS Extensible DBMS", Proc. International Workshop On Object-Oriented Database Systems, Pacific Grove, CA, September 1986.

[Gra87] G. Graefe, "The EXODUS Optimizer Generator", Proc. SIGMOD Conf., San Francisco, CA, May 1987.

[Kho86] S. Khoshafian, "Object Identity", Proc. 1st International OOSPLA Conf., Portland, OR, October 1986.

[Nix87] B. Nixon, "Implementation Of A Compiler For A Semantic Data Model: Experiences With TAXIS", Proc. SIGMOD Conf., San Francisco, CA, 1987.

[Rep84] T. Reps et al., "The Synthesizer Generator", Proc. ACM SIGPLAN-SIGSOFT Sofware Engineering Symposium On Practical Software Engineering Developments", Pittsburgh, PA, April 1984.

[Sha89] G. Shaw et al., "A Query Algebra For Object-Oriented Databases", Brown University Technical Report CS-89-19, March 1989.

[Ska86] A. Skarra et al., "An Object Server For An Object-Oriented System", Proc. International Workshop On Database Systems, September 1986.

[Sto76] M. Stonebraker et al., "The Design And Implementation Of

INGRES", ACM Transactions On Database Systems, vol.1 , no.3, September 1976.
[Zdo86] S. Zdonik et al., "Language And Methodology For Object-Oriented Database Environments", Proc. 19th Annual International Conf. on System Sciences, Honolulu, Hawaii, January 1986.
[Zdo89] S. Zdonik, "Query Optimization For Object-Oriented Databases", Proc. 22th Annual International Conf. on System Sciences, Honolulu, Hawaii, January 1989.

```
%{

/* THINGS TO DO:

  - Remove REPLACES
  - More efficient method for typedef names
  - preprocessor syntax

  - Handle #includes
  - Handle C_INVOK E

*/

/*** include files */
#include <stdio.h>
#include <string.h>

/*** ENCORE stuff ***/
#include "entypedef.h"
#include "englobal.h"
#include "enmacro.h"
/***********/

/*** local definitions */
#define STACK_SIZE      20
#define GLOBAL 0                          /* global scope */
#define MAX_ARGUMENTS 30                  /* maximum variable arguments */
#define MAX_TYPEDEF_NAMES 100             /* maximum typedef names allowed */

static struct {
        char    *body[STACK_SIZE];
        short   top;
} term_stack;


typedef struct frame_item {             /* describes a lambda variable */
        char *name;                     /* name of variable */
        int first_in_frame;             /* Is this the 1st variable defined */
                                        /* current scope? */
        ENType type;                    /* type of variable */
        struct frame_item *next, *prev; /* ptrs to next/prev variables */
                                        /* on the stack */
        } param_descr;

typedef struct coll {                   /* describes a collection */
        char *name;                     /* name of collection */
        ENType type;                    /* type of collection */
        struct coll *next, *prev;       /* ptrs to next/prev collections */
                                        /* in the list */
        } coll_descr;

typedef struct {
        char *text;
        int numpairs;
        } DUPLE;

typedef struct {
        char *text;
        ENType type;
        } VAR;

/**********NOW IN  symtabent.h ********
typedef struct symbol_table_entry {
  char *symbol_name;
  char *symbol_type;
```

```
  char *code;
  int symbol_scope;
  struct symbol_table_entry *next;
} SYMBOL;
************************************/
#include "symtabent.h"

typedef struct code_info {                  /* info returned by expressions */
  int                   changed;
  int                   use_l_invoke;
  int                   num_exprs;
  char                  *old_code;
  char                  *new_code;
  ENType                en_type;
  char                  *type;
  char                  *actual_type;
  struct code_info      *next;
} CODE;

/*** variables defined in this module */

/* points to beginning of lambda-variable stack */
/* containing all variables available within current scope */
static param_descr *StackFrame;

/* points to beginning of list containing all */
/* collections referenced thus far */
static coll_descr *Coll_List;

SYMBOL typedef_names;       /* names that have been typedef'ed */

SYMBOL symbol_table;        /* symbol table */

int current_scope;                      /* keeps track of current scope */

char *arg_list[MAX_ARGUMENTS];              /* used for variable args */

static char *includes;

static ENType   callerType;

FILE            *yy_file_desc;

/* declarations of all operations (but not collections) generated thus far*/
char    *declarations;

/* initializations of all operation-objects generated thus far */
char    *initializations;

/* forward declaration for generated functions */
char    *fwd_decl;

/* 1 if type-checking is to be done at runtime */
int     runtime_check = 1;

/* generated code (from embedded queries) */
/* which will be appended to original source */
char *query_functions;

/*** variables defined in other modules */
extern char *yy_filename;
extern int line_counter;                    /* current line of source file */

/*** function forward declarations */
char *malloc (), *realloc (), *Concat (), *Concat_With_Spaces (),
```

```
     *Coerce_Type (), *Lookup_Encore_Property_Type (),
     *Lookup_Type ();
SYMBOL *Add_To_List (), *Build_Symbol ();
CODE *Allocate_Code_Block ();
char * GetBaseType();
char *pop_term();
bool is_empty_term();

extern  ENType  GetTypeObject();
extern  ENOperationType predop;
extern  ENObject GetObjFromUID();

%}

%start program

%union {
  int ival;
  char *sval;
  SYMBOL *sym_val;
  CODE *code_info;
  char *name;
  DUPLE *duple;
  VAR *var;
}

%token SELECT IMAGE PROJECT OJOIN FLATTEN NEST UNNEST IN SUBSETOF
%token NOT TRUE_TOKEN FALSE_TOKEN DUPELIM COALESCE
%type <name> pred explist attrname tuple pair number query_prog
%type <duple> pairlist
%type <var> variable exp t f query obj func funcname image_func

%token <sval> LAMBDA
%token <sval> en_identifier
%token <sval> sym_identifier
%token <sval> sym_key_auto
%token <sval> sym_key_break
%token <sval> sym_key_case
%token <sval> sym_key_char
%token <sval> sym_key_continue
%token <sval> sym_key_default
%token <sval> sym_key_do
%token <sval> sym_key_double
%token <sval> sym_key_else
%token <sval> sym_key_entry
%token <sval> sym_key_extern
%token <sval> sym_key_float
%token <sval> sym_key_for
%token <sval> sym_key_goto
%token <sval> sym_key_if
%token <sval> sym_key_int
%token <sval> sym_key_long
%token <sval> sym_key_register
%token <sval> sym_key_return
%token <sval> sym_key_short
%token <sval> sym_key_sizeof
%token <sval> sym_key_static
%token <sval> sym_key_struct
%token <sval> sym_key_switch
%token <sval> sym_key_typedef
%token <sval> sym_key_unsigned
%token <sval> sym_key_while
%token <sval> sym_constant
%token <sval> sym_string
%token <sval> sym_op_plus
```

```
%token <sval> sym_op_minus
%token <sval> sym_op_mult
%token <sval> sym_op_div
%token <sval> sym_op_mod
%token <sval> sym_op_shift
%token <sval> sym_op_rel
%token <sval> sym_op_eq
%token <sval> sym_op_and
%token <sval> sym_op_or
%token <sval> sym_op_bit_and
%token <sval> sym_op_bit_or
%token <sval> sym_op_bit_xor
%token <sval> sym_op_unary
%token <sval> sym_op_inc
%token <sval> sym_asgn
%token <sval> sym_op_asgn
%token <sval> sym_comma
%token <sval> sym_period
%token <sval> sym_pound
%token <sval> sym_arrow
%token <sval> sym_question
%token <sval> sym_semi
%token <sval> sym_colon
%token <sval> sym_l_parn
%token <sval> sym_r_parn
%token <sval> sym_l_sbracket
%token <sval> sym_r_sbracket
%token <sval> sym_l_brace
%token <sval> sym_r_brace
%token <sval> sym_at
%token <sval> sym_tick
%token <sval> sym_typedef_name
%token <sval> sym_query_start
%token <sval> sym_query_end
%token <sval> error

%type <sval> optional_attributes
%type <sval> attributes
%type <sval> sc_specifier
%type <sval> type_specifier
%type <sval> a_type
%type <sval> struct_specifier
%type <sval> struct_header
%type <sval> type_name
%type <sval> abstract_declarator

%type <code_info> constant_expression
%type <code_info> expression_list
%type <code_info> expression
%type <code_info> _expression
%type <code_info> en_expression
%type <code_info> constant_expression
%type <code_info> term
%type <code_info> en_term
%type <code_info> function_prefix

%type <code_info> optional_expression

%type <sym_val> declarator_list
%type <sym_val> declarator
%type <sym_val> non_function_declarator
%type <sym_val> function_declarator
%type <sym_val> name_with_l_parn
%type <sym_val> init_declarator_list
%type <sym_val> init_declarator
```

```
%type <sval> program
%type <sval> external_definition
%type <sval> data_definition
%type <sval> function_body
%type <sval> arg_declaration_list
%type <sval> declaration
%type <sval> declaration_statement_list
%type <sval> optional_semi
%type <sval> type_declaration
%type <sval> type_decl_list
%type <sval> parameter_list
%type <sval> initializer_list
%type <sval> initializer
%type <sval> optional_comma
%type <sval> compound_statement
%type <sval> statement_list
%type <sval> statement
%type <sval> label
%type <sval> do_prefix
%type <sval> for_prefix
%type <sval> if_prefix
%type <sval> if_else_prefix
%type <sval> switch_prefix
%type <sval> while_prefix
%type <sval> required_declaration_statement_list

%left sym_comma
%right sym_op_asgn sym_asgn
%right sym_question sym_colon
%left sym_op_or
%left sym_op_and
%left sym_op_bit_or
%left sym_op_bit_xor
%left sym_op_bit_and
%left sym_op_eq
%left sym_op_rel
%left sym_op_shift
%left sym_op_plus sym_op_minus
%left sym_op_mult sym_op_div sym_op_mod
%right sym_op_unary
%right sym_op_inc sym_key_sizeof
%left sym_l_parn sym_l_sbracket sym_arrow sym_period sym_at sym_pound

%%

program:
          program external_definition
                {
#ifdef DEBUG
fprintf(debug,"program --> program external_definition\n");
fflush(debug);
#endif DEBUG
/*
                        $$ = malloc(strlen($1) + strlen($2) + strlen(query_functions) + 1);
                        strcpy($$, "");
                        strcat($$, $1);
                        strcat($$, $2);
                        strcat($$, query_functions);
                        fwrite($$, 1, strlen($$), yy_file_desc);
*/

                        fwrite($2, 1, strlen($2), yy_file_desc);

                        if (*query_functions != '0')
```

```
                        {
                                fwrite(query_functions, 1,
                                        strlen(query_functions), yy_file_desc);
                                query_functions[0] = '\0';
                        }
                }
        |
                {
#ifdef DEBUG
fprintf(debug,"program --> <NULL>\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(1);
                        strcpy($$,"");
                }
        ;

external_definition:
          data_definition
                {
#ifdef DEBUG
fprintf(debug,"external_definition --> data_definition\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + 1);
                        strcpy($$, $1);
                }
        | sym_pound sym_constant sym_string sym_constant
                {
#ifdef DEBUG
fprintf(debug,"external_definition --> sym_pound sym_constant sym_string sym_constant\n");
fflush(debug);
#endif DEBUG
                        arg_list[0] = $1;
                        arg_list[1] = $2;
                        arg_list[2] = $3;
                        arg_list[3] = $4;
                        arg_list[4] = "\n";
                        $$ = Concat_With_Spaces(5,arg_list);
                        goto get_new_line_no;
                }
        | sym_pound sym_constant sym_string
                {
                        char *fname;
#ifdef DEBUG
fprintf(debug,"external_definition --> sym_pound sym_constant sym_string\n");
fflush(debug);
#endif DEBUG
                        arg_list[0] = $1;
                        arg_list[1] = $2;
                        arg_list[2] = $3;
                        arg_list[3] = "\n";
                        $$ = Concat_With_Spaces(4,arg_list);
        get_new_line_no:

                        fname = &($3[1]);
                        fname[strlen(fname) - 1] = '\0';
                        if (!strcmp(fname,yy_filename))
                        {
                                line_counter = atoi($2) - 1;
                        }
                }
        | error
        ;
```

```
data_definition:
          optional_attributes sym_semi
                { /* what happens - there are no symbols being declared */
#ifdef DEBUG
fprintf(debug,"data_definition --> optional_attributes sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 1);
                        strcpy($$,$1);
                        strcat($$,$2);
                        strcat($$,"\n");
                }
        | optional_attributes init_declarator_list sym_semi
                {
#ifdef DEBUG
fprintf(debug,"data_definition --> optional_attributes init_declarator sym_semi\n");
fflush(debug);
#endif DEBUG
                 /* if this is a typedef, then add the name(s) being
                    defined to the list of typedef names */
#ifdef DEBUG
fprintf(debug,"data_definition rule #2 -- $1 = '%s'\n", $1);
fprintf(debug,"data_definition rule #2 -- $2->code = '%s'\n", $2->code);
fflush(debug);
#endif DEBUG
                 if ($1 && !strncmp ($1,"typedef",7))
                 {
                       Add_Typedef_Names ($2);
                       $$ = malloc(strlen($1) + strlen($2->code) + strlen($3) + 3);
                       strcpy($$,$1);
                 }
                 /* else add the name(s) being defined to the symbol table */
                 else
                 {
                       Add_Symbol_Names ($1,$2,GLOBAL);

                       if (Is_Encore_Type($1) && (!Is_Existing_Encore_Type($1)))
                       {
                              $$ = malloc(strlen("ENObject") + strlen($2->code) + strlen($3) + 3);
                              strcpy($$,"ENObject");
                       }
                       else
                       {
                              $$ = malloc(strlen($1) + strlen($2->code) + strlen($3) + 3);
                              strcpy($$,$1);
                       }
                 }
                       strcat($$, " ");
                       strcat($$,$2->code);
                       strcat($$,$3);
                       strcat($$,"\n");
                }
        | optional_attributes function_declarator
                {
#ifdef DEBUG
fprintf(debug,"data_definition --> optional_attributes function_declarator ...\n");
fflush(debug);
#endif DEBUG
                        Enter_Scope ();

                        /* Since this is the start of a new function, */
                        /* we re-initialize our list of forward declarations */
                        fwd_decl = "";

                }
```

```
        function_body   /* Should we Add_Symbol here as well? */
                {
#ifdef DEBUG
fprintf(debug,"...function_body:\n");
fflush(debug);
#endif DEBUG

                        /* Forward declarations are placed immediately */
                        /* before the function currently being preprocessed */
                        if (Is_Encore_Type($1) && (!(Is_Existing_Encore_Type($1))))
                        {
                                if (*fwd_decl)
                                {
                                        $$ = malloc(strlen(fwd_decl) + strlen("ENObject") + strlen($2->code) + strlen($4) + 4);
                                        strcpy($$, fwd_decl);
                                        strcat($$, "ENObject");
                                        fwd_decl = "";
                                }
                                else
                                {
                                        $$ = malloc(strlen("ENObject") + strlen($2->code) + strlen($4) + 4);
                                        strcpy($$, "ENObject");
                                }
                        }
                        else
                        {
                                if (*fwd_decl)
                                {
                                        $$ = malloc(strlen(fwd_decl) + strlen($1) + strlen($2->code) + strlen($4) + 4);
                                        strcpy($$, fwd_decl);
                                        fwd_decl = "";
                                        strcat($$, $1);
                                }
                                else
                                {
                                        $$ = malloc(strlen($1) + strlen($2->code) + strlen($4) + 4);
                                        strcpy($$, $1);
                                }
                        }
                        strcat($$, "\n");
                        strcat($$,$2->code);
                        strcat($$,"\n");
                        strcat($$,$4);
                        strcat($$,"\n");
                        Exit_Scope ();
                }
        ;

function_body:
          arg_declaration_list
                {
                        char    *typeName;
                        bool    isPtr;

                        Call_NewArgs(predop,0);
                        typeName = GetBaseType(Lookup_Type("self"),&isPtr);
                        callerType = GetTypeObject(typeName);
                }
                                compound_statement
                {
#ifdef DEBUG
fprintf(debug,"function_body --> arg_declaration compound_statement\n");
fflush(debug);
#endif DEBUG
                        Display_Symbol_Table();
```

```
                        $$ = malloc(strlen($1) + strlen($3) + 1);
                        strcpy($$,$1);
                        strcat($$,$3);
                }
        ;

arg_declaration_list:
          arg_declaration_list declaration
                {
#ifdef DEBUG
fprintf(debug,"arg_declaration_list --> arg_declaration_list declaration\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 1);
                        strcpy($$,$1);
                        strcat($$,$2);
                }
        |
                {
#ifdef DEBUG
fprintf(debug,"arg_declaration_list --> <NULL>\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(1);
                        strcpy($$,"");
                }
        ;

declaration:
          attributes declarator_list sym_semi
                {
#ifdef DEBUG
fprintf(debug,"declaration --> attributes declarator_list sym_semi\n");
fflush(debug);
#endif DEBUG
                /* add symbol names to the symbol table */
                Add_Symbol_Names ($1,$2,current_scope);

                if (Is_Encore_Type($1) && (!Is_Existing_Encore_Type($1)))
                {
                        $$ = malloc(strlen("ENObject") + strlen($2->code) + strlen($3) + 3);
                        strcpy($$,"ENObject");
                }
                else
                {
                        $$ = malloc(strlen($1) + strlen($2->code) + strlen($3) + 3);
                        strcpy($$,$1);
                }
                        strcat($$, " ");
                        strcat($$,$2->code);
                        strcat($$,$3);
                        strcat($$,"\n");
                }
        | attributes sym_semi
                {
#ifdef DEBUG
fprintf(debug,"declaration --> sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$,$1);
                        strcat($$,$2);
                        strcat($$,"\n");
                }
        | error sym_semi
```

```
        ;

required_declaration_statement_list:
          declaration_statement_list attributes sym_semi
                { /* what happens - there are no symbols being declared */
#ifdef DEBUG
fprintf(debug,"required_declaration_list --> declaration_statement_list attributes sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + 4);
                        strcpy($$,$1);
                        strcat($$, " ");
                        strcat($$,$2);
                        strcat($$, " ");
                        strcat($$,$3);
                        strcat($$,"\n");
                }
        | declaration_statement_list attributes init_declarator_list sym_semi
                { char *type_name;
#ifdef DEBUG
fprintf(debug,"required_declaration_list --> declaration_statement_list attributes init_declarator_list sym_semi\n");
fflush(debug);
#endif DEBUG
                 /* if this is a typedef, then add the name(s) being
                    defined to the list of typedef names */
                 if ($2 && !strncmp ($2,"typedef",7))
                 {
                       Add_Typedef_Names ($3);
                       type_name = malloc(strlen($2)+1);
                       strcpy(type_name, $2);
                 }
                 /* else add the name(s) being defined to the symbol table */
                 else
                 {
                       Add_Symbol_Names ($2,$3, current_scope);

                       if (Is_Encore_Type($2) && (!Is_Existing_Encore_Type($2)))
                       {
                               type_name = malloc(strlen("ENObject")+1);
                               strcpy(type_name, "ENObject");
                       }
                       else
                       {
                               type_name = malloc(strlen($2)+1);
                               strcpy(type_name, $2);
                       }
                 }
                       $$ = malloc(strlen($1) + strlen(type_name) + strlen($3->code) + strlen($4) + 4);
                       strcpy($$,$1);
                       strcat($$, " ");
                       /*
                       strcat($$,$2);
                       */
                       strcat($$,type_name);
                       strcat($$, " ");
                       strcat($$,$3->code);
                       strcat($$,$4);
                       strcat($$,"\n");
                }
        ;

declaration_statement_list:
          declaration_statement_list attributes sym_semi
                { /* what happens - there are no symbols being declared */
#ifdef DEBUG
```

```
fprintf(debug,"declaration_statement_list --> declaration_statement_list attributes sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + 4);
                        strcpy($$,$1);
                        strcat($$, " ");
                        strcat($$,$2);
                        strcat($$, " ");
                        strcat($$,$3);
                        strcat($$,"\n");
                }
        | declaration_statement_list attributes init_declarator_list sym_semi
                { char *type_name;
#ifdef DEBUG
fprintf(debug,"declaration_statement_list --> declaration_statement_list attributes init_declarator_list sym_semi\n");
fflush(debug);
#endif DEBUG
                  /* if this is a typedef, then add the name(s) being
                     defined to the list of typedef names */
                  if ($2 && !strncmp ($2,"typedef", 7))
                  {
                        Add_Typedef_Names ($3);
                        type_name = malloc(strlen($2)+1);
                        strcpy(type_name, $2);
                  } /* else add the name(s) being defined to the symbol table */
                  else
                  {
                        Add_Symbol_Names ($2,$3, current_scope);

                        if (Is_Encore_Type($2) && (!Is_Existing_Encore_Type($2)))
                        {
                                type_name = malloc(strlen("ENObject")+1);
                                strcpy(type_name, "ENObject");
                        }
                        else
                        {
                                type_name = malloc(strlen($2)+1);
                                strcpy(type_name, $2);
                        }

                  }
                        $$ = malloc(strlen($1) + strlen(type_name) + strlen($3->code) + strlen($4) + 4);
                        strcpy($$,$1);
                        strcat($$, " ");
                        /*
                        strcat($$,$2);
                        */
                        strcat($$,type_name);
                        strcat($$, " ");
                        strcat($$,$3->code);
                        strcat($$, $4);
                        strcat($$,"\n");
                }
        |
                {
#ifdef DEBUG
fprintf(debug,"declaration_statement_list --> <NULL>\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(1);
                  strcpy($$, "");
                }
        ;

optional_attributes:
```

```
          attributes
                {
#ifdef DEBUG
fprintf(debug,"optional_attributes --> attributes\n");
fflush(debug);
#endif DEBUG
                        $$ = $1;
                }
        |
                {
#ifdef DEBUG
fprintf(debug,"optional_attributes --> <NULL>\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(1);
                        strcpy($$, "");
                }
        ;

attributes:
          sc_specifier type_specifier
                {
#ifdef DEBUG
fprintf(debug,"attributes --> sc_specific type_specifier\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$,$1);
                        strcat($$, " ");
                        strcat($$,$2);
                }
        | type_specifier sc_specifier
                {
#ifdef DEBUG
fprintf(debug,"attributes --> type_specifier sc_specifier\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$,$1);
                        strcat($$, " ");
                        strcat($$,$2);
                }
        | sc_specifier
                {
#ifdef DEBUG
fprintf(debug,"attributes --> sc_specific\n");
fflush(debug);
#endif DEBUG
                        $$ = $1;
                }
        | type_specifier
                {
                        $$ = $1;
#ifdef DEBUG
fprintf(debug,"attributes --> type_specifier %s\n", $$);
fflush(debug);
#endif DEBUG
                }
        | type_specifier sc_specifier type_specifier
                {
#ifdef DEBUG
fprintf(debug,"attributes --> type_specifier sc_specific type_specifier\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + 3);
```

```
                        strcpy($$,$1);
                        strcat($$, " ");
                        strcat($$,$2);
                        strcat($$, " ");
                        strcat($$,$3);
                }
                /* REPLACE - can't think of what this would be */
        ;

sc_specifier:
          sym_key_auto
                {
                        $$ = malloc(5);
                        strcpy($$,$1);
                }
        | sym_key_static
                {
                        $$ = malloc(7);
                        strcpy($$,$1);
                }
        | sym_key_extern
                {
                        $$ = malloc(7);
                        strcpy($$,$1);
                }
        | sym_key_register
                {
                        $$ = malloc(9);
                        strcpy($$,$1);
                }
        | sym_key_typedef
                {
                        $$ = malloc(8);
                        strcpy($$,$1);
                }
        ;

type_specifier:
          a_type
                {
#ifdef DEBUG
fprintf(debug,"type_specifier --> a_type\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
        | a_type a_type
                {
#ifdef DEBUG
fprintf(debug,"type_specifier --> a_type a_type\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$,$1);
                        strcat($$," ");
                        strcat($$,$2);
                }
        | a_type a_type a_type
                {
#ifdef DEBUG
fprintf(debug,"type_specifier --> a_type a_type a_type\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + 3);
```

```
                        strcpy($$,$1);
                        strcat($$," ");
                        strcat($$,$2);
                        strcat($$," ");
                        strcat($$,$3);
                }
        | struct_specifier
                {
#ifdef DEBUG
fprintf(debug,"type_specifier --> struct_specifier\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
        | sym_typedef_name
                {
#ifdef DEBUG
fprintf(debug,"type_specifier --> sym_typedef_name\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
        ;

a_type:
          sym_key_char
                {
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
        | sym_key_short
                {
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
        | sym_key_int
                {
#ifdef DEBUG
fprintf(debug,"a_type --> sym_key_int\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
        | sym_key_long
                {
#ifdef DEBUG
fprintf(debug,"a_type --> sym_key_long\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
        | sym_key_unsigned
                {
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
        | sym_key_float
                {
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
```

```
        | sym_key_double
                {
                        $$ = malloc(strlen($1)+1);
                        strcpy($$,$1);
                }
        ;

struct_specifier:
          struct_header sym_l_brace type_decl_list optional_semi sym_r_brace
                {
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + strlen($4) + strlen($5) + 2);
                        strcpy($$,$1);
                        strcat($$,$2);
                        strcat($$,"\n");
                        strcat($$, $3);
                        strcat($$, $4);
                        strcat($$, $5);
                }
        | sym_key_struct sym_identifier
                {
                  arg_list[0] = $1;
                  arg_list[1] = $2;

                  $$ = Concat_With_Spaces (2,arg_list);
                }
        ;

optional_semi:
                {
                        $$ = malloc(1);
                  strcpy($$, "");
                }
        | sym_semi
                {
                        $$ = malloc(2);
                        strcpy($$,$1);
                        strcat($$,"\n");
                }
        ;

struct_header:
          sym_key_struct
                {
                        $$ = malloc(7);
                        strcpy($$,$1);
                }
        | sym_key_struct sym_identifier
                {
                  arg_list[0] = $1;
                  arg_list[1] = $2;

                  $$ = Concat_With_Spaces (2,arg_list);
                }
        ;

type_decl_list:
          type_declaration
                {
                        $$ = $1;
                }
        | type_decl_list sym_semi type_declaration
                {
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + 2);
                        strcpy($$,$1);
                        strcat($$,$2);
```

```
                        strcat($$,"\n");
                        strcat($$,$3);
                }
        ;

type_declaration:
          type_specifier declarator_list
                {
#ifdef DEBUG
fprintf(debug,"type_declaration --> type_specifier declarator_list\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2->code) + 2);
                        strcpy($$,$1);
                        strcat($$," ");
                        strcat($$,$2->code);
                }
        | type_specifier
                {
#ifdef DEBUG
fprintf(debug,"type_declaration --> type_specifier \n");
fflush(debug);
#endif DEBUG
                        $$ = $1;
                }
        ;

declarator_list:
          declarator
                {
                        $$ = $1;
                }
        | sym_typedef_name
                { /* HACK for cases where uninitialized name is also a
                     typedef_name - in argument declarations and
                     struct field declarations */

                  $$ = Build_Symbol ($1);
                  $$->code = malloc(strlen($1)+1);
                  strcpy($$->code, $1);
                }
        | declarator_list sym_comma declarator
                {
                        char *save1 = malloc(strlen($1->code)+1);
                        char *save3 = malloc(strlen($3->code)+1);

                        strcpy(save1, $1->code);
                        strcpy(save3, $3->code);

                        $$ = Add_To_List ($1,$3);
                        $$->code = malloc(strlen(save1) + strlen($2) + strlen(save3)+1);
                        strcpy($$->code, save1);
                        strcat($$->code, $2);
                        strcat($$->code, save3);
                }
        | declarator_list sym_comma sym_typedef_name
                { /* HACK for cases where uninitialized name is also a
                     typedef_name - in argument declarations and
                     struct field declarations */

                  char *save1 = malloc(strlen($1->code)+1);

                  strcpy(save1, $1->code);
                  $$ = Add_To_List ($1,Build_Symbol($3));
                  $$->code = malloc(strlen(save1) + strlen($2) + strlen($3)+1);
```

```
                  strcpy($$->code, savel);
                  strcat($$->code, $2);
                  strcat($$->code, $3);
                }
        ;

declarator:
          function_declarator
                {
                        $$ = $1;
                }
        | non_function_declarator
                {
                        $$ = $1;
                }
        | non_function_declarator sym_colon constant_expression
                %prec sym_comma
                {
                        $$->code = malloc(strlen($1->code) + strlen($2) + strlen($3)+1);
                        strcpy($$->code, $1->code);
                        strcat($$->code, $2);
                        strcat($$->code, $3);
                }
        | sym_colon constant_expression
                %prec sym_comma
                {
                        $$->code = malloc(1);
                  strcpy($$->code, "");
                }
        | error
                { $<sym_val>$ = 0; }
        ;

non_function_declarator:
          sym_op_mult non_function_declarator
                {
                  char *save = malloc(strlen($2->code)+1);
#ifdef DEBUG
fprintf(debug,"non_function_declarator --> sym_op_mult non_function_declarator \n");
fflush(debug);
#endif DEBUG
                  Update_Type ($2,$1);
                  strcpy(save, $2->code);
                  $$ = $2;
                  $$->code = malloc(strlen($1) + strlen($2->code)+1);
                  strcpy($$->code, $1);
                  strcat($$->code, save);
                }
        | non_function_declarator sym_l_parn sym_r_parn
                {
                  char *save = malloc(strlen($1->code)+1);
#ifdef DEBUG
fprintf(debug,"non_function_declarator --> non_function_declarator sym_l_parn sym_r_parn\n");
fflush(debug);
#endif DEBUG
                  arg_list[0] = $2;
                  arg_list[1] = $3;

                  strcpy(save, $1->code);
                  Update_Type ($1,Concat(2,arg_list));
                  $$ = $1;
                  $$->code = malloc(strlen(save) + strlen($2) + strlen($3)+1);
                  strcpy($$->code, save);
                  strcat($$->code, $2);
                  strcat($$->code, $3);
```

```
                }
        | non_function_declarator sym_l_sbracket sym_r_sbracket
                {
                  char *save = malloc(strlen($1->code)+1);
#ifdef DEBUG
fprintf(debug,"non_function_declarator --> non_function_declarator sym_l_bracket sym_r_bracket\n");
fflush(debug);
#endif DEBUG
                  arg_list[0] = $2;
                  arg_list[1] = $3;

                  strcpy(save, $1->code);
                  Update_Type ($1,Concat(2,arg_list));
                  $$ = $1;
                  $$->code = malloc(strlen(save) + strlen($2) + strlen($3)+1);
                  strcpy($$->code, save);
                  strcat($$->code, $2);
                  strcat($$->code, $3);
                }
        | non_function_declarator sym_l_sbracket constant_expression sym_r_sbracket
                {
                  char *save = malloc(strlen($1->code)+1);
#ifdef DEBUG
fprintf(debug,"non_function_declarator --> non_function_declarator sym_l_bracket constant_expression sym_r_bracket\n");
fflush(debug);
#endif DEBUG
                  arg_list[0] = $2;
                  arg_list[1] = $4;

                  strcpy(save, $1->code);
                  Update_Type ($1,Concat(2,arg_list));
                  $$ = $1;
                  $$->code = malloc(strlen(save) + strlen($2) + strlen($3->new_code) + strlen($4)+1);
                  strcpy($$->code, save);
                  strcat($$->code, $2);
                  strcat($$->code, $3->new_code);
                  strcat($$->code, $4);
                }
        | sym_identifier
                {
#ifdef DEBUG
fprintf(debug,"non_function_declarator --> sym_identifier\n");
fflush(debug);
#endif DEBUG
                        $$ = Build_Symbol ($1);
                        $$->code = malloc(strlen($1)+1);
                        strcpy($$->code, $1);
#ifdef DEBUG
fprintf(debug,"$$->code = %s\n", $$->code);
fflush(debug);
#endif DEBUG
                }
        | sym_l_parn non_function_declarator sym_r_parn
                {
                  char *save = malloc(strlen($2->code)+1);
#ifdef DEBUG
fprintf(debug,"non_function_declarator --> sym_l_parn non_function_declarator sym_r_parn\n");
fflush(debug);
#endif DEBUG
                  Update_Type ($2,$1);
                  Update_Type ($2,$3);
                  strcpy(save, $2->code);
                  $$ = $2;
                  $$->code = malloc(strlen($1) + strlen($2->code) + strlen($3)+1);
                  strcpy($$->code, $1);
```

```
                strcat($$->code, save);
                strcat($$->code, $3);
              }
        | sym_typedef_name
              {
#ifdef DEBUG
fprintf(debug,"non_function_declarator --> sym_typedef_name\n");
fflush(debug);
#endif DEBUG
                    $$ = Build_Symbol ($1);
                    $$->code = malloc(strlen($1)+1);
                    strcpy($$->code, $1);
#ifdef DEBUG
fprintf(debug,"$$->code = %s\n", $$->code);
fflush(debug);
#endif DEBUG
              }
        ;

function_declarator:
          sym_op_mult function_declarator
              {
                char *save = malloc(strlen($2->code)+1);
#ifdef DEBUG
fprintf(debug,"function_declarator --> sym_op_mult function_declarator\n");
fflush(debug);
#endif DEBUG
                Update_Type ($2,$1);
                strcpy(save, $2->code);
                $$ = $2;
                $$->code = malloc(strlen($1) + strlen($2->code)+1);
                strcpy($$->code, $1);
                strcat($$->code, save);
#ifdef DEBUG
fprintf(debug,"$$->code = '%s'\n", $$->code);
fflush(debug);
#endif DEBUG
              }
        | function_declarator sym_l_parn sym_r_parn
              {
                char *save = malloc(strlen($1->code)+1);
#ifdef DEBUG
fprintf(debug,"function_declarator --> function_declarator sym_l_parn sym_r_parn\n");
fflush(debug);
#endif DEBUG
                arg_list[0] = $2;
                arg_list[1] = $3;

                strcpy(save, $1->code);
                Update_Type ($1,Concat(2,arg_list));
                $$ = $1;
                $$->code = malloc(strlen(save) + strlen($2) + strlen($3)+1);
                strcpy($$->code, save);
                strcat($$->code, $2);
                strcat($$->code, $3);
#ifdef DEBUG
fprintf(debug,"$$->code = '%s'\n", $$->code);
fflush(debug);
#endif DEBUG
              }
        | function_declarator sym_l_sbracket sym_r_sbracket
              {
                char *save = malloc(strlen($1->code)+1);
#ifdef DEBUG
fprintf(debug," function_declarator --> function_declarator sym_l_sbracket sym_r_sbracket\n");
```

```
fflush(debug);
#endif DEBUG
                    arg_list[0] = $2;
                    arg_list[1] = $3;

                    strcpy(save, $1->code);
                    Update_Type ($1,Concat(2,arg_list));
                    $$ = $1;
                    $$->code = malloc(strlen(save) + strlen($2) + strlen($3)+1);
                    strcpy($$->code, save);
                    strcat($$->code, $2);
                    strcat($$->code, $3);
                  }
        | function_declarator sym_l_sbracket constant_expression sym_r_sbracket
                  {
                    char *save = malloc(strlen($1->code)+1);
#ifdef DEBUG
fprintf(debug," function_declarator --> function_declarator sym_l_sbracket constant_expresion sym_r_sbracket\n");
fflush(debug);
#endif DEBUG
                    arg_list[0] = $2;
                    arg_list[1] = $4;

                    strcpy(save, $1->code);
                    Update_Type ($1,Concat(2,arg_list));
                    $$ = $1;
                    $$->code = malloc(strlen(save) + strlen($2) +
                                strlen($3->new_code) + strlen($4) + 1);
                    strcpy($$->code, save);
                    strcat($$->code, $2);
                    strcat($$->code, $3->new_code);
                    strcat($$->code, $4);
                  }
        | sym_l_parn function_declarator sym_r_parn
                  {
                    char *save = malloc(strlen($2->code) + 1);
                    Update_Type ($2,$1);
                    Update_Type ($2,$3);
                    strcpy(save, $2->code);
                    $$ = $2;
                    $$->code = malloc(strlen($1) + strlen($2->code) +
                                strlen($3)+1);
                    strcpy($$->code, $1);
                    strcat($$->code, save);
                    strcat($$->code, $3);
                  }
        | name_with_l_parn parameter_list sym_r_parn
                  {
                    char *save = malloc(strlen($1->code)+1);
#ifdef DEBUG
fprintf(debug," function_declarator --> name_with_l_parn parameter_list sym_r_parn\n");
fflush(debug);
#endif DEBUG
                    arg_list[0] = "(";
                    arg_list[1] = $3;

                    strcpy(save, $1->code);
                    Update_Type ($1,Concat(2,arg_list));
                    $$ = $1;
                    $$->code = malloc(strlen(save) + strlen($2) + strlen($3)+1);
                    strcpy($$->code, save);
                    strcat($$->code, $2);
                    strcat($$->code, $3);

                  }
```

```
        | name_with_l_parn sym_r_parn
                {
                  char *save = malloc(strlen($1->code)+1);
#ifdef DEBUG
fprintf(debug," function_declarator --> name_with_l_parn sym_r_parn\n");
fflush(debug);
#endif DEBUG
                  arg_list[0] = "(";
                  arg_list[1] = $2;

                  strcpy(save, $1->code);
                  Update_Type ($1,Concat(2,arg_list));
                  $$ = $1;
                  $$->code = malloc(strlen(save) + strlen($2)+1);
                  strcpy($$->code, save);
                  strcat($$->code, $2);
                }
        ;

name_with_l_parn:
          sym_identifier sym_l_parn
                {
                 $$ = Build_Symbol ($1);
                 $$->code = malloc(strlen($1) + strlen($2)+1);
                 strcpy($$->code, $1);
                 strcat($$->code, $2);
                }
        ;

parameter_list:
          sym_identifier
                {
                        $$ = $1;
                }
        | sym_typedef_name
                { /* HACK for cases where uninitialized name is also a
                     typedef_name - in argument declarations and
                     struct field declarations */
                        $$ = $1;
                }
        | parameter_list sym_comma sym_identifier
                {
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3)+1);
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$, $3);
                }
        | parameter_list sym_comma sym_typedef_name
                /* HACK for cases where uninitialized name is also a
                     typedef_name - in argument declarations and
                     struct field declarations */
                {
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3)+1);
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$, $3);
                }
        | error
        ;

init_declarator_list:
          init_declarator
                %prec sym_comma
                {
#ifdef DEBUG
```

```
fprintf(debug,"init_declarator_list --> init_declarator\n");
fprintf(debug,"$1 = '%s'\n", $1->code);
fflush(debug);
#endif DEBUG
                        $$ = $1;
                }
        | init_declarator_list sym_comma init_declarator
                {
                        char *save1 = malloc(strlen($1->code)+1);
                        char *save3 = malloc(strlen($3->code)+1);

#ifdef DEBUG
fprintf(debug,"init_declarator_list --> init_declarator_list sym_comma init_declarator\n");
fflush(debug);
#endif DEBUG
                        strcpy(save1, $1->code);
                        strcpy(save3, $3->code);

                        $$ = Add_To_List ($1,$3);

                        $$->code = malloc(strlen(save1) + strlen($2) +
                                        strlen(save3)+1);
#ifdef DEBUG
fprintf(debug,"save1 = %s\n", save1);
fflush(debug);
#endif DEBUG
                        strcpy($$->code, save1);
#ifdef DEBUG
fprintf(debug,"$2 = %s\n", $2);
fflush(debug);
#endif DEBUG
                        strcat($$->code, $2);
#ifdef DEBUG
fprintf(debug,"save3= %s\n", save3);
fflush(debug);
#endif DEBUG
                        strcat($$->code, save3);
                }
        ;

init_declarator:
         non_function_declarator
                {
#ifdef DEBUG
fprintf(debug,"init_declarator--> non_function_declarator\n");
fflush(debug);
#endif DEBUG
                        $$ = $1;
#ifdef DEBUG
fprintf(debug,"$$->code = %s\n", $$->code);
fflush(debug);
#endif DEBUG
                }
        | function_declarator
                {
#ifdef DEBUG
fprintf(debug,"init_declarator--> function_declarator \n");
fflush(debug);
#endif DEBUG
                        $$ = $1;
                }
        | non_function_declarator sym_asgn expression
                %prec sym_comma
                {
#ifdef DEBUG
```

```
fprintf(debug,"init_declarator--> non_function_declarator sym_asgn expression\n");
fflush(debug);
#endif DEBUG
                        $$ = Build_Symbol("");
                        $$->code = malloc(strlen($1->code) + strlen($2) + strlen($3->new_code) + 1);
                        strcpy($$->code, $1->code);
                        strcat($$->code, $2);
                        strcat($$->code, $3->new_code);
                }
        |  non_function_declarator sym_asgn sym_l_brace initializer_list optional_comma sym_r_brace
                {
#ifdef DEBUG
fprintf(debug,"init_declarator--> non_function_declarator_list sym_asgn sym_l_brace initializer_list optional_comma sym_r_brace\n");
fflush(debug);
#endif DEBUG
                        $$ = Build_Symbol("");
                        $$->code = malloc(strlen($1->code) + strlen($2) + strlen($3) + strlen($4) + strlen($5) + strlen($6) + 1);
                        strcpy($$->code, $1->code);
                        strcat($$->code, $2);
                        strcat($$->code, $3);
                        strcat($$->code, $4);
                        strcat($$->code, $5);
                        strcat($$->code, $6);
                }
        | error
                { $<sym_val>$ = 0; }
        ;

initializer_list:
          initializer
                %prec sym_comma
                {
#ifdef DEBUG
fprintf(debug, "initializer_list --> initializer\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + 1);
                        strcpy($$, $1);
                }
        | initializer_list sym_comma initializer
                {
#ifdef DEBUG
fprintf(debug, "initializer_list --> initializer_list sym_comma initializer\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + 1);
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$, $3);
                }
        ;

initializer:
          expression
                %prec sym_comma
                {
#ifdef DEBUG
fprintf(debug, "initializer --> expression\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1->new_code) + 1);
                        strcpy($$, $1->new_code);
                }
        | sym_l_brace initializer_list optional_comma sym_r_brace
                {
```

```
#ifdef DEBUG
fprintf(debug, "initializer --> sym_l_brace initializer_list optional_comma sym_r_brace\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3)
                                    + strlen($4) + 1);
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$, $3);
                        strcat($$, $4);
                }
        ;

optional_comma:
                {
#ifdef DEBUG
fprintf(debug, "optional_comma --> <NULL>\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(1);
                        strcpy($$, "");
                }
        | sym_comma
                {
#ifdef DEBUG
fprintf(debug, "optional_comma --> sym_comma\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + 1);
                        strcpy($$, $1);
                }
        ;

compound_statement:
          sym_l_brace statement_list sym_r_brace
                {
#ifdef DEBUG
fprintf(debug,"compound_statement --> sym_l_brace statement_list sym_r_brace\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + 3);
                        strcpy($$, $1);
                        strcat($$, "\n");
                        strcat($$, $2);
                        strcat($$, "\n");
                        strcat($$, $3);
                }
        | sym_l_brace
                { Enter_Scope ();
#ifdef DEBUG
fprintf(debug,"compound_statement --> sym_l_brace...\n");
fflush(debug);
#endif DEBUG
                }
          required_declaration_statement_list statement_list sym_r_brace
                {
#ifdef DEBUG
fprintf(debug,"... required_declaration_statement_list statment_list sym_r_brace\n");
fflush(debug);
#endif DEBUG

                        Exit_Scope ();
                        $$ = malloc(strlen($1) + strlen($3) + strlen($4)
                                    + strlen($5) + 3);
                        strcpy($$, "{");
```

```
                        strcat($$, "\n");
                        strcat($$, $3);
                        strcat($$, $4);
                        strcat($$, "\n");
                        strcat($$, "}");
                }
        ;

statement_list:
          statement_list statement
                {
#ifdef DEBUG
fprintf(debug,"statement_list --> statement_list statement\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 1);
                        strcpy($$, $1);
                        strcat($$, $2);
                }
        |
                {
#ifdef DEBUG
fprintf(debug,"statement_list --> <NULL>\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(1);
                        strcpy($$,"");
                }
        ;

statement:
          expression sym_semi
                {
#ifdef DEBUG
fprintf(debug,"statement --> expression sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1->new_code) + strlen($2) + 2);
                        strcpy($$, $1->new_code);
                        strcat($$, $2);
                        strcat($$,"\n");
                }
        | compound_statement
                {
#ifdef DEBUG
fprintf(debug,"statement --> compound_statement\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + 1);
                        strcpy($$, $1);
                }
        | if_prefix statement
                {
#ifdef DEBUG
fprintf(debug,"statement --> if_prefix statement\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$, $1);
                        strcat($$, "\n");
                        strcat($$, $2);
                }
        | if_else_prefix statement
                {
#ifdef DEBUG
```

```
fprintf(debug,"statement --> if_else_prefix statement\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$, $1);
                        strcat($$, "\n");
                        strcat($$, $2);
                }
        | while_prefix statement
                {
#ifdef DEBUG
fprintf(debug,"statement --> while_prefix statement\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$, $1);
                        strcat($$, "\n");
                        strcat($$, $2);
                }
        | do_prefix statement sym_key_while sym_l_parn expression sym_r_parn sym_semi
                {
#ifdef DEBUG
fprintf(debug,"statement --> do_prefix statement sym_key_while  sym_l_parn expression sym_r_parn sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + strlen($4) + strlen($5->new_code) + strlen($6) + strlen($7) + 2);
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$, $3);
                        strcat($$, $4);
                        strcat($$, $5->new_code);
                        strcat($$, $6);
                        strcat($$, $7);
                        strcat($$,"\n");
                }
        | for_prefix optional_expression sym_r_parn statement
                {
#ifdef DEBUG
fprintf(debug,"statement --> for_prefix optional_expression sym_r_parn statement\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2->new_code) + strlen($3) + strlen($4) + 1);
                        strcpy($$, $1);
                        strcat($$, $2->new_code);
                        strcat($$, $3);
                        strcat($$, $4);
                }
        | switch_prefix statement
                {
#ifdef DEBUG
fprintf(debug,"statement --> switch_prefix statement\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$, $1);
                        strcat($$, "\n");
                        strcat($$, $2);
                }
        | sym_key_break sym_semi
                {
#ifdef DEBUG
fprintf(debug,"statement --> sym_key_break sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
```

```
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$,"\n");
                }
        | sym_key_continue sym_semi
                {
#ifdef DEBUG
fprintf(debug,"statement --> sym_key_continue sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$,"\n");
                }
        | sym_key_return sym_semi
                {
#ifdef DEBUG
fprintf(debug,"statement --> sym_key_return sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 2);
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$,"\n");
                }
        | sym_key_return expression sym_semi
                {
#ifdef DEBUG
fprintf(debug,"statement --> sym_key_return expression sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2->new_code)
                                        + strlen($3) + 2);
                        strcpy($$, $1);
                        strcat($$, $2->new_code);
                        strcat($$, $3);
                        strcat($$,"\n");
                }
        | sym_key_goto sym_identifier sym_semi
                {
#ifdef DEBUG
fprintf(debug,"statement --> sym_key_goto sym_identifier sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + strlen($3) + 2);
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$, $3);
                        strcat($$,"\n");
                }
        | sym_semi
                {
#ifdef DEBUG
fprintf(debug,"statement --> sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + 2);
                        strcpy($$, $1);
                        strcat($$,"\n");
                }
        | error sym_semi
        | error sym_r_brace
        | label statement
                {
```

```
#ifdef DEBUG
fprintf(debug,"statement --> label statement\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 1);
                        strcpy($$, $1);
                        strcat($$, $2);
                }
        ;

label:
          sym_identifier sym_colon
                {
#ifdef DEBUG
fprintf(debug,"label --> sym_identifier sym_colon\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 1);
                        strcpy($$, $1);
                        strcat($$, $2);
                }
        | sym_key_case expression sym_colon
                {
#ifdef DEBUG
fprintf(debug,"label --> sym_key_case expression sym_colon\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2->new_code)
                                        + strlen($3) + 1);
                        strcpy($$, $1);
                        strcat($$, $2->new_code);
                        strcat($$, $3);
                }
        | sym_key_default sym_colon
                {
#ifdef DEBUG
fprintf(debug,"label --> sym_key_default sym_colon\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2) + 1);
                        strcpy($$, $1);
                        strcpy($$, $2);
                }
        ;

do_prefix:
          sym_key_do
                {
#ifdef DEBUG
fprintf(debug,"do_prefix --> sym_key_do \n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + 1);
                        strcpy($$, $1);
                }
        ;

for_prefix:
          sym_key_for sym_l_parn optional_expression sym_semi optional_expression sym_semi
                {
#ifdef DEBUG
fprintf(debug,"for_prefix --> sym_key_for_large sym_l_parn optional_expression sym_semi optional_expression sym_semi optional_expression sym_semi\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen($1) + strlen($2)
```

```
                        + strlen($3->new_code) + strlen($4)
                        + strlen($5->new_code) + strlen($6) + 1);
                strcpy($$, $1);
                strcat($$, $2);
                strcat($$, $3->new_code);
                strcat($$, $4);
                strcat($$, $5->new_code);
                strcat($$, $6);
            }
        ;

if_prefix:
         sym_key_if sym_l_parn expression sym_r_parn
            {
#ifdef DEBUG
fprintf(debug,"if_prefix --> sym_key_if sym_l_parn expression sym_r_parn \n");
fflush(debug);
#endif DEBUG
                $$ = malloc(strlen($1) + strlen($2) +
                        strlen($3->new_code) + strlen($4) + 1);
                strcpy($$, $1);
                strcat($$, $2);
                strcat($$, $3->new_code);
                strcat($$, $4);
            }
        ;

if_else_prefix:
          if_prefix statement sym_key_else
            {
#ifdef DEBUG
fprintf(debug,"if_else_prefix --> if_prefix statement sym_key_else \n");
fflush(debug);
#endif DEBUG
                $$ = malloc(strlen($1) + strlen($2) + strlen($3) + 1);
                strcpy($$, $1);
                strcat($$, $2);
                strcat($$, $3);
            }
        ;

switch_prefix:
          sym_key_switch sym_l_parn expression sym_r_parn
            {
#ifdef DEBUG
fprintf(debug,"switch_prefix --> sym_key_switch sym_l_parn expression sym_r_parn\n");
fflush(debug);
#endif DEBUG
                $$ = malloc(strlen($1) + strlen($2) +
                        strlen($3->new_code) + strlen($4) + 1);
                strcpy($$, $1);
                strcat($$, $2);
                strcat($$, $3->new_code);
                strcat($$, $4);
            }
        ;

while_prefix:
          sym_key_while sym_l_parn expression sym_r_parn
            {
#ifdef DEBUG
fprintf(debug,"while_prefix --> sym_l_parn expression sym_r_parn\n");
fflush(debug);
#endif DEBUG
                $$ = malloc(strlen($1) + strlen($2) +
```

```
                                        strlen($3->new_code) + strlen($4) + 1);
                        strcpy($$, $1);
                        strcat($$, $2);
                        strcat($$, $3->new_code);
                        strcat($$, $4);
                }
        ;

constant_expression:
          expression
                %prec sym_comma
                {
#ifdef DEBUG
fprintf(debug,"constant_expression --> expression \n");
fflush(debug);
#endif DEBUG
                  $$ = $1;
                }
        ;

optional_expression:
          expression
                {
#ifdef DEBUG
fprintf(debug,"optional_expression --> expression \n");
fflush(debug);
#endif DEBUG
                  $$ = $1;
                }
        |
                {
#ifdef DEBUG
fprintf(debug,"optional_expression --> <NULL> \n");
fflush(debug);
#endif DEBUG
                        $$->new_code = malloc(1);
                  strcpy($$->new_code, "");
                }
        ;

expression_list:
          expression
                %prec sym_comma
                {
#ifdef DEBUG
fprintf(debug,"expression_list --> expression \n");
fflush(debug);
#endif DEBUG
                  $$ = $1;
                }
        | expression_list sym_comma expression
                {
                  int temp;
#ifdef DEBUG
fprintf(debug,"expression_list --> expression_list sym_comma expression \n");
fflush(debug);
#endif DEBUG
                  $$ = $1;
                  $$->next = $3;

                  $$->changed = $1->changed || $3->changed;

                  arg_list[0] = $1->old_code;
                  arg_list[1] = ",";
                  arg_list[2] = $3->old_code;
```

```
                $$->old_code = Concat (3,arg_list);

                arg_list[0] = $1->new_code;
                arg_list[2] = $3->new_code;

                $$->new_code = Concat (3,arg_list);
              }
        ;

expression:
          en_expression
                {
#ifdef DEBUG
fprintf(debug,"expression --> en_expression \n");
fflush(debug);
#endif DEBUG
                /* display the changes that have been made to the expression,
                   if any */
                if ($1 && $1->changed)
                  printf ("Line %4d - %s  ==>  %s\n",line_counter,$1->old_code,$1->new_code);

                $$ = $1;

                }
        ;

en_expression:
         en_expression sym_asgn
                {
                        push_term("$");
                }
                                en_expression
                {
                ENType          returnType;
                ENType          objType;
                ENString        errMsg;
                ENBoolean       retval;
                ENType          obj_type;
                char            *propName;
                long            callArgFlag;
                char            *type_name;
#ifdef DEBUG
fprintf(debug,"en_expression --> en_expression sym_asgn en_expression\n");
fflush(debug);
#endif DEBUG
                $$ = $1;

                /* right hand side - means there is an encore expression
                                left on the stack which we must typecheck */
                if (!is_empty_term())
                      {
                      propName = pop_term();
                      objType = *(ENType *)(pop_term());
                      callArgFlag = 0;
                      SET_INVOKE_MASK(callArgFlag,2);
                      SET_INVOKE_MASK(callArgFlag,3);
                      retval = BOOL(INVOKE(objType,"TypeCheckGetPropValue",4L,
                              callArgFlag,0L,0L,ENFromString(propName),
                              callerType,&returnType,&errMsg));
                      if (EN_OBJ_EQ(retval,EN_FALSE))
                              {
                              yyerror(ENToString(errMsg));
                              }
                      $4->en_type = returnType;
```

```
            $4->actual_type = (char *)ENCTypeName(returnType);
            }

      pop_term();

      /* left hand side of equals - means that there is an encore
            expression on the left hand side of the assign */
      if (!is_empty_term())
            {
            propName = pop_term();
            objType = *(ENType *)pop_term();
            callArgFlag = 0;
            SET_INVOKE_MASK(callArgFlag,2);
            SET_INVOKE_MASK(callArgFlag,3);
            retval = BOOL(INVOKE(objType,"TypeCheckSetPropValue",4L,
                  callArgFlag,0L,0L,ENFromString(propName),
                  callerType,$4->en_type,&errMsg));
            if (EN_OBJ_EQ(retval,EN_FALSE))
                  {
                  yyerror(ENToString(errMsg));
                  }
            $1->en_type = TYPEBoolean;
            $1->actual_type = "ENBoolean";
            }
      else if (EN_OBJ_NEQ($1->en_type,EN_NO_TYPE))
            { /* this will be an assignment */
            callArgFlag = 0;
            SET_INVOKE_MASK(callArgFlag,1);
            retval = BOOL(INVOKE($1->en_type,"TypeCheckAssign",
                  2L,callArgFlag,0L,0L,$4->en_type,
                  &errMsg));
            if (EN_OBJ_EQ(retval,EN_FALSE))
                  {
                  yyerror(ENToString(errMsg));
                  }
            }

      /* concatenate the old code fragments together */
      arg_list[0] = $1->old_code;
      arg_list[1] = $2;
      arg_list[2] = $4->old_code;
      $$->old_code = Concat (3,arg_list);

      /* check if the first expression has been changed to a
         call to GET_ PROP_VALUE.  If so, change it into a call
         to SET_ PROP_VALUE since it occurs on the left hand
         side of the assignment */
      if ($1->changed &&
          Substring ($1->new_code,"GET_PROP_VALU") == 0) {
        char *temp = strrchr($1->new_code,',');
        temp[0] = '\0';

        $1->new_code[0] = 'S';
/*
        $1->new_code[strlen($1->new_code)-1] = '\0';
*/

        arg_list[0] = $1->new_code;
        arg_list[1] = ",";
        arg_list[2] = $4->new_code;
        arg_list[3] = ",0L)";
        $$->new_code = Concat (4,arg_list);

      } else {
```

```
                    arg_list[0] = $1->new_code;
                    arg_list[1] = $2;
                    arg_list[2] = $4->new_code;
                    $$->new_code = Concat (3,arg_list);
                  }

                  $$->changed = $1->changed || $4->changed;
                }
        | en_expression sym_op_eq
                {
                        push_term("$");
                }
                                        en_expression
                {
                ENType          returnType;
                ENType          objType;
                ENString        errMsg;
                ENBoolean       retval;
                ENType          obj_type;
                char            *propName;
                long            callArgFlag;
                char            *type_name;
#ifdef DEBUG
fprintf(debug,"en_expression --> en_expression sym_op_eq en_expression\n");
fflush(debug);
#endif DEBUG
                  $$ = $1;
                  if (!is_empty_term())
                        {
                        propName = pop_term();
                        objType = *(ENType *)(pop_term());
                        callArgFlag = 0;
                        SET_INVOKE_MASK(callArgFlag,2);
                        SET_INVOKE_MASK(callArgFlag,3);
                        retval = BOOL(INVOKE(objType,"TypeCheckGetPropValue",4L,
                                callArgFlag,0L,0L,ENFromString(propName),
                                callerType,&returnType,&errMsg));
                        if (EN_OBJ_EQ(retval,EN_FALSE))
                                {
                                yyerror(ENToString(errMsg));
                                }
                        $4->en_type = returnType;
                        $4->actual_type = (char *)ENCTypeName(returnType);
                        }
                  /* get rid of $ if it's there */
                  pop_term();

                  if (!is_empty_term())
                        {
                        propName = pop_term();
                        objType = *(ENType *)(pop_term());
                        callArgFlag = 0;
                        SET_INVOKE_MASK(callArgFlag,2);
                        SET_INVOKE_MASK(callArgFlag,3);
                        retval = BOOL(INVOKE(objType,"TypeCheckGetPropValue",4L,
                                callArgFlag,0L,0L,ENFromString(propName),
                                callerType,&returnType,&errMsg));
                        if (EN_OBJ_EQ(retval,EN_FALSE))
                                {
                                yyerror(ENToString(errMsg));
                                }
                        $1->en_type = returnType;
                        $1->actual_type = (char *)ENCTypeName(returnType);
                        }
```

```
                if (Is_Encore_Type ($1->actual_type) ||
                    Is_Encore_Type ($4->actual_type)) {

                      $$->changed = TRUE;

                      arg_list[0] = $1->old_code;
                      arg_list[1] = $2;
                      arg_list[2] = $4->old_code;
                      $$->old_code = Concat (3,arg_list);

                      if (!strcmp($2,"=="))
                        arg_list[0] = "!ENUidCmp(";
                      else
                        arg_list[0] = "ENUidCmp(";
                      arg_list[1] = $1->new_code;
                      arg_list[2] = ",";
                      arg_list[3] = $4->new_code;
                      arg_list[4] = ")";
                      $$->new_code = Concat (5,arg_list);

                      $$->type = "int";

                      }
                      else
                              goto merge_logical_expression;
              }
        | _expression
              {
#ifdef DEBUG
fprintf(debug,"en_expression --> _expression\n");
fflush(debug);
#endif DEBUG
                      $$ = $1;
              }
        | en_term
              {
#ifdef DEBUG
fprintf(debug,"en_expression --> en_term\n");
fflush(debug);
#endif DEBUG
                $$ = $1;
                $$->num_exprs = 1;
              }
        ;

_expression:
          _expression sym_op_rel _expression
              {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_rel _expression\n");
fflush(debug);
#endif DEBUG
                      goto merge_logical_expression;
              }
        | _expression sym_comma _expression
              {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_comma _expression\n");
fflush(debug);
#endif DEBUG
                      $$->num_exprs = $1->num_exprs + 1;
                      goto merge_arithmetic_expression;
              }
        | _expression sym_op_div _expression
              {
```

```
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_div _expression\n");
fflush(debug);
#endif DEBUG
                        goto merge_arithmetic_expression;
                }
        | _expression sym_op_mod _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_mod_expression\n");
fflush(debug);
#endif DEBUG
                        goto merge_arithmetic_expression;
                }
        | _expression sym_op_plus _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_plus _expression\n");
fflush(debug);
#endif DEBUG
                        goto merge_arithmetic_expression;
                }
        | _expression sym_op_minus _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_minus _expression\n");
fflush(debug);
#endif DEBUG
                        goto merge_arithmetic_expression;
                }
        | _expression sym_op_shift _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_shift _expression\n");
fflush(debug);
#endif DEBUG
                        goto merge_arithmetic_expression;
                }
        | _expression sym_op_mult _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_mult _expression\n");
fflush(debug);
#endif DEBUG
                merge_arithmetic_expression:
                  $$ = $1;

                  $$->changed = $1->changed || $3->changed;

                  arg_list[0] = $1->old_code;
                  arg_list[1] = $2;
                  arg_list[2] = $3->old_code;
                  $$->old_code = Concat (3,arg_list);

                  arg_list[0] = $1->new_code;
                  arg_list[2] = $3->new_code;
                  $$->new_code = Concat (3,arg_list);
                }
        | _expression sym_op_bit_and _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_bit_and _expression\n");
fflush(debug);
#endif DEBUG
                        goto merge_logical_expression;
```

```
                }
        | _expression sym_op_bit_or _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_bit_or _expression\n");
fflush(debug);
#endif DEBUG
                        goto merge_logical_expression;
                }
        | _expression sym_op_bit_xor _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_bit_xor _expression\n");
fflush(debug);
#endif DEBUG
                        goto merge_logical_expression;
                }
        | _expression sym_op_and _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_and _expression\n");
fflush(debug);
#endif DEBUG
                        goto merge_logical_expression;
                }
        | _expression sym_op_or _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_or _expression\n");
fflush(debug);
#endif DEBUG
                merge_logical_expression:
                  $$ = $1;

                  $$->changed = $1->changed || $3->changed;

                  arg_list[0] = $1->old_code;
                  arg_list[1] = $2;
                  arg_list[2] = $3->old_code;
                  $$->old_code = Concat (3,arg_list);

                  arg_list[0] = $1->new_code;
                  arg_list[2] = $3->new_code;
                  $$->new_code = Concat (3,arg_list);

                  $$->type = "int";
                }
        | _expression sym_op_asgn _expression
                {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_asgn _expression\n");
fflush(debug);
#endif DEBUG
                  $$ = $1;

                  /* concatenate the old code fragments together */
                  arg_list[0] = $1->old_code;
                  arg_list[1] = $2;
                  arg_list[2] = $3->old_code;
                  $$->old_code = Concat (3,arg_list);

                  /* first check to see if there is an actual type which is
                     different from the 'advertised' type and do the
                     necessary coersion */
                  if ($3->actual_type &&
```

```
                    (arg_list[0] = Coerce_Type ($3->actual_type,$3->type))) {

                  arg_list[1] = "(";
                  arg_list[2] = $3->new_code;
                  arg_list[3] = ")";
                  $3->new_code = Concat (4,arg_list);
                  $3->changed = TRUE;
                  $3->type = $3->actual_type;
                  $3->actual_type = 0;
                }

                /* check to see if a type coersion function needs to be
                   called */
                if ((arg_list[0] = Coerce_Type ($1->type,$3->type))) {

                  arg_list[1] = "(";
                  arg_list[2] = $3->new_code;
                  arg_list[3] = ")";
                  $3->new_code = Concat (4,arg_list);
                  $3->changed = TRUE;
                }

                /* check if the first expression has been changed to a
                   call to GET_ PROP_VALUE.  If so, change it into a call
                   to SET_ PROP_VALUE since it occurs on the left hand
                   side of the assignment */
                if ($1->changed &&
                    Substring ($1->new_code,"GET_PROP_VALU") == 0) {
                  char *temp = strrchr($1->new_code,',');
                  temp[0] = '\0';

                  $1->new_code[0] = 'S';
              /*
                  $1->new_code[strlen($1->new_code)-1] = '\0';
              */

                  arg_list[0] = $1->new_code;
                  arg_list[1] = ",";
                  arg_list[2] = $3->new_code;
                  arg_list[3] = ",0L)";
                  $$->new_code = Concat (4,arg_list);

                } else {

                  arg_list[0] = $1->new_code;
                  arg_list[1] = $2;
                  arg_list[2] = $3->new_code;
                  $$->new_code = Concat (3,arg_list);
                }

                $$->changed = $1->changed || $3->changed;
              }
        | _expression sym_question _expression sym_colon _expression
              {
#ifdef DEBUG
fprintf(debug,"_expression --> _expression sym_op_colon _expression\n");
fflush(debug);
#endif DEBUG
                $$ = $1;

                $$->changed = $1->changed || $3->changed || $5->changed;

                arg_list[0] = $1->old_code;
                arg_list[1] = $2;
                arg_list[2] = $3->old_code;
```

```
                arg_list[3] = $4;
                arg_list[4] = $5->old_code;
                $$->old_code = Concat (5,arg_list);

                arg_list[0] = $1->new_code;
                arg_list[2] = $3->new_code;
                arg_list[4] = $5->new_code;
                $$->new_code = Concat (5,arg_list);
              }
      | sym_query_start query_prog sym_query_end
              {
                char *start_call, *end_call;

              /* Here we've just processed an embedded query */
              /* (sym_query_{start,end} are delimiters). */
              /* The rule "query_prog" marks the beginning */
              /* of the query processing portion of this */
              /* grammar */

#ifdef DEBUG
fprintf(debug,"_expression --> sym_query_start query_prog sym_query_end\n");
fflush(debug);
#endif DEBUG
              /* REPLACE - this will change later */
              /*
                Write_Out_Query ($1);
                Write_Out_Symbol_Table ();

                Call_Query_Parser ();
              */

                $$ = Allocate_Code_Block ();
                $$->changed = TRUE;

              /* Is this necessar? */

                $$->old_code = "<old code>";

              /* The original embedded query is replaced by */
              /* a call to the function which has been generated */
              /* and which invokes the query method. This function */
              /* will be appended to the source immediately after */
              /* the function currently being preprocessed. */

                start_call = (char *)index($2, '\n');
                start_call++;
                $$->new_code = strdup (start_call);
                end_call = (char *)index($$->new_code, '\n');
                end_call[0] = '\0';

              /* Since we're done with they query, we can */
              /* clear off the lambda-variable and collection stacks */

                StackFrame = (param_descr *) NULL;
                Coll_List = (coll_descr *) NULL;

              /* This needs to be changed, but to what? */

                $$->type = "EN_TYPE_XXX";
              }
      | term
              {
                      $$ = $1;
                      $$->num_exprs = 1;
              }
```

```
        ;

en_term:
         en_term sym_pound sym_identifier
                {
                ENType          returnType;
                ENString        errMsg;
                ENType          objType;
                ENBoolean       retval;
                char            *propName;
                long            callArgFlag;
#ifdef DEBUG
fprintf(debug," en_term --> en_term sym_pound sym_identifier\n");
fflush(debug);
#endif DEBUG
                 $$ = $1;
                 if (!is_empty_term())
                       {
                       propName = pop_term();
                       objType = *(ENType *)pop_term();
                       SET_INVOKE_MASK(callArgFlag,2);
                       SET_INVOKE_MASK(callArgFlag,3);
                       retval = BOOL(INVOKE(objType,"TypeCheckGetPropValue",4L,
                               callArgFlag,0L,0L,ENFromString(propName),
                               callerType,&returnType,&errMsg));
                       if (EN_OBJ_EQ(retval,EN_FALSE))
                               {
                               yyerror(ENToString(errMsg));
                               }
                       $1->en_type = returnType;
                       }

                 SET_INVOKE_MASK(callArgFlag,2);
                 SET_INVOKE_MASK(callArgFlag,3);
                 retval = BOOL(INVOKE($1->en_type,"TypeCheckGetProperty",
                               4L,callArgFlag,0L,0L,ENFromString($3),
                               callerType,&returnType,&errMsg));
                 if (!ENUidCmp(retval,EN_FALSE))
                       {
                       yyerror(ENToString(errMsg));
                       }
                 $1->en_type = returnType;
                 $1->actual_type = (char *)ENCTypeName($1->en_type);

                 $$->changed = TRUE;

                 arg_list [0] = $1->old_code;
                 arg_list [1] = $2;
                 arg_list [2] = $3;
                 $$->old_code = Concat (3,arg_list);

                 arg_list [0] = "GET_PROPERTY(";
                 arg_list [1] = $1->new_code;
                 arg_list [2] = ",\"";
                 arg_list [3] = $3;
                 arg_list [4] = "\",0L)";
                 $$->new_code = Concat (5,arg_list);

                 $$->type = "ENObject";
                }
       | en_term sym_at sym_identifier
                {
                ENType          returnType;
                ENString        errMsg;
                ENType          objType;
```

```
                ENBoolean       retval;
                char            *propName;
                long            callArgFlag;
#ifdef DEBUG
fprintf(debug," en_term --> en_term sym_at sym_identifier\n");
fflush(debug);
#endif DEBUG
                  if (!is_empty_term())
                        {
                        propName = pop_term();
                        objType = *(ENType *)pop_term();
                        SET_INVOKE_MASK(callArgFlag,2);
                        SET_INVOKE_MASK(callArgFlag,3);
                        retval = BOOL(INVOKE(objType,"TypeCheckGetPropValue",4L,
                                callArgFlag,0L,0L,ENFromString(propName),
                                callerType,&returnType,&errMsg));
                        if (EN_OBJ_EQ(retval,EN_FALSE))
                                {
                                yyerror(ENToString(errMsg));
                                }
                        $1->en_type = returnType;
                        }

                  push_term(&($1->en_type));
                  push_term($3);

                  $$ = $1;

                  $$->changed = TRUE;

                  arg_list [0] = $1->old_code;
                  arg_list [1] = $2;
                  arg_list [2] = $3;
                  $$->old_code = Concat (3,arg_list);

                  arg_list [0] = "GET_PROP_";
                  arg_list [1] = "VALUE(";
                  arg_list [2] = $1->new_code;
                  arg_list [3] = ",\"";
                  arg_list [4] = $3;
                  arg_list [5] = "\",0L)";
                  $$->new_code = Concat (6,arg_list);

                  $$->actual_type = 0;
                  $$->type = "ENObject";
                }
        | en_term sym_at sym_tick sym_identifier sym_tick
                {
                  $$ = $1;

                  $$->changed = TRUE;

                  arg_list [0] = $1->old_code;
                  arg_list [1] = $2;
                  arg_list [2] = $3;
                  arg_list [3] = $4;
                  arg_list [4] = $5;
                  $$->old_code = Concat (5,arg_list);

                  arg_list [0] = "GET_PROP_";
                  arg_list [1] = "VALUE(";
                  arg_list [2] = $1->new_code;
                  arg_list [3] = ",";
                  arg_list [4] = $4;
                  arg_list [5] = ",0L)";
```

```
                $$->new_code = Concat (6,arg_list);

                /* REPLACE - the lookup of the property type here will have
                   to be done at runtime */
                $$->actual_type = 0;
                $$->type = "ENObject";
              }
        | en_identifier
              {
              char *type_name;
              bool    isPtr;

#ifdef DEBUG
fprintf(debug," en_term --> en_identifier\n");
fflush(debug);
#endif DEBUG
                $$ = Allocate_Code_Block ();
                $$->type = Lookup_Type ($1);
                type_name = GetBaseType($$->type,&isPtr);
                $$->en_type = GetTypeObject(type_name);
                $$->old_code = $1;
                $$->new_code = $1;
              }
        | en_term sym_at sym_identifier sym_l_parn sym_r_parn
              {
              ENUIDBytes      argList;
              ENType          evalType;
              ENString        errMsg;
              ENBoolean       retval;
              ENType          objType;
              CODE            *argPtr;
              long            argFlag;
              long            callArgFlag;
              char            *propName;
#ifdef DEBUG
fprintf(debug," en_term --> en_term sym_at identifier sym_l_parn sym_r_parn \n");
fflush(debug);
#endif DEBUG
                $$ = $1;

                if (!is_empty_term())
                      {
                      propName = pop_term();
                      objType = *(ENType *)pop_term();
                      callArgFlag = 0;
                      SET_INVOKE_MASK(callArgFlag,2);
                      SET_INVOKE_MASK(callArgFlag,3);
                      retval = BOOL(INVOKE(objType,"TypeCheckGetPropValue",4L,
                              callArgFlag,0L,0L,ENFromString(propName),
                              callerType,&evalType,&errMsg));
                      if (!ENUidCmp(retval,EN_FALSE))
                              {
                              yyerror(ENToString(errMsg));
                              }
                      $1->en_type = evalType;
                      }

                callArgFlag = 0;
                SET_INVOKE_MASK(callArgFlag,4);
                SET_INVOKE_MASK(callArgFlag,5);
                retval = BOOL(INVOKE($1->en_type,"TypeCheckInvoke",6L,
                              callArgFlag,0L,0L,ENFromString($3),
                              callerType,EN_NO_UIDBYTES,EN_ZERO,
                              &evalType,&errMsg));
```

```
                if (EN_OBJ_EQ(retval,EN_FALSE))
                      {
                      yyerror(ENToString(errMsg));
                      evalType = TYPEObject;
                      }

                $$->en_type = evalType;
                $$->actual_type = (char *)ENCTypeName(evalType);
                $$->type = "ENObject";

                $$->changed = TRUE;

                arg_list [0] = $1->old_code;
                arg_list [1] = $2;
                arg_list [2] = $3;
                arg_list [3] = $4;
                arg_list [4] = $5;
                $$->old_code = Concat (6,arg_list);

                arg_list[0] = "INVOKE(";
                arg_list [1] = $1->new_code;
                arg_list [2] = ",\"";
                arg_list [3] = $3;
                arg_list [4] = "\" ";
                arg_list [5] = ",0L,0L,0L,0L)";
                $$->new_code = Concat (6,arg_list);
              }
      | en_term sym_at sym_identifier sym_l_parn expression_list sym_r_parn
              {
              ENType          *argList;
              ENType          evalType;
              ENString        errMsg;
              ENBoolean       retval;
              ENType          objType;
              ENUIDBytes      argObj;
              char            *propName;
              CODE            *argPtr;
              long            argFlag;
              long            callArgFlag;
              bool            isPtr;
              short           i;
              short           len;
#ifdef DEBUG
fprintf(debug," en_term --> en_term sym_at sym_identifier expression_list sym_r_parn\n");
fflush(debug);
#endif DEBUG
                $$ = $1;

                if (!is_empty_term())
                      {
                      propName = pop_term();
                      objType = *(ENType *)pop_term();
                      callArgFlag = 0;
                      SET_INVOKE_MASK(callArgFlag,2);
                      SET_INVOKE_MASK(callArgFlag,3);
                      retval = BOOL(INVOKE(objType,"TypeCheckGetPropValue",4L,
                              callArgFlag,0L,0L,ENFromString(propName),
                              callerType,&evalType,&errMsg));
                      if (!ENUidCmp(retval,EN_FALSE))
                              {
                              yyerror(ENToString(errMsg));
                              }
                      $1->en_type = evalType;
                      }
```

```
                /* make the argument list for typechecking */
                argList = GETMEM($5->num_exprs,ENType);
                argFlag = 0;
                for (i=0,argPtr = $5;argPtr;i++,argPtr = argPtr->next)
                      {
                      argPtr->en_type =
                            GetTypeObject(GetBaseType(argPtr->type,&isPtr));
                      argList[i] = argPtr->en_type;
                      len = strlen(argPtr->type);
                      if (isPtr == TRUE)
                            SET_INVOKE_MASK(argFlag,i);
                      }
                argObj = ENFromUIDBytes(argList,$5->num_exprs);

                callArgFlag = 0;
                SET_INVOKE_MASK(callArgFlag,4);
                SET_INVOKE_MASK(callArgFlag,5);
                retval = BOOL(INVOKE($1->en_type,"TypeCheckInvoke",6L,
                            callArgFlag,0L,0L,ENFromString($3),
                            callerType,argObj,ENFromLong(argFlag),
                            &evalType,&errMsg));

                if (EN_OBJ_EQ(retval,EN_FALSE))
                      {
                      yyerror(ENToString(errMsg));
                      evalType = TYPEObject;
                      }
                $$->en_type = evalType;
                $$->actual_type = (char *)ENCTypeName(evalType);
                $$->type = "ENObject";

                $$->changed = TRUE;

                arg_list [0] = $1->old_code;
                arg_list [1] = $2;
                arg_list [2] = $3;
                arg_list [3] = $4;
                arg_list[4] = $5->old_code;
                arg_list[5] = $6;
                $$->old_code = Concat (6,arg_list);

                arg_list[0] = "INVOKE(";
                arg_list [1] = $1->new_code;
                arg_list [2] = ",\"";
                arg_list [3] = $3;
                arg_list [4] = "\" ";
                arg_list[5] = ",";
                arg_list[6] = itoa($5->num_exprs);
                arg_list[7] = "L,";
                arg_list[8] = itoa(argFlag);
                arg_list[9] = "L,0L,0L,";
                arg_list[10] = $5->new_code;
                arg_list[11] = $6;
                $$->new_code = Concat (12,arg_list);
              }
        | sym_l_parn type_name sym_r_parn en_term
              %prec sym_op_inc
              {
#ifdef DEBUG
fprintf(debug," en_term --> sym_l_parn type_name sym_r_parn en_term\n");
fflush(debug);
#endif DEBUG
                $$ = $4;

                /* concatenate the old code fragments together */
```

```
                arg_list[0] = $1;
                arg_list[1] = $2;
                arg_list[2] = $3;
                arg_list[3] = $4->old_code;
                $$->old_code = Concat (4,arg_list);

                /* Now coerce the type of term to type type specified by
                   type_name, if possible */
                if (arg_list[0] = Coerce_Type ($2,$4->type))
                  {
                  arg_list[1] = "(";
                  arg_list[2] = $4->new_code;
                  arg_list[3] = ")";
                  $$->new_code = Concat (4,arg_list);
                  $$->changed = TRUE;
                  }
                else
                 {
                 arg_list[0] = $1;
                 arg_list[1] = $2;
                 arg_list[2] = $3;
                 arg_list[3] = $4->new_code;
                 $$->new_code = Concat (4,arg_list);
                 }

                $$->type = $2;
              }
        ;

term:
          term sym_op_inc
                {
                  $$ = $1;

                  arg_list[0] = $1->old_code;
                  arg_list[1] = $2;
                  $$->old_code = Concat (2,arg_list);

                  arg_list[0] = $1->new_code;
                  $$->new_code = Concat (2,arg_list);
                }
        | sym_op_mult term
                { goto op_term_merge; }
        | sym_op_mult en_term
                { goto op_term_merge; }
        | sym_op_bit_and term
                { goto op_term_merge; }
        | sym_op_bit_and en_term
                { goto op_term_merge; }
        | sym_op_minus term
                { goto op_term_merge; }
        | sym_op_unary term
                { goto op_term_merge; }
        | sym_op_inc term
                {
                op_term_merge:
                  $$ = $2;

                  arg_list[0] = $1;
                  arg_list[1] = $2->old_code;
                  $$->old_code = Concat (2,arg_list);

                  arg_list[1] = $2->new_code;
                  $$->new_code = Concat (2,arg_list);
                }
```

```
        | sym_key_sizeof term
                %prec sym_key_sizeof
                { $$ = $2;

                  arg_list[0] = $1;
                  arg_list[1] = $2->old_code;
                  $$->old_code = Concat (2,arg_list);

                  arg_list[1] = $2->new_code;
                  $$->new_code = Concat (2,arg_list);

                  $$->type = "int";
                }
        | sym_l_parn type_name sym_r_parn term
                %prec sym_op_inc
                {
#ifdef DEBUG
fprintf(debug," term --> sym_l_parn type_name sym_r_parn term\n");
fflush(debug);
#endif DEBUG
                  $$ = $4;

                  /* concatenate the old code fragments together */
                  arg_list[0] = $1;
                  arg_list[1] = $2;
                  arg_list[2] = $3;
                  arg_list[3] = $4->old_code;
                  $$->old_code = Concat (4,arg_list);

                  /* first check to see if there is an actual type which is
                     different from the 'advertised' type and do the
                     necessary coersion */
                  if ($4->actual_type &&
                      (arg_list[0] = Coerce_Type ($4->actual_type,$4->type))) {

                    arg_list[1] = "(";
                    arg_list[2] = $4->new_code;
                    arg_list[3] = ")";
                    $4->new_code = Concat (4,arg_list);
                    $4->changed = TRUE;
                    $4->type = $4->actual_type;
                    $4->actual_type = 0;
                  }

                  /* Now coerce the type of term to type type specified by
                     type_name, if possible */
                  if ((arg_list[0] = Coerce_Type ($2,$4->type))) {

                    arg_list[1] = "(";
                    arg_list[2] = $4->new_code;
                    arg_list[3] = ")";
                    $$->new_code = Concat (4,arg_list);
                    $$->changed = TRUE;
                  } else {

                    arg_list[0] = $1;
                    arg_list[1] = $2;
                    arg_list[2] = $3;
                    arg_list[3] = $4->new_code;
                    $$->new_code = Concat (4,arg_list);
                  }

                  $$->type = $2;
                }
        | sym_key_sizeof sym_l_parn type_name sym_r_parn
```

```
                %prec sym_key_sizeof
                {
                  $$ = Allocate_Code_Block ();

                  arg_list[0] = $1;
                  arg_list[1] = $2;
                  arg_list[2] = $3;
                  arg_list[3] = $4;
                  $$->old_code = Concat (4,arg_list);
                  $$->new_code = Concat (4,arg_list);

                  $$->type = "int";
                }
        | term sym_l_sbracket _expression sym_r_sbracket
                {
#ifdef DEBUG
fprintf(debug,"\nterm --> sym_l_sbracket _expression sym_r_sbracket:\n");
fflush(debug);
#endif DEBUG
                  $$ = $1;

                  $$->changed = $1->changed || $3->changed;

                  arg_list[0] = $1->old_code;
                  arg_list[1] = $2;
                  arg_list[2] = $3->old_code;
                  arg_list[3] = $4;
                  $$->old_code = Concat (4,arg_list);

                  arg_list[0] = $1->new_code;
                  arg_list[2] = $3->new_code;
                  $$->new_code = Concat (4,arg_list);
                }
        | function_prefix sym_r_parn
                {
                  $$ = $1;

                  /* REPLACE - have to get the type from ENCORE */

                  arg_list[0] = $1->old_code;
                  arg_list[1] = $2;
                  $$->old_code = Concat (2,arg_list);

                  if ($1->changed) {

                /*
                    if ($1->use_l_invoke)
                    {
                        arg_list[0] = "L_INV";
                        arg_list[1] = "OKE(";
                    }
                    else
                */
                    {
                        arg_list[0] = "INV";
                        arg_list[1] = "OKE(";
                    }
                    arg_list[2] = $1->new_code;
                    arg_list[3] = ",0L,0L,0L,0L";
                    arg_list[4] = $2;
                    $$->new_code = Concat (5,arg_list);

                  } else {

                    arg_list[0] = $1->new_code;
```

```
                $$->new_code = Concat (2,arg_list);
              }
            }
        | function_prefix expression_list sym_r_parn
            {
              $$ = $1;

              /* REPLACE - have to get the type from ENCORE */

              arg_list[0] = $1->old_code;
              arg_list[1] = $2->old_code;
              arg_list[2] = $3;
              $$->old_code = Concat (3,arg_list);

              if ($1->changed) {

            /*
                if ($1->use_l_invoke)
                {
                    arg_list[0] = "L_INV";
                    arg_list[1] = "OKE(";
                }
                else
            */
                {
                    arg_list[0] = "INV";
                    arg_list[1] = "OKE(";
                }
                $1->new_code[strlen($1->new_code)-1] = ',';
                arg_list[2] = $1->new_code;
                arg_list[3] = itoa($2->num_exprs);
                arg_list[4] = "L,0L,0L,0L,";
                arg_list[5] = $2->new_code;
                arg_list[6] = $3;
                $$->new_code = Concat (7,arg_list);

              } else {

                arg_list[0] = $1->new_code;
                arg_list[1] = $2->new_code;
                $$->new_code = Concat (3,arg_list);
              }
            }
        | term sym_period sym_identifier
            {
              $$ = $1;

              /* REPLACE - have to get the type from the record definition
                 (or can just treat it as a structure reference?) */

              arg_list [0] = $1->old_code;
              arg_list [1] = $2;
              arg_list [2] = $3;
              $$->old_code = Concat (3,arg_list);

              arg_list [0] = $1->new_code;
              $$->new_code = Concat (3,arg_list);

              if (Is_Encore_Type ($1->type))
                $$->type = "ENProperty";
            }
        | term sym_arrow sym_identifier
            {
              $$ = $1;
```

```
                /* REPLACE - have to get the type from the record definition
                   (or can just treat it as a structure reference?) */

                arg_list [1] = $2;
                arg_list [2] = $3;
                $$->old_code = Concat (3,arg_list);

                arg_list [0] = $1->new_code;
                $$->new_code = Concat (3,arg_list);
              }
        | sym_identifier
              {
                $$ = Allocate_Code_Block ();

                $$->type = Lookup_Type ($1);

                $$->old_code = $1;
                $$->new_code = $1;
              }
        | sym_constant
              {
                $$ = Allocate_Code_Block ();

                /* REPLACE - this should include float,int,char */

                $$->type = "long";
                $$->old_code = $1;
                $$->new_code = $1;
              }
        | sym_string
              {
                $$ = Allocate_Code_Block ();

                $$->type = "char *";
                $$->old_code = $1;
                $$->new_code = strdup ($1);
              }
        | sym_l_parn _expression sym_r_parn
              {
                $$ = $2;

                arg_list[0] = $1;
                arg_list[1] = $2->old_code;
                arg_list[2] = $3;
                $$->old_code = Concat (3,arg_list);

                arg_list[1] = $2->new_code;
                $$->new_code = Concat (3,arg_list);
              }
        ;


type_name:
        type_specifier abstract_declarator
              {
                arg_list[0] = $1;
                arg_list[1] = $2;

                $$ = Concat_With_Spaces (2,arg_list);
              }
        ;

abstract_declarator:
        /* empty */
              { $$ = 0; }
```

```
        | sym_l_parn sym_r_parn
                {
                  arg_list[0] = $1;
                  arg_list[1] = $2;

                  $$ = Concat_With_Spaces (2,arg_list);
                }
        | sym_l_parn abstract_declarator sym_r_parn sym_l_parn sym_r_parn
                {
                  arg_list[0] = $1;
                  arg_list[1] = $2;
                  arg_list[2] = $3;
                  arg_list[3] = $4;
                  arg_list[4] = $5;

                  $$ = Concat_With_Spaces (5,arg_list);
                }
        | sym_op_mult abstract_declarator
                {
                  arg_list[0] = $1;
                  arg_list[1] = $2;

                  $$ = Concat_With_Spaces (2,arg_list);
                }
        | abstract_declarator sym_l_sbracket sym_r_sbracket
                {
#ifdef DEBUG
fprintf(debug,"abstract_declarator --> abstract_declarator sym_l_bracket sym_r_sbracket\n");
fflush(debug);
#endif DEBUG
                  arg_list[0] = $1;
                  arg_list[1] = $2;
                  arg_list[2] = $3;

                  $$ = Concat_With_Spaces (3,arg_list);
                }
        | abstract_declarator sym_l_sbracket constant_expression sym_r_sbracket
                {
#ifdef DEBUG
fprintf(debug,"abstract_declarator --> abstract_declarator sym_l_bracket constant_expresion sym_r_sbracket\n");
fflush(debug);
#endif DEBUG
                  arg_list[0] = $1;
                  arg_list[1] = $2;
                  arg_list[2] = $3->new_code;
                  arg_list[3] = $4;

                  $$ = Concat_With_Spaces (4,arg_list);
                }

        | sym_l_parn abstract_declarator sym_r_parn
                {
                  arg_list[0] = $1;
                  arg_list[1] = $2;
                  arg_list[2] = $3;

                  $$ = Concat_With_Spaces (3,arg_list);
                }
        ;


function_prefix:
        sym_identifier sym_l_parn
                {
                  $$ = Allocate_Code_Block ();
```

```
                $$->type = Lookup_Type ($1);

                arg_list[0] = $1;
                arg_list[1] = $2;
                $$->old_code = Concat (2,arg_list);
                $$->new_code = Concat (2,arg_list);
              }
        | term sym_l_parn
              {

                $$ = $1;

                arg_list[0] = $1->old_code;
                arg_list[1] = $2;
                $$->old_code = Concat (2,arg_list);

                arg_list[0] = $1->new_code;
                $$->new_code = Concat (2,arg_list);
              }
        ;

/************ This rule marks the beginning of the QUERY PROCESSOR ******************/

query_prog:     { init_query(); } query
                      {
                        FILE *fs;
                        ENType enType;
                        short         pos;
                        int random = rand();

#ifdef DEBUG
fprintf(debug,"prog -> query\n");
fflush(debug);
#endif DEBUG
                      /* Start to build the function which actually */
                      /* invokes the user query */
                      /* "fwd_decl" will be placed immediately before */
                      /* the original function containing the query, */
                      /* while the generated function will be placed */
                      /* immediately following */

                        pos = 0;
                        if (!runtime_check)
                        {
                              /* do the forward declaration */
                              arg_list[0] = "ENObject";
                              arg_list[1] = " Query";
                              arg_list[2] = strdup(itoa(random));
                              arg_list[3] = "();\n";
                              fwd_decl = Concat(4,arg_list);
                              arg_list[pos++] = "ENObject";   /****TEMP. HACK****/
                        }
                        else
                        {
                              /* do the forward declaration */
                              arg_list[0] = "ENObject";
                              arg_list[1] = " Query";
                              arg_list[2] = strdup(itoa(random));
                              arg_list[3] = "();\n";
                              fwd_decl = Concat(4,arg_list);
                              arg_list[pos++] = "ENObject";
                        }

                      /* Generate the name of the query-function */
```

```
                        arg_list[pos++] = "\nQuery";
                        arg_list[pos++] = strdup(itoa(random));
#ifdef DEBUG
fprintf(debug,"query_prog rule: itoa(random) = %s\n", arg_list[pos - 1]);
#endif DEBUG
                        /* Generate its argument list */

                        arg_list[pos++] = "(";
                        write_coll_list(&pos,arg_list);
                        arg_list[pos++] = ")";
                        arg_list[pos++] = "\n";


                        /* write_coll_decls() writes out definitions */
                        /* for the arguments */

                        write_coll_decls(&pos,arg_list);
                        arg_list[pos++] = "\n{\n";

                        /* Generate declarations for local variables */
                        arg_list[pos++] = "     ENObject args;";
                        arg_list[pos++] = "\n";
                        arg_list[pos++] = declarations;

                        /* Generate initializations for local variables */

                        arg_list[pos++] = initializations;
                        arg_list[pos++] = "\n";

                        /* make_BuildArg() writes out call to ENBuildArgList */
                        /* putting all local variables into a list */

                        make_BuildArg(&pos,arg_list);
                        arg_list[pos++] = "\n";
                        arg_list[pos++] = "     ";
                        arg_list[pos++] = "return(";
                        arg_list[pos++] = $2->text;
                        arg_list[pos++] = ")";
                        arg_list[pos++] = ";\n}\n";
                        $$ = Concat(pos,arg_list);

#ifdef DEBUG
fprintf(debug,"Call Generated:\n%s\n", $$);
fflush(debug);
#endif DEBUG

                        /* "query_functions" will be appended to the end */
                        /* of the source in which the query was originally */
                        /* embedded */
                        arg_list[0] = query_functions;
                        arg_list[1] = $$;
                        query_functions = Concat(2,arg_list);

                        }
        ;

query   :       SELECT sym_l_parn obj sym_comma lambdaexp
                        {
                          ENType enType;
                        /*
                          if (!runtime_check)
                          {
                                enType = TYPE(GET_PROP_VALUE($3->type, "type", 0L));

#ifdef DEBUG
```

```
fprintf(debug,"name of type of %s: %s\n", $3->text, ENToString(STRING(GET_PROP_VALUE(enType, "name", 0L))));
#endif DEBUG
                                if (EN_OBJ_NEQ(enType, TYPEColType))
                                {
                                        fprintf(debug,"ERROR  -- %s must be a collection\n", $3->text);
                                        fflush(debug);
                                        exit(1);
                                }
                                assign_type(GET_PROP_VALUE($3->type, "memType", 0L));
                          }
                          else
                        */
                          {
                                assign_type(TYPEObject);
                          }
                        }
                pred sym_r_parn
                        {
#ifdef DEBUG
fprintf(debug,"query -> SELECT ( obj , lambdaexp pred)\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                        /* type-check INVOKE here */

                          arg_list[0] = "INVOKE(";
                          arg_list[1] = $3->text;
                          arg_list[2] = ",";
                          arg_list[3] = "\"Select\"";
                          arg_list[4] = ",";
                          arg_list[5] = "2L,0L,0L,0L";
                          arg_list[6] = ",";
        "
              alling gen_pred_code\n");


                              red_code() generates a function (of type ENBoolean) */
                                   ns EN_TRUE or EN_FALSE based on the value */
                                        pred" */
                                             code($7);
                                                 new operation object */



                                                st);

                                                 this query */

                                                   */









                                                 VALUE($3->type, "type", 0L));
                                                 TYPEColType))
```

```
                                fprintf(debug,"ERROR  -- %s must be a collection\n", $3->text);
                                fflush(debug);
                                exit(1);
                        }
                        assign_type(GET_PROP_VALUE($3->type, "memType", 0L));
                      }
                      else
                     */
                      {
                        assign_type(TYPEObject);
                      }
                    }
              image_func sym_r_parn
                    {
#ifdef DEBUG
fprintf(debug,"query -> IMAGE ( obj , lambdaexp func )\n");
fflush(debug);
#endif DEBUG
                      $$ = Alloc_Var();

                      arg_list[0] = "INVOKE(";
                      arg_list[1] = $3->text;
                      arg_list[2] = ",";
                      arg_list[3] = "\"Image\"";
                      arg_list[4] = ",";
                      arg_list[5] = "2L,0L,0L,0L";
                      arg_list[6] = ",";

                      /* gen_FuncOp_code() generates a function which
                            returns object resulting when the code is
                            "image_func" is executed */

                      arg_list[7] = gen_FuncOp_code($7->text);

                      /* may also have to return operation object */
                      arg_list[8] = ",";
                      arg_list[9] = "args";
                      arg_list[10] = ")";

                      $$->text = Concat(11,arg_list);

                     /* discard variables local to this query */
                      pop_params();
                     /* Need to assign $$->type here */
#ifdef DEBUG
fprintf(debug, "Call Generated: %s\n", $$->text);
fflush(debug);
#endif DEBUG
                    }
        |       OJOIN sym_l_parn obj sym_comma obj sym_comma attrname sym_comma attrname sym_comma lambdaexp
                    {
                      ENType type1, type2;
                     /*
                      if (!runtime_check)
                      {
                            type1 = TYPE(GET_PROP_VALUE($3->type, "type", 0L));
                            type2 = TYPE(GET_PROP_VALUE($5->type, "type", 0L));
                            if ( (EN_OBJ_NEQ(type1, TYPEColType)) ||
                                 (EN_OBJ_NEQ(type2, TYPEColType)) )
                            {
                                fprintf(debug,"ERROR  -- %s and %s must be collections\n", $3->text, $5->text);
                                fflush(debug);
                                exit(1);
                            }
                            type1 = TYPE(GET_PROP_VALUE($3->type, "memType", 0L));
```

```
                        type2 = TYPE(GET_PROP_VALUE($5->type, "memType", 0L));
                        ojoin_assign_type(type1, type2);
                      }
                      else
                     */
                      {
                        ojoin_assign_type(TYPEObject, TYPEObject);
                      }
                    }
              pred sym_r_parn
                    {
#ifdef DEBUG
fprintf(debug,"query -> OJOIN(obj, obj, attrname, attrname, lambdaexp pred)\n");
fflush(debug);
#endif DEBUG
                      $$ = Alloc_Var();

                      arg_list[0]="INVOKE(";
                      arg_list[1]= $3->text;
                      arg_list[2] = ",";
                      arg_list[3] = $5->text;
                      arg_list[4] = ",";
                      arg_list[5] = "\"Ojoin\"";
                      arg_list[6] = ",";
                      arg_list[7] = "4L,0L,0L,0L";
                      arg_list[8] = ",";
                      arg_list[9] = $7;
                      arg_list[10] = ",";
                      arg_list[11] = $9;
                      arg_list[12] = ",";

                     /* gen_pred_code() generates a function (of type ENBoolean) */
                     /* which returns EN_TRUE or EN_FALSE based on the value */
                     /* of the test in "pred" */

                      arg_list[13] =gen_pred_code($13);

                     /* may also have to return operation object */

                      arg_list[14] =",";
                      arg_list[15] ="args";
                      arg_list[16] =")";

                      $$->text = Concat(17,arg_list);

                     /* discard variables local to this query */
                      pop_params();
                     /* Need to assign $$->type here */
#ifdef DEBUG
fprintf(debug,"Call Generated: %s\n", $$->text);
fflush(debug);
#endif DEBUG
                    }
        |       PROJECT sym_l_parn obj sym_comma lambdaexp
                    {
                      ENType enType;
                     /*
                      if (!runtime_check)
                      {
                            enType = TYPE(GET_PROP_VALUE($3->type, "type", 0L));
                            if (EN_OBJ_NEQ(enType, TYPEColType))
                            {
                                    fprintf(debug,"ERROR  -- %s must be a collection\n", $3->text);
                                    fflush(debug);
                                    exit(1);
```

```
                        }
                        assign_type(GET_PROP_VALUE($3->type, "memType", 0L));
                      }
                      else
                    */
                      {
                        assign_type(TYPEObject);
                      }
                    }
            tuple sym_r_parn
                    {
                      param_descr *param_list;
                      char *type_name;
                      short         pos;

                      param_list = StackFrame->prev;
#ifdef DEBUG
fprintf(debug,"query -> PROJECT ( obj , lambdaexp tuple)\n");
fflush(debug);
#endif DEBUG
                      $$ = Alloc_Var();
                      $$->text = malloc(strlen("INV") + strlen("OKE(") +
                            strlen($3->text) +
                            strlen(",Project,TupleArgs,args)") + 1);

                    /* Generate declarations of local variables */
                      pos = 0;
                      arg_list[pos++] = declarations;
                      if (param_list)
                      do {
                            arg_list[pos++] =  "     ENObject ";
                            arg_list[pos++] = param_list->name;
                            arg_list[pos++] = ";\n";
                            param_list = param_list->prev;
                      } while (param_list);
                      arg_list[pos++] = "\n";

                    /* "TupleArgs" contains the functions used by Project */
                    /* to create the specified tuple-attributes */
                      arg_list[pos++] = "     ENObject TupleArgs;\n";
                      declarations = Concat(pos,arg_list);

                      arg_list[0] = initializations;
                      arg_list[1] = "     TupleArgs = ";
                      arg_list[2] = $7;
                      arg_list[3] = ";\n";
                      initializations = Concat(4,arg_list);

                      arg_list[0] = "INVOKE(";
                      arg_list[1] = $3->text;
                      arg_list[2] = ",";
                      arg_list[3] = "\"Project\"";
                      arg_list[4] = ",";
                      arg_list[5] =  "2L,0L,0L,0L";
                      arg_list[6] = ",";
                      arg_list[7] = "TupleArgs";
                      arg_list[8] = ",";
                      arg_list[8] = "args";
                      arg_list[10] = ")";

                      $$->text = Concat(11,arg_list);

                    /* discard variables local to this query */
                      pop_params();
                    /* Need to assign $$->type here */
```

```
#ifdef DEBUG
fprintf(debug,"Call Generated: %s\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       FLATTEN sym_l_parn obj  sym_r_parn
                        {
#ifdef DEBUG
fprintf(debug,"query -> FLATTEN ( obj )\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          arg_list[0] = "INVOKE(";
                          arg_list[1] = $3->text;
                          arg_list[2] = ",";
                          arg_list[3] = "\"Flatten\"";
                          arg_list[4] = ",";
                          arg_list[5] = "0L,0L,0L,0L";
                          arg_list[6] = ")";
                          $$->text = Concat(7,arg_list);

#ifdef DEBUG
fprintf(debug,"Call Generated: %s\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       NEST sym_l_parn obj sym_comma attrname sym_r_parn
                        {
#ifdef DEBUG
fprintf(debug,"query -> NEST ( obj , attrname )\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          arg_list[0] = "INV";
                          arg_list[1] = "OKE(";
                          arg_list[2] = $3->text;
                          arg_list[3] = ",";
                          arg_list[4] = "\"Nest\"";
                          arg_list[5] = ",";
                          arg_list[6] = "1L,0L,0L,0L";
                          arg_list[7] = ",";
                          arg_list[8] = $5;
                          arg_list[9] = ")";
                          $$->text = Concat(10,arg_list);

#ifdef DEBUG
fprintf(debug,"Call Generated: %s\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       UNNEST sym_l_parn obj sym_comma attrname sym_r_parn
                        {
#ifdef DEBUG
fprintf(debug,"query -> UNNEST ( obj , attrname )\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          arg_list[0] = "INVOKE(";
                          arg_list[1] = $3->text;
                          arg_list[2] = ",";
                          arg_list[3] = "\"Unnest\"";
                          arg_list[4] = ",";
                          arg_list[5] = "1L,0L,0L,0L";
                          arg_list[6] = ",";
                          arg_list[7] = $5;
```

```
                        arg_list[8] = ")";
                        $$->text = Concat(9,arg_list);

#ifdef DEBUG
fprintf(debug,"Call Generated: %s\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       DUPELIM sym_l_parn obj sym_comma number sym_r_parn
                        {
#ifdef DEBUG
fprintf(debug,"query -> DUPELIM(obj, number)\n");
fflush(debug);
#endif DEBUG
                        $$ = Alloc_Var();
                        arg_list[0] = "INV";
                        arg_list[1] = "OKE(";
                        arg_list[2] = $3->text;
                        arg_list[3] = ",";
                        arg_list[4] = "\"DupEliminate\"";
                        arg_list[5] = ",";
                        arg_list[6] = "1L,0L,0L,0L";
                        arg_list[7] = ",";
                        arg_list[8] = $5;
                        arg_list[9] = ")";
                        $$->text = Concat(10,arg_list);

#ifdef DEBUG
fprintf(debug,"Call Generated: %s\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       COALESCE sym_l_parn obj sym_comma attrname sym_comma number sym_r_parn
                        {
#ifdef DEBUG
fprintf(debug,"query -> COALESCE(obj, attrname, number)\n");
fflush(debug);
#endif DEBUG
                        $$ = Alloc_Var();
                        arg_list[0] = "INVOKE(";
                        arg_list[1] = $3->text;
                        arg_list[2] = ",";
                        arg_list[3] = "\"Coalesce\"";
                        arg_list[4] = ",";
                        arg_list[5] = "2L,0L,0L,0L";
                        arg_list[6] = ",";
                        arg_list[7] = $5;
                        arg_list[8] = ",";
                        arg_list[8] = $7;
                        arg_list[10] = ")";
                        $$->text = Concat(11,arg_list);

#ifdef DEBUG
fprintf(debug,"Call Generated: %s\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        ;

image_func:     obj
                        {
#ifdef DEBUG
fprintf(debug,"image_func -> obj\n");
fflush(debug);
#endif DEBUG
```

```
                        $$ = $1; }
        |       func
                        {
#ifdef DEBUG
fprintf(debug, "image_func -> func\n");
fflush(debug);
#endif DEBUG
                        $$ = $1; }
        ;

attrname:       sym_identifier
                        {
#ifdef DEBUG
fprintf(debug,"attrname -> sym_identifier\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen("(ENFromString(\"") +
                              strlen($1) + strlen("\"))") + 1);
                        strcpy($$, "(ENFromString");
                        strcat($$, "(");
                        strcat($$, "\"");
                        strcat($$, $1);
                        strcat($$, "\"");
                        strcat($$, ")");
                        strcat($$, ")"); }
        ;

number  :       sym_constant
                        {
#ifdef DEBUG
fprintf(debug,"number -> sym_constant\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen("(ENFromLong(") + strlen($1) +
                                    strlen("))") + 1);
                        strcpy($$, "(ENFromLong");
                        strcat($$, "(");
                        strcat($$, $1);
                        strcat($$, ")");
                        strcat($$, ")"); }
        ;

obj     :       variable
                        {
#ifdef DEBUG
fprintf(debug,"obj -> variable\n");
fflush(debug);
#endif DEBUG
                        $$ = $1; }
        |       en_identifier
                        {
                        char    *type_of_coll;
                        bool    isPtr;
                        /* This is where a collection has been */
                        /* directly referenced, i.e. "Emps" */
#ifdef DEBUG
fprintf(debug,"obj -> en_identifier\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          $$->text = malloc(strlen($1) + 1);
                        /* Add the collection to the list if it's */
                        /* not there already */
                          if (check_param($1) == 0)
                          {
```

```
                        push_coll($1);
                    }

                    /* Here we call Lookup_Type($1) and get its */
                    /* "type" property (which is a string). If the */
                    /* type is "UnDefinedType" (or something like that), */
                    /* turn on the "run-time-type-check" switch and */
                    /* set $$->type to TYPEObject (the default). Otherwise */
                    /* assign the retrieved type  to $$->type, and also maybe */
                    /* store it in the Symbol_Table entry.   */

                    /*
                    type_of_coll = Lookup_Type($1);
                    if (!type_of_coll)
                    {
                        fprintf(debug,"ERROR -- collection %s undefined\n" $1);
                        fflush(debug);
                        exit(1);
                    }

                    $$->type =
                        GetTypeObject(GetBaseType(type_of_coll,&isPtr));
                    if (EN_OBJ_EQ($$->type, TYPEObject))
                    {
                        runtime_check = 1;
                    }
                    else
                    {
                        runtime_check = 0;
                    }

                    if (check_param($1) == 0)
                    {
                        push_coll($1, $$->type);
                    }

                    */
/****************** THIS LINE IS JUST TEMPORARY FOR TESTING ******************/
                    /*
                      $$->type = GetTypeObject("ColOfPerson");
                    */
                      $$->type = TYPEObject;
/*****************************************************************************/

                      strcpy($$->text, $1); }
        |       query
                    {
#ifdef DEBUG
fprintf(debug,"obj -> query\n");
fflush(debug);
#endif DEBUG
                      $$ = $1; }
        ;

lambdaexp:      LAMBDA sym_identifier
                    {
#ifdef DEBUG
fprintf(debug,"lambdaexp -> LAMBDA sym_identifier\n");
fflush(debug);
#endif DEBUG
                    /* This is the first lambda variable declared in the */
                    /* current query, so let's mark it as the start of a */
                    /* new scope */
                      push_param($2, 1); }
        |       lambdaexp LAMBDA sym_identifier
```

```
                        {
#ifdef DEBUG
fprintf(debug,"lambdaexp -> lambdaexp LAMBDA sym_identifier\n");
fflush(debug);
#endif DEBUG
                          push_param($3, 0); }
        ;

pred    :       exp     /* Maybe "func" instead? */
                        {
#ifdef DEBUG
fprintf(debug,"pred -> exp\n");
fflush(debug);
#endif DEBUG
                          $$ = malloc(strlen($1->text) + 1);
                          strcpy($$, $1->text);
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$);
fflush(debug);
#endif DEBUG
                        }
        |       pred sym_op_bit_and exp
                        {
#ifdef DEBUG
fprintf(debug, "pred -> pred && exp\n");
fflush(debug);
#endif DEBUG
                          $$ = malloc(strlen($1) + strlen(" && ")
                                + strlen($3->text) + 1);
                          strcpy($$, $1);
                          strcat($$, " && ");
                          strcat($$, $3->text);
#ifdef DEBUG
fprintf(debug,"OUTPUT: '%s'\n", $$);
fflush(debug);
#endif DEBUG
                        }
        |       pred sym_op_or exp
                        {
#ifdef DEBUG
fprintf(debug, "pred -> pred || exp\n");
fflush(debug);
#endif DEBUG
                          $$ = malloc(strlen($1) + strlen(" || ") +
                                        strlen($3->text) + 1);
                          strcpy($$, $1);
                          strcat($$, " || ");
                          strcat($$, $3->text);
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$);
fflush(debug);
#endif DEBUG
                        }
        |       exp sym_op_eq  exp      /* Need to think about Gail's "i-equality" */
                        {
                          if (!strcmp($2, "=="))
                          {
                                $$ = malloc(strlen("!ENUidCmp(") +
                                        strlen($1->text) + strlen(",") +
                                        strlen($3->text) + strlen(")") + 1);
#ifdef DEBUG
fprintf(debug, "pred -> exp == exp\n");
fflush(debug);
#endif DEBUG
                                strcpy($$, "!ENUidCmp(");
```

```
                        }
                        else
                        {
                              $$ = malloc(strlen("ENUidCmp(") +
                                    strlen($1->text) + strlen(",") +
                                    strlen($3->text) + strlen(")")  + 1);
#ifdef DEBUG
fprintf(debug, "pred -> exp != exp\n");
fflush(debug);
#endif DEBUG
                              strcpy($$, "ENUidCmp(");
                        }
                        strcat($$, $1->text);
                        strcat($$, ",");
                        strcat($$, $3->text);
                        strcat($$, ")");
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n",$$);
fflush(debug);
#endif DEBUG
                    }
        |       exp sym_op_rel exp
                    {
                     if (!runtime_check)
                     {
                           /* Both exp's must be numbers or strings */
                           if ( !( (EN_OBJ_EQ($1->type, TYPEInteger) &&
                                    EN_OBJ_EQ($3->type, TYPEInteger))    ||
                                (EN_OBJ_EQ($1->type, TYPEString) &&
                                    EN_OBJ_EQ($3->type, TYPEString)) ) )
                           {
                                 fprintf(debug,"ERROR -- arguments to <= must be of type ENInteger\n");
                                 fflush(debug);
                                 exit(1);
                           }
                     }
                     else
                     {
                     }
                     if (!(strcmp($2, "<=")))
                     {
                           printf("pred -> exp <= exp\n");
                     fflush(stdout);
                           arg_list[0] = "EN_OBJ_LEQ(";
                     }
                     else if (!(strcmp($2, ">=")))
                     {
                           printf("pred -> exp >= exp\n");
                     fflush(stdout);
                           arg_list[0] = "EN_OBJ_GEQ(";
                     }
                     else if (!(strcmp($2, ">")))
                     {
                           printf("pred -> exp > exp\n");
                     fflush(stdout);
                           arg_list[0] = "EN_OBJ_GT(";
                     }
                     else if (!(strcmp($2, ">")))
                     {
                           printf("pred -> exp > exp\n");
                     fflush(stdout);
                           arg_list[0] = "EN_OBJ_LT(";
                     }
                     arg_list[1] = $1->text;
                     arg_list[2] = ",";
```

```
                        arg_list[3] = $3->text;
                        arg_list[4] = ")";

                        $$ = Concat(5,arg_list);
                        printf("OUTPUT: '%s'\n",$$);
                        }
        |       exp SUBSETOF exp
                        {
                        ENType enType1;
                        ENType enType3;
#ifdef DEBUG
fprintf(debug, "pred -> exp SUBSETOF exp\n");
fflush(debug);
#endif DEBUG
                        /*
                        if (!runtime_check)
                        {
                              enType1 = TYPE(GET_PROP_VALUE($1->type, "type", 0L));
                              enType3 = TYPE(GET_PROP_VALUE($3->type, "type", 0L));
                              if ( EN_OBJ_NEQ(enType1, TYPEColType) ||
                                   EN_OBJ_NEQ(enType3, TYPEColType) )
                              {
                                      fprintf(debug,"ERROR -- arguments to SubsetOf must be collections\n");
                                      fflush(debug);
                                      exit(1);
                              }
                        }
                        else
                        {
                        }
                        */
                        $$ = malloc(strlen("SubsetOf(") + strlen($1->text) +
                              strlen(",") + strlen($3->text) + strlen(")")+1);
                        strcpy($$, "SubsetOf(");
                        strcat($$, $1->text);
                        strcat($$, ",");
                        strcat($$, $3->text);
                        strcat($$, ")");
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n",$$);
fflush(debug);
#endif DEBUG
                        }
        |       exp IN exp
                        {
                        ENType enType;
#ifdef DEBUG
fprintf(debug, "pred -> exp IN exp\n");
fflush(debug);
#endif DEBUG
                        /*
                        if (!runtime_check)
                        {
                              enType = TYPE(GET_PROP_VALUE($3->type, "type", 0L));
                              if ( EN_OBJ_NEQ(enType, TYPEColType) )
                              {
                                      fprintf(debug,"ERROR -- second argument to In must be a collection\n");
                                      fflush(debug);
                                      exit(1);
                              }
                        }
                        else
                        {
                        }
                        */
```

```
                        $$ = malloc(strlen("MemberOf(") + strlen($1->text) +
                              strlen(",") + strlen($3->text) + strlen(")")+1);
                        strcpy($$, "MemberOf(");
                        strcat($$, $1->text);
                        strcat($$, ",");
                        strcat($$, $3->text);
                        strcat($$, ")");
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n",$$);
fflush(debug);
#endif DEBUG
                        }
        |       NOT pred
                        {
#ifdef DEBUG
fprintf(debug, "pred -> NOT pred\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc( strlen("(!(") + strlen($2) +
                                    strlen("))") + 1);
                        strcpy($$, "(!(");
                        strcat($$, $2);
                        strcat($$, "))");
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n",$$);
fflush(debug);
#endif DEBUG
                        }
        ;

exp     :       t
                        {
#ifdef DEBUG
fprintf(debug, "exp -> t\n");
fflush(debug);
#endif DEBUG
                        $$ = $1;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       exp sym_op_plus t
                        { ENType enType1, enType2;
#ifdef DEBUG
fprintf(debug, "exp -> exp + t\n");
fflush(debug);
#endif DEBUG
                        /*
                          if (!runtime_check)
                          {
                                if ( EN_OBJ_NEQ($1->type, TYPEInteger) ||
                                     EN_OBJ_NEQ($3->type, TYPEInteger) )
                                {
                                    fprintf(debug,"ERROR -- arguments to '+' must be of type ENInteger\n");
                                    fflush(debug);
                                    exit(1);
                                }
                          }
                          else
                          {
                          }
                        */
                          $$ = Alloc_Var();
                          $$->text = malloc(strlen("Add(") + strlen($1->text) +
```

```
                        strlen(",") + strlen($3->text) + strlen(")")+1);
                    strcpy($$->text, "Add(");
                    strcat($$->text, $1->text);
                    strcat($$->text, ",");
                    strcat($$->text, $3->text);
                    strcat($$->text, ")");
                    $$->type = TYPEInteger;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                    }
        |       exp sym_op_minus t
                    {
#ifdef DEBUG
fprintf(debug, "exp -> exp - t\n");
fflush(debug);
#endif DEBUG
                    /*
                      if (!runtime_check)
                      {
                            if ( EN_OBJ_NEQ($1->type, TYPEInteger) ||
                                 EN_OBJ_NEQ($3->type, TYPEInteger) )
                            {
                                fprintf(debug,"ERROR -- arguments to '-' must be of type ENInteger\n");
                                fflush(debug);
                                exit(1);
                            }
                      }
                      else
                      {
                      }
                    */
                      $$ = Alloc_Var();
                      $$->text = malloc(strlen("Sub(") + strlen($1->text) +
                            strlen(",") + strlen($3->text) + strlen(")")+1);
                      strcpy($$->text, "Sub(");
                      strcat($$->text, $1->text);
                      strcat($$->text, ",");
                      strcat($$->text, $3->text);
                      strcat($$->text, ")");
                      $$->type = TYPEInteger;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                    }
        |       query /* ? */
                    {
#ifdef DEBUG
fprintf(debug,"exp -> query\n");
fflush(debug);
#endif DEBUG
                      $$ = $1;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$);
fflush(debug);
#endif DEBUG
                    }
        ;

t       :       f
                    {
#ifdef DEBUG
fprintf(debug, "t -> f\n");
```

```
fflush(debug);
#endif DEBUG
                        $$ = $1;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                    }
        |       t sym_op_mult f
                    {
#ifdef DEBUG
fprintf(debug, "t -> t * f\n");
fflush(debug);
#endif DEBUG
                    /*
                      if (!runtime_check)
                      {
                            if ( EN_OBJ_NEQ($1->type, TYPEInteger) ||
                                 EN_OBJ_NEQ($3->type, TYPEInteger) )
                            {
                                fprintf(debug,"ERROR -- arguments to '*' must be of type ENInteger\n");
                                fflush(debug);
                                exit(1);
                            }
                      }
                      else
                      {
                      }
                    */
                      $$ = Alloc_Var();
                      $$->text = malloc(strlen("Mult(") + strlen($1->text) +
                             strlen(",") +strlen($3->text) + strlen(")")+1);
                      strcpy($$->text, "Mult(");
                      strcat($$->text, $1->text);
                      strcat($$->text, ",");
                      strcat($$->text, $3->text);
                      strcat($$->text, ")");
                      $$->type = TYPEInteger;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                    }
        |       t sym_op_div f
                    {
#ifdef DEBUG
fprintf(debug, "t -> t / f\n");
fflush(debug);
#endif DEBUG
                    /*
                      if (!runtime_check)
                      {
                            if ( EN_OBJ_NEQ($1->type, TYPEInteger) ||
                                 EN_OBJ_NEQ($3->type, TYPEInteger) )
                            {
                                fprintf(debug,"ERROR -- arguments to '/' must be of type ENInteger\n");
                                fflush(debug);
                                exit(1);
                            }
                      }
                      else
                      {
                      }
                    */
                      $$ = Alloc_Var();
```

```
                        $$->text = malloc(strlen("Div(") + strlen($1->text) +
                              strlen(",") + strlen($3->text) + strlen(")")+1);
                        strcpy($$->text, "Div(");
                        strcat($$->text, $1->text);
                        strcat($$->text, ",");
                        strcat($$->text, $3->text);
                        strcat($$->text, ")");
                        $$->type = TYPEInteger;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                      }
        ;

f       :       sym_string
                      {
#ifdef DEBUG
fprintf(debug, "f -> sym_string\n");
fflush(debug);
#endif DEBUG
                        $$ = Alloc_Var();
                        $$->text = malloc(strlen("(ENFromString(") +
                              strlen($1) + strlen("))") + 1);
                      /* strings must be "cast" into ENString objects */
                        strcpy($$->text, "(ENFromString");
                        strcat($$->text, "(");
                        strcat($$->text, $1);
                        strcat($$->text, ")");
                        strcat($$->text, ")");
                        $$->type = TYPEString;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                      }
        |       variable
                      {
#ifdef DEBUG
fprintf(debug, "f -> variable\n");
fflush(debug);
#endif DEBUG
                        $$ = $1;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                      }
        |       sym_constant
                      {
#ifdef DEBUG
fprintf(debug, "f -> sym_constant\n");
fflush(debug);
#endif DEBUG
                        $$ = Alloc_Var();
                        $$->text = malloc(strlen("(ENFromLong(") +
                                    strlen($1) + strlen("))") + 1);
                      /* numbers must be "cast" into ENInteger objects */
                        strcpy($$->text, "(ENFromLong");
                        strcat($$->text, "(");
                        strcat($$->text, $1);
                        strcat($$->text, ")");
                        strcat($$->text, ")");
                        $$->type = TYPEInteger;
#ifdef DEBUG
```

```
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       TRUE_TOKEN
                        {
#ifdef DEBUG
fprintf(debug, "f -> TRUE\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          $$->text = malloc("EN_TRUE");
                          strcpy($$->text, "EN_TRUE");
                          $$->type = TYPEBoolean;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       FALSE_TOKEN
                        {
#ifdef DEBUG
fprintf(debug, "f -> FALSE_TOKEN\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          $$->text = malloc("EN_FALSE");
                          strcpy($$->text, "EN_FALSE");
                          $$->type = TYPEBoolean;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       sym_l_parn exp sym_r_parn
                        {
#ifdef DEBUG
fprintf(debug, "f -> (exp)\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          $$->text = malloc(strlen("(") + strlen($2) +
                                  strlen(")") + 1);
                          strcpy($$->text, "(");
                          strcat($$->text, $2);
                          strcat($$->text, ")");
                          $$->type = $2->type;
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       sym_l_parn pred sym_r_parn
                        {
#ifdef DEBUG
fprintf(debug, "f -> (pred)\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          $$->text = malloc(strlen("(") + strlen($2) +
                                  strlen(")")  + 1);
                          strcpy($$->text, "(");
                          strcat($$->text, $2);
                          strcat($$->text, ")");
                          $$->type = TYPEBoolean;
```

```
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        ;

funcname:       sym_identifier  /* Should probably assign $$->type (check Symbol_Table) */
                        {
#ifdef DEBUG
fprintf(debug, "funcname -> sym_identifier\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          $$->text = malloc(strlen($1) + 1);
                          strcpy($$->text, $1);
#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        ;

func    :       funcname sym_l_parn explist sym_r_parn
                        {
#ifdef DEBUG
fprintf(debug, "func -> ( explist )\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          $$->text = malloc(strlen($1->text) + strlen("(") +
                                strlen($3) + strlen(")")+ 1);
                          strcpy($$->text, $1->text);
                          strcat($$->text, "(");
                          strcat($$->text, $3);
                          strcat($$->text, ")");
                          $$->type = $1->type;
                        }
        ;

explist :       exp
                        {
#ifdef DEBUG
fprintf(debug, "explist -> exp\n");
fflush(debug);
#endif DEBUG
                          $$ = malloc(strlen($1->text) + 1);
                          strcpy($$, $1->text); }
        |       explist sym_comma exp
                        {
#ifdef DEBUG
fprintf(debug, "explist -> explist, exp\n");
fflush(debug);
#endif DEBUG
                          $$ = malloc(strlen($1) + strlen(",") +
                                        strlen($3->text) + 1);
                          strcpy($$, $1);
                          strcat($$, ",");
                          strcat($$, $3->text); }
        ;

variable:       sym_identifier
                        {
#ifdef DEBUG
fprintf(debug, "variable -> sym_identifier\n");
```

```
fflush(debug);
#endif DEBUG
                        /* This is a reference to a lambda variable; */
                        /* check if it's been declared */
                          if (check_param($1) == 0)
                          {
                                fprintf(debug,"ERROR - symbol %s undeclared\n",$1);
                                fflush(debug);
                                exit(1);
                          }
                          $$ = Alloc_Var();
                          $$->text = malloc(strlen($1) + 1);
                          strcpy($$->text,$1);

                        /* Get its type */
                        /*
                          if (!runtime_check)
                          {
                                $$->type = Find_Type($1);
                          }
                          else
                        */
                          {
                                $$->type = TYPEObject;
                          }

#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                        }
        |       variable sym_at sym_identifier
                        {
                          ENPropertyType prop_type;
                          ENString enString;

#ifdef DEBUG
fprintf(debug, "variable -> variable @ sym_identifier\n");
fflush(debug);
#endif DEBUG
                          $$ = Alloc_Var();
                          $$->text = malloc(strlen("GET_PROP_VALUE(") +
                                strlen($1->text) + 2*strlen(",") +
                                strlen($3) + strlen("0L)") + 11);
                          /*
                          strcpy($$, $1);
                          strcat($$, "@");
                          strcat($$, $3);
                          */
                          strcpy($$->text,"GET_PROP_");
                          strcat($$->text, "VALUE(");
                          strcat($$->text, $1->text);
                          strcat($$->text, ",\"");
                          strcat($$->text, $3);
                          strcat($$->text, "\",");
                          strcat($$->text, "0L");
                          strcat($$->text, ")");

                        /** type-check here
                          if (!runtime_check)
                          {
                                enString = ENFromString($3);
                                prop_type = PROPTYPE(INVOKE($1->type, "GetPropertyType", 1L,0L,0L,0L,enString) );
                                $$->type = TYPE(GET_PROP_VALUE(prop_type, "valueClass", 0L));
```

```
#ifdef DEBUG
fprintf(debug, "Type of the value of this property: %s\n", ENToString( STRING( GET_PROP_VALUE($$->type,"name", 0L) ) ) );
fflush(debug);
#endif DEBUG
                        }
                        else
                      */
                        {
                              $$->type = TYPEObject;
                        }

#ifdef DEBUG
fprintf(debug, "OUTPUT: '%s'\n", $$->text);
fflush(debug);
#endif DEBUG
                      }
        ;

tuple   :       sym_l_brace pairlist sym_r_brace
                      {
#ifdef DEBUG
fprintf(debug,"tuple -> {pairlist}\n");
fflush(debug);
#endif DEBUG
                        $$ = malloc(strlen("EnBuildArgList(") +
                              strlen(itoa($2->numpairs)) + strlen(",") +
                              strlen($2->text) + strlen(")")+ 1);
                        strcpy($$, "ENBuildArgList(");
                        strcat($$, itoa($2->numpairs));
                        strcat($$, ",");
                        strcat($$, $2->text);
                        strcat($$, ")");
                      }
        ;

pairlist:       pair
                      {
#ifdef DEBUG
fprintf(debug, "pairlist -> pair\n");
fflush(debug);
#endif DEBUG
                        $$ = Alloc_Duple();
                        $$->text = malloc(strlen($1) + 1);
                        strcpy($$->text, $1);
                        $$->numpairs = 1;
                      }
        |       pairlist sym_comma pair
                      {
#ifdef DEBUG
fprintf(debug, "pairlist -> pairlist, pair\n");
fflush(debug);
#endif DEBUG
                        $$ = Alloc_Duple();
                        $$->text = malloc(strlen($1->text) + strlen(",") +
                              strlen($3)  + 1);
                        strcpy($$->text, $1->text);
                        strcat($$->text, ",");
                        strcat($$->text, $3);
                        $$->numpairs = $1->numpairs + 1;
                      }
        ;

pair    :       sym_l_parn funcname sym_comma obj sym_r_parn
                      {
#ifdef DEBUG
```

```
fprintf(debug, "pair -> (attrname, obj)\n");
fflush(debug);
#endif DEBUG
                        arg_list[0] = "\"";
                        arg_list[0] = $2->text;
                        arg_list[0] = "\"";
                        arg_list[0] = ", ";
                        if ( (strncmp($4->text, "Select(", 7) == 0) ||
                             (strncmp($4->text, "Project(", 8) == 0) ||
                             (strncmp($4->text, "Ojoin(", 6) == 0) ||
                             (strncmp($4->text, "Image(", 6) == 0) ||
                             (strncmp($4->text, "Flatten(", 8) == 0) ||
                             (strncmp($4->text, "DupEliminate(", 13) == 0) ||
                             (strncmp($4->text, "Coalesce(", 9) == 0) ||
                             (strncmp($4->text, "Nest(", 5) == 0) ||
                             (strncmp($4->text, "Unnest(", 7) == 0) ||
                       (strncmp($4->text, "INVOK", 5) == 0)) /* TEMPORARY */
                        {
                              arg_list[1] = "&";
                              arg_list[2] = gen_FuncOp_code($4->text);
                           /* may also have to pass in self's type */
                           /* (from top of lambda-var stack) */
                        }
                        else
                        if ((strchr($4->text, '@')) ||
                              (strncmp($4->text, "GET_PROP_VALU", 13) == 0)) /* TEMPORARY */
                        {
                              arg_list[1] = "&";
                              arg_list[2] = gen_FuncOp_code($4->text);
                           /* may also have to pass in self's type */
                           /* (from top of lambda-var stack) */
                        }
                        else  /* Need to do something different here.... */
                        {
                              arg_list[1] = "&";
                              arg_list[2] = gen_FuncOp_code($4->text);
                           /* may also have to pass in self's type */
                           /* (from top of lambda-var stack) */
                        }
                        $$ = Concat(3,arg_list);
#ifdef DEBUG
fprintf(debug,"Call Generated: %s\n", $$);
fflush(debug);
#endif DEBUG
                     }
        ;
%%

/*****************************************************************************/

#include "scanner.c"

/*****************************************************************/

VAR *
Alloc_Var()
{
        VAR *new = (VAR *)malloc (sizeof (VAR));

        if (new == 0)
        {
                printf ("Error-Alloc_Var-malloc returned 0\n");
                return 0;
        }
```

```
        new->text = 0;

        return(new);
}

/*********************************************************************/

DUPLE *
Alloc_Duple()
{
        DUPLE *new = (DUPLE *)malloc (sizeof (DUPLE));

        if (new == 0)
        {
                printf ("Error-Alloc_Duple-malloc returned 0\n");
                return 0;
        }

        new->text = 0;
        new->numpairs = 0;

        return(new);
}

/*********************************************************************/

param_descr *
Alloc_ParamDescr()
{
        param_descr *new = (param_descr *)malloc (sizeof(param_descr));

        if (new == 0)
        {
                printf ("Error-Alloc_ParamDescr-malloc returned 0\n");
                return 0;
        }

        new->name = 0;
        new->first_in_frame = 0;
        new->next = 0;
        new->prev = 0;

        return(new);
}

/*********************************************************************/

coll_descr *
Alloc_CollDescr()
{
        coll_descr *new = (coll_descr *)malloc (sizeof(coll_descr));

        if (new == 0)
        {
                printf ("Error-Alloc_CollDescr-malloc returned 0\n");
                return 0;
        }

        new->name = 0;
        new->next = 0;
        new->prev = 0;

        return(new);
}
```

```
/*****************************************************************/

/* Find_Type() searches through the lambda-variable stack for the */
/* variable "var_str" and returns its type */

ENType
Find_Type(var_str)
char *var_str;
{
        param_descr *param_list;
        param_list = StackFrame->prev;

        if (param_list)
        do {
                if (!strcmp(param_list->name, var_str))
                {
#ifdef DEBUG
fprintf(debug,"Found type of %s -- it's %s\n", var_str, ENToString(STRING(GET_PROP_VALUE(param_list->type, "name",0L))) );
#endif DEBUG
                        return(param_list->type);
                }
                param_list = param_list->prev;
        } while (param_list);
}

/*****************************************************************/

Power(x, n)
int x, n;
{
        int i, p;

        p = 1;
        for (i = 1; i <= n; ++i)
        {
                p = p*x;
        }
        return(p);
}

/* This is a general purpose routine which converts an integer into its
   ASCII-string representation. */

char *
itoa(number)
int number;
{
        int i, n, temp, m;
        char *number_chars = (char *)malloc(32);
        sprintf(number_chars,"%d",number);
#ifdef DEBUG
fprintf(debug,"itoa(%d): returning %s\n",number, number_chars);
fflush(stderr);
#endif DEBUG
        return(number_chars);
}

/*****************************************************************/

/* Assign type to lambda variable on stack */

assign_type(var_type)
ENType var_type;
{
        param_descr *param_list;
```

```
        param_list = StackFrame->prev;
        if (param_list)
        {
                param_list->type = var_type;
        }
}

/*****************************************************************/

/* Assign types to lambda variables (used in OJOIN) on stack */

ojoin_assign_type(var1_type, var2_type)
ENType var1_type;
ENType var2_type;
{
        param_descr *param_list;

        param_list = StackFrame->prev;
        if (param_list)
        {
                param_list->type = var2_type;
        }

        if (param_list->prev)
        {
                (param_list->prev)->type = var1_type;
        }
}

/*****************************************************************/

/* push_param() pushes lambda-variable "symbol" onto the lambda-variable */
/* stack. first_in_list is set if it's the 1st variable declared in */
/* the current scope */

push_param(symbol, first_in_list)
char *symbol;
int first_in_list;
{
        param_descr *saved_param;

        if (!StackFrame)
        {
#ifdef DEBUG
fprintf(debug,"First entry into StackFrame\n");
fflush(debug);
#endif DEBUG
                StackFrame = Alloc_ParamDescr();
        }
#ifdef DEBUG
fprintf(debug,"push_param: Pushing '%s'\n",symbol);
fflush(debug);
#endif DEBUG
        StackFrame->name = malloc(strlen(symbol) + 1);
        strcpy(StackFrame->name, symbol);
        StackFrame->first_in_frame = first_in_list;
        StackFrame->next = Alloc_ParamDescr();
        saved_param = StackFrame;
        StackFrame = StackFrame->next;
        StackFrame->prev = saved_param;
}

/*****************************************************************/
```

```
/* push_coll() addres collection "coll_name" to the list */

push_coll(coll_name)    /* should also pass in ENType coll_type */
char *coll_name;
{
        coll_descr *saved_coll;
        coll_descr *colls;

        if (!Coll_List)
        {
#ifdef DEBUG
fprintf(debug, "First entry into Coll_List\n");
fflush(debug);
#endif DEBUG
                Coll_List = Alloc_CollDescr();
        }
        else
        {
                colls = Coll_List->prev;
                if (colls)
                do {
                        if (!(strcmp(coll_name, colls->name)))
                        {
#ifdef DEBUG
fprintf(debug, "collection '%s' already in list\n",colls->name);
fflush(debug);
#endif DEBUG
                                return(0);
                        }
                        colls = colls->prev;
                } while (colls);
        }

#ifdef DEBUG
fprintf(debug, "push_coll: Pushing '%s'\n",coll_name);
fflush(debug);
#endif DEBUG
        Coll_List->name = malloc(strlen(coll_name) + 1);
        strcpy(Coll_List->name, coll_name);
        /*
        Coll_list->type = coll_type;
        */
        Coll_List->next = Alloc_CollDescr();
        saved_coll = Coll_List;
        Coll_List = Coll_List->next;
        Coll_List->prev = saved_coll;
}

/*********************************************************************/

/* pop_params() discards local variables for the current scope */

pop_params()
{
        do
        {
                StackFrame = StackFrame->prev;
#ifdef DEBUG
fprintf(debug, "pop_params: Popping %s\n", StackFrame->name);
fflush(debug);
#endif DEBUG
        } while (!(StackFrame->first_in_frame));
}

/*********************************************************************/
```

```
/* check_param() returns 0 if "symbol" is not on the lambda-variable stack */

check_param(symbol)
char *symbol;
{
        param_descr *param_list;
        int found_it = 0;

        if (StackFrame)
        {
                param_list = StackFrame->prev;

                do {
                        if (!(strcmp(param_list->name,symbol)))
                        {
#ifdef DEBUG
fprintf(debug, "Found arg '%s' in param_list\n",symbol);
fflush(debug);
#endif DEBUG
                                found_it = 1;
                        }
                        else
                        {
                                param_list = param_list->prev;
                        }
                } while ( (!(found_it)) && (param_list) );
        }

        if (!(found_it))
        {
                return(0);
        }
        else
        {
                return(1);
        }
}

/*********************************************************************/

/* write_coll_list() writes out all the collections */
/* used in the query in the form of a function arg-list */

write_coll_list(startPos,list)
short   *startPos;
char    *list[];
{
        char            *text;
        coll_descr      *colls;

        colls = Coll_List->prev;

        if (colls)
                {
                do {
                        list[(*startPos)++] =  colls->name;
                        if (colls->prev)
                        {
                                list[(*startPos)++] = ",";
                        }
                        colls = colls->prev;
                   } while (colls);
                }
}
```

```
/*****************************************************************/

/* write_coll_decls() writes out declarations for all the collections */
/* used in the query */

write_coll_decls(startPos,list)
short *startPos;
char    *list[];
{
        coll_descr *colls;
        colls = Coll_List->prev;

        if (colls)
        do {
                if (!runtime_check)
                {
                        list[(*startPos)++] = "ENObject ";
                }
                else
                {
                        list[(*startPos)++] = "ENObject ";
                }
                list[(*startPos)++] = colls->name;
                list[(*startPos)++] = ";";
                if (colls->prev)
                {
                        list[(*startPos)] = "\n";
                }
                colls = colls->prev;
        } while (colls);
}

/*****************************************************************/

/* make_BuildArg() writes out a call to ENBuildArgList() putting all */
/* the collections referenced thus far into a list "args" */

make_BuildArg(startPos,list)
short   *startPos;
char    *list[];
{
        coll_descr *colls;
        int arg_count = 0;
        int saved_pos;

        arg_list[(*startPos)++] =  "\n";
        arg_list[(*startPos)++] =  "     args = ENBuildArgList(";
        saved_pos = *startPos;
        arg_list[(*startPos)++] = itoa(arg_count);
        arg_list[(*startPos)++] = ",";

        colls = Coll_List->prev;
        if (colls)
        do {
                arg_list[(*startPos)++] =  "\"";
                arg_list[(*startPos)++] = colls->name;
                arg_list[(*startPos)++] = "\"";
                arg_list[(*startPos)++] = ",";
                arg_list[(*startPos)++] = " &";
                arg_list[(*startPos)++] = colls->name;
                if (colls->prev)
                {
                        arg_list[(*startPos)++] = ",";
                }
```

```
                arg_count++;
                colls = colls->prev;
        } while (colls);

        arg_list[(*startPos)++] = ");\n";
        arg_list[saved_pos] = itoa(arg_count);
}

/*****************************************************************/

/* make_GetArg() writes out a series of calls to ENGetArg(), which extracts */
/* collections from the list "args" */

char *
make_GetArg()
{
        char            *list[100];
        coll_descr      *colls;
        char            *text;
        short           pos;

        pos = 0;
        colls = Coll_List->prev;
        if (colls)
        do {
                list[pos++] = "     ";
                list[pos++] =  colls->name;
                list[pos++] =  " = ENGetArg(args, ";
                list[pos++] =  "\"";
                list[pos++] =  colls->name;
                list[pos++] =  "\"";
                list[pos++] =  ");";
                list[pos++] =  "\n";
                colls = colls->prev;
        } while (colls);

        text = Concat(pos,list);

        return(text);
}

/*****************************************************************/

/* make_AddArgs() writes out a call to ENAddArgs(), which adds the current */
/* scope's lambda variables to the list "args" */

char *
make_AddArgs()
{
        char            *list[100];
        param_descr *param_list;
        char *code;
        int arg_count = 0;
        short           pos;

        list[0] = "     ENAddArgs(&args, ";
        /* leave room for arc count below */
        pos = 2;

        param_list = StackFrame;
        do {
                param_list = param_list->prev;

                list[pos++] = ", ";
                list[pos++] = "\"";
```

```
                list[pos++] = param_list->name;
                list[pos++] = "\"";
                list[pos++] = ", ";
                list[pos++] = "&";
                list[pos++] = param_list->name;
                arg_count++;

        } while (!(param_list->first_in_frame));

        list[1] = itoa(arg_count);
        list[pos++] = ");";
        list[pos++] = "\n";

        code = Concat(pos,list);
        return(code);
}

/*********************************************************************/

/* gen_pred_code() builds a boolean function which returns EN_TRUE or EN_FALSE */
/* based on the value of the test in pred_str. "self_type" is the type of the */
/* 1st argument of the function to be built */

char *
gen_pred_code(pred_str, self_type)
char *pred_str;
ENType self_type;
{
        FILE *fs;
        param_descr *param_list;
        char *routine_name = malloc(128);
        char *short_routine_name = malloc(128);
        char *GetArg_stmts;
        char *coll_decls;
        char *local_inits;
        char *AddArgs_stmt;
        /*
        char *includes;
        */
        char *system_arg = malloc(64);
        char *file_name;
        char *type_name;
        char    *my_args[100];
        ENString enString;
        ENBytes args;
        ENOperationType newOp;
        int val1, val2, random;
        ENInteger seg;
        ENType enType;
        short           pos;

#ifdef DEBUG
fprintf(debug,"Just entered gen_pred_code\n");
fflush(debug);
#endif DEBUG

        random = rand();
        /*
        fs = fopen("output.c", "a+");
        */

        /* Start to generate the name of the function to be constructed. */
        /* All functions generated within a process must be uniquely named, */
        /* so we append a unique random number to the prefix "PredOp" */
```

```
        strcpy(short_routine_name, "PredOp");
        strcat(short_routine_name, itoa(random) );
        file_name = malloc(strlen(gLoadDir) + strlen("/") +
                        strlen(short_routine_name) + strlen(".c") + 1);

        /* The portion of the name generated thus far will serve as the name */
        /* of the file containing the function */

        sprintf(file_name, "%s/%s.c",gLoadDir,short_routine_name);
        fs = fopen(file_name, "w");

        fwrite(includes, 1, strlen(includes), fs);

/* Generate name and args of routine */

        /* The name portion is now prefixed with the type of the "self" argument */

        /** Is this necessary?
        if (!runtime_check)
        {
                enString = STRING( GET_PROP_VALUE(self_type,"name",0L) );
                strcpy(routine_name, ENToString(enString));
        }
        else
        */
        {
                strcpy(routine_name, "Object");
        }
        strcat(routine_name, "_PredOp");
        strcat(routine_name, itoa(random) );

        fwrite("ENObject", 1, strlen("ENObject"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite(routine_name, 1, strlen(routine_name), fs);
        fwrite("(", 1, strlen("("), fs);

        /* write out the arg-list of the function; */
        /* this consists of the lambda-variables defined within */
        /* the current scope, plus the list "args", */
        /* which will contain all others */

        param_list = StackFrame;
        do {
                param_list = param_list->prev;
                fwrite(param_list->name, 1, strlen(param_list->name), fs);
                fwrite(", ", 1, 2, fs);

        } while (!(param_list->first_in_frame));

        fwrite("args)", 1, strlen("args)"), fs);
        fwrite("\n", 1, 1, fs);

/* Generate arg declarations of routine */

        param_list = StackFrame;
        do {
                param_list = param_list->prev;
                /** Is this necessary?
                if (!runtime_check)
                {
                        type_name = ENToString(STRING(GET_PROP_VALUE(param_list->type, "name",0L)));
                        fwrite("EN", 1, 2, fs);
                        fwrite(type_name, 1, strlen(type_name), fs);
                        fwrite(" ", 1, 1, fs);
                }
```

```
                else
                */
                {
                        fwrite("ENObject ", 1, strlen("ENObject "), fs);
                }

                fwrite(param_list->name, 1, strlen(param_list->name), fs);
                fwrite(";", 1, strlen(";"), fs);
                fwrite("\n", 1, 1, fs);
        } while (!(param_list->first_in_frame));

        fwrite("ENObject args;", 1, strlen("ENObject args;"), fs);
        fwrite("\n", 1, 1, fs);

        fwrite("{", 1, strlen("{"), fs);
        fwrite("\n", 1, 1, fs);

/* Generate local variable declarations -- this is for all */
/* variables defined in previous scopes */

        local_inits[0] = '\0';

        if (param_list->prev)
        {
                pos = 0;
                param_list = param_list->prev;
                do
                {
                        /** Is this necessary?
                        if (!runtime_check)
                        {
                                enString = STRING(GET_PROP_VALUE(param_list->type, "name",0L));
                                type_name = ENToString(enString);
                                fwrite("EN", 1, 2, fs);
                                fwrite(type_name, 1, strlen(type_name), fs);
                                fwrite(" ", 1, 1, fs);
                        }
                        else
                        */
                        {
                                fwrite("     ENObject ", 1, strlen("     ENObject "), fs);
                        }

                        fwrite(param_list->name, 1, strlen(param_list->name), fs);
                        fwrite(";", 1, strlen(";"), fs);
                        fwrite("\n", 1, 1, fs);

                        /* Now that we've written out the local variable */
                        /* declarations, they must be initialized by */
                        /* being extracted from "args". So we generate */
                        /* the necessary ENGetArg() calls and save them */
                        /* to be written out later */

                        my_args[pos++] = "     ";
                        my_args[pos++] = param_list->name;
                        my_args[pos++] = " = ENGetArg(args,";
                        my_args[pos++] = "\"";
                        my_args[pos++] = param_list->name;
                        my_args[pos++] = "\"";
                        my_args[pos++] = ");\n";

                        param_list = param_list->prev;

                } while (param_list);
                local_inits = Concat(pos,my_args);
```

```
        }

        /* Now we write out the collection declarations */

        pos = 0;
        write_coll_decls(&pos,my_args);
        coll_decls = Concat(pos,my_args);
        fwrite(coll_decls, 1, strlen(coll_decls), fs);
        fwrite("\n", 1, 1, fs);

        /* Write out declarations for operation-objects */

        fwrite(declarations, 1, strlen(declarations), fs);
        fwrite("\n\n", 1, 1, fs);

        /* NOW write out the ENGetArg() calls which extract local */
        /* variables from "args" */

        fwrite(local_inits, 1, strlen(local_inits), fs);

        /* make_GetArg() generates ENGetArg() calls which extract */
        /* collections from "args" */

        GetArg_stmts = make_GetArg();
        fwrite(GetArg_stmts, 1, strlen(GetArg_stmts), fs);
        fwrite("\n", 1, 1, fs);

        /* Now write out all initializations for the operation-objects */

        fwrite(initializations, 1, strlen(initializations), fs);
        fwrite("\n", 1, 1, fs);

/* Generate body of routine */

        AddArgs_stmt = make_AddArgs();
        fwrite(AddArgs_stmt, 1, strlen(AddArgs_stmt), fs);

        fwrite("\n", 1, 1, fs);
        fwrite("    if (", 1, strlen("    if ("), fs);
        fwrite(pred_str, 1, strlen(pred_str), fs);
        fwrite(")", 1, strlen(")"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("    {", 1, strlen("    {"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("        return(EN_TRUE);", 1, strlen("        return(EN_TRUE);"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("    }", 1, strlen("    }"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("    else", 1, strlen("    else"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("    {", 1, strlen("    {"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("        return(EN_FALSE);", 1, strlen("        return(EN_FALSE);"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("    }", 1, strlen("    }"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("}", 1, strlen("}"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("\n", 1, 1, fs);
        fclose(fs);

        /* We now have a complete boolean function in a file. */
        /* It must be run through a speciall scanner and then compiled */

        /*
```

```
        strcpy(system_arg, "sh -x comp ");
        strcat(system_arg, short_routine_name);
        system(system_arg);
        */

        /* The new function is added to the list of operation declarations */

        my_args[0] = declarations;
        my_args[1] = "     ENObject ";
        my_args[2] = short_routine_name;
        my_args[3] = ";\n";
        declarations = Concat(4,my_args);

        /* Now we create an operation-object from the function, */
        /* and then add it to the database */

        enString = ENFromString(short_routine_name);
        if (!runtime_check)
        {
                args = ENBuildArgList(3, "name", &enString, "refines", &TYPEOperation, "owningType", &self_type );
        }
        else
        {
                args = ENBuildArgList(3, "name", &enString, "refines", &TYPEOperation, "owningType", &TYPEObject);
        }
        seg = ENFromLong(TYPE_SEG);
        newOp = OPTYPE(INVOKE(TYPEOperationType, "CreateInstance",2L,0L,0L,0L, seg, args));

        enString = ENFromString(short_routine_name);

#ifdef DEBUG
fprintf(debug,"Calling NewSource\n");
fflush(debug);
#endif

        INVOKE(newOp, "NewSource",3L,0L,0L,0L, enString, enType, ENFromBool(FALSE));

#ifdef DEBUG
fprintf(debug,"Just returned from NewSource\n");
fflush(debug);
#endif

        SET_PROP_VALUE(newOp, "returnType", &TYPEBoolean,0L);

        Call_NewArgs(newOp,1);

#ifdef DEBUG
fprintf(debug,"Calling GetObjFromUID\n");
fflush(debug);
#endif

        predop = OPTYPE(GetObjFromUID(newOp.ptr, newOp.typeFlag));

#ifdef DEBUG
fprintf(debug,"Just returned from GetObjFromUID\n");
fflush(debug);
#endif

        GetUIDFromObj(newOp, &val1, &val2);

        /* We use this UID to generate an initialization statment */
        /* for the operation. */
        my_args[0] = initializations;
        my_args[1] = "     ";
        my_args[2] = short_routine_name;
```

```
        my_args[3] = " = GetObjFromUID( ";
        my_args[4] = itoa(val1);
        my_args[5] = ", ";
        my_args[6] = itoa(val2);
        /*
        strcat(initializations, "Squery.ptr, Squery.typeFlag");
        */
        my_args[7] = ")";
        my_args[8] = ";\n";

        initializations = Concat(9,my_args);

        return(short_routine_name);
}

/*****************************************************************/

/* gen_FuncOp_code builds a function which returns the object resulting when the code   */
/* in query_str is executed. The type of this result will become the type of            */
/* the function itself. "self_type" is the type of the first argument of the function   */
/* to be generated.*/

char *
gen_FuncOp_code(query_str, self_type)
char *query_str;
ENType self_type;
{
        FILE *fs;
        param_descr *param_list;
        char *FuncOp_name = malloc(128);
        char *short_FuncOp_name = malloc(128);
        char *GetArg_stmts;
        char *coll_decls;
        char *local_inits;
        char *AddArgs_stmt;
        /*
        char *includes;
        */
        char *system_arg = malloc(64);
        char *file_name;
        char *type_name;
        ENString enString;
        ENBytes args;
        ENOperationType newOp;
        int val1, val2, random;
        ENInteger seg;
        ENType  enType;
        short   pos;
        char            *my_args[100];

        random = rand();
        /*
        fs = fopen("output.c", "a+");
        */

        /* Start to generate the name of the function to be constructed. */
        /* All functions generated within a process must be uniquely named, */
        /* so we append a unique random number to the prefix "PredOp" */

        strcpy(short_FuncOp_name, "FuncOp");
        strcat(short_FuncOp_name, itoa(random) );
        file_name = malloc(strlen(gLoadDir) + strlen("/") +
                strlen(short_FuncOp_name) + strlen(".c") + 1);

        /* The portion of the name generated thus far will serve as the name */
```

```
        /* of the file containing the function */

        sprintf(file_name, "%s/%s.c",gLoadDir,short_FuncOp_name);
        fs = fopen(file_name, "w");

        fwrite(includes, 1, strlen(includes), fs);

/* Generate name and args of routine */

        /* The name portion is now prefixed with the type of the "self" argument */

        /** Is this necessary?
        if (!runtime_check)
        {
                enString = STRING( GET_PROP_VALUE(self_type,"name",0L) );
                strcpy(FuncOp_name, ENToString(enString));
        }
        else
        */
        {
                strcpy(FuncOp_name, "Object");
        }
        strcat(FuncOp_name, "_FuncOp");
        strcat(FuncOp_name, itoa(random) );

/* TYPE of query_str (or maybe just ENObject?) needed here */
        fwrite("ENObject", 1, strlen("ENObject"), fs);

        fwrite("\n", 1, 1, fs);
        fwrite(FuncOp_name, 1, strlen(FuncOp_name), fs);
        fwrite("(", 1, strlen("("), fs);

        param_list = StackFrame;
        do {
                param_list = param_list->prev;
                fwrite(param_list->name, 1, strlen(param_list->name), fs);
                fwrite(",", 1, strlen(","), fs);
        } while (!(param_list->first_in_frame));

        fwrite("args)", 1, strlen("args)"), fs);
        fwrite("\n", 1, 1, fs);

/* Generate arg declarations of routine */

        param_list = StackFrame;
        do {
                param_list = param_list->prev;
                /** Is this necessary?
                if (!runtime_check)
                {
                        type_name = ENToString(STRING(GET_PROP_VALUE(param_list->type, "name",0L)));
                        fwrite("EN", 1, 2, fs);
                        fwrite(type_name, 1, strlen(type_name), fs);
                        fwrite(" ", 1, 1, fs);
                }
                else
                */
                {
                        fwrite("ENObject ", 1, strlen("ENObject "), fs);
                }
                fwrite(param_list->name, 1, strlen(param_list->name), fs);
                fwrite(";", 1, strlen(";"), fs);
                fwrite("\n", 1, 1, fs);

        } while (!(param_list->first_in_frame));
```

```
        fwrite("ENObject args;", 1, strlen("ENObject args;"), fs);
        fwrite("\n", 1, 1, fs);

/* Generate local variable declarations -- this is for all */
/* variables defined in previous scopes */

        fwrite("{", 1, strlen("{"), fs);
        fwrite("\n", 1, 1, fs);

        if (param_list->prev)
        {
                pos = 0;
                param_list = param_list->prev;
                do
                {
                        /** Is this necessary?
                        if (!runtime_check)
                        {
                                type_name = ENToString(STRING(GET_PROP_VALUE(param_list->type, "name",0L)));
                                fwrite("EN", 1, 2, fs);
                                fwrite(type_name, 1, strlen(type_name), fs);
                                fwrite(" ", 1, 1, fs);
                        }
                        else
                        */
                        {
                                fwrite("     ENObject ", 1, strlen("     ENObject "), fs);
                        }

                        fwrite(param_list->name, 1, strlen(param_list->name), fs);
                        fwrite(";", 1, 1, fs);
                        fwrite("\n", 1, 1, fs);

                        /* Now that we've written out the local variable */
                        /* declarations, they must be initialized by */
                        /* being extracted from "args". So we generate */
                        /* the necessary ENGetArg() calls and save them */
                        /* to be written out later */

                        my_args[pos++] = "     ";
                        my_args[pos++] =  param_list->name;
                        my_args[pos++] =  " = ENGetArg(args,";
                        my_args[pos++] =  "\"";
                        my_args[pos++] =  param_list->name;
                        my_args[pos++] =  "\"";
                        my_args[pos++] = `");\n";

                        param_list = param_list->prev;

                } while (param_list);
        }

        local_inits = Concat(pos,my_args);

        /* Now we write out the collection declarations */
        pos = 0;
        write_coll_decls(&pos,my_args);
        coll_decls = Concat(pos,my_args);
        fwrite(coll_decls, 1, strlen(coll_decls), fs);
        fwrite("\n", 1, 1, fs);

        /* Write out declarations for operation-objects */

        fwrite(declarations, 1, strlen(declarations), fs);
```

```
        fwrite("\n\n", 1, 1, fs);

        /* NOW write out the ENGetArg() calls which extract local */
        /* variables from "args" */

        fwrite(local_inits, 1, strlen(local_inits), fs);

        /* make_GetArg() generates ENGetArg() calls which extract */
        /* collections from "args" */

        GetArg_stmts = make_GetArg();
        fwrite(GetArg_stmts, 1, strlen(GetArg_stmts), fs);
        fwrite("\n", 1, 1, fs);

        /* Now write out all initializations for the operation-objects */

        fwrite(initializations, 1, strlen(initializations), fs);
        fwrite("\n", 1, 1, fs);

/* Generate body of routine */

        AddArgs_stmt = make_AddArgs();
        fwrite(AddArgs_stmt, 1, strlen(AddArgs_stmt), fs);
        fwrite("\n", 1, 1, fs);

        fwrite("     return(", 1, strlen("     return("), fs);
        fwrite(query_str, 1, strlen(query_str), fs);
        fwrite(");", 1, strlen(");"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("}", 1, strlen("}"), fs);
        fwrite("\n", 1, 1, fs);
        fwrite("\n", 1, 1, fs);
        fclose(fs);

        /* We now have a complete function in a file. */
        /* It must be run through a special scanner and then compiled */

        /*
        strcpy(system_arg, "sh -x comp ");
        strcat(system_arg, short_FuncOp_name);
        system(system_arg);
        */

        /* The new function is added to the list of operation declarations */

        my_args[0] = declarations;
        my_args[1] = "     ENObject ";
        my_args[2] = short_FuncOp_name;
        my_args[3] = ";\n";
        declarations = Concat(4,my_args);

        /* Now we create an operation-object from the function, */
        /* and then add it to the database */

        enString = ENFromString(short_FuncOp_name);
        if (!runtime_check)
        {
                args = ENBuildArgList(3, "name", &enString, "refines", &TYPEOperation, "owningType", &self_type );
        }
        else
        {
                args = ENBuildArgList(3, "name", &enString, "refines", &TYPEOperation, "owningType", &TYPEObject );
        }
        seg = ENFromLong(TYPE_SEG);
        newOp = OPTYPE(INVOKE(TYPEOperationType, "CreateInstance",2L,0L,0L,0L, seg, args));
```

```
        enString = ENFromString(short_FuncOp_name);

        INVOKE(newOp, "NewSource",3L,0L,0L,0L,enString, enType, ENFromBool(FALSE));
        SET_PROP_VALUE(newOp, "returnType", &TYPEObject,0L);

        Call_NewArgs(newOp,1);

        predop = OPTYPE(GetObjFromUID(newOp.ptr, newOp.typeFlag));

        GetUIDFromObj(newOp, &val1, &val2);

        my_args[0] = initializations;
        my_args[1] = "     ";
        my_args[2] = short_FuncOp_name;
        my_args[3] = " = GetObjFromUID( ";
        my_args[4] = itoa(val1);
        my_args[5] = ", ";
        my_args[6] = itoa(val2);
        my_args[7] = ")";
        my_args[8] = ";\n";

        return(short_FuncOp_name);
}

/*****************************************************************************/

Call_NewArgs(self_arg,generated_code)
ENOperationType self_arg;
int generated_code;
{
        ENUIDBytes      argList;
        ENInteger       argFlag;
        ENObject        *list, *actual_list;
        int             Index = 0;
        int             numargs = 0;
        int             i = 0;
        ENType          type_obj;
        long            flag = (long)0;

#ifdef DEBUG
fprintf(debug,"Just entered Call_NewArgs\n");
fflush(debug);
#endif DEBUG

        if (!generated_code)
        {
                SYMBOL *place = symbol_table.next;
                char *type_dec = 0;
                char *end_of_type_str;
                int current_scope = place->symbol_scope;

                while (place->symbol_scope == current_scope)
                {
                        numargs++;
                        place = place->next;
                }

                list = (ENObject *) malloc(numargs*sizeof(ENObject));
                place = symbol_table.next;

                while (place->symbol_scope == current_scope)
                {
                        int is_ptr = 0;
#ifdef DEBUG
```

```
fprintf(debug,"Call_NewArgs: get type for %s\n",place->symbol_name);
fflush(debug);
#endif DEBUG
                        if (Is_Encore_Type(place->symbol_type))
                        {
                                bool    isPtr;
                                type_dec =
                                        GetBaseType(place->symbol_type,&isPtr);
                                type_obj = GetTypeObject(type_dec);
                                if (EN_OBJ_NEQ(type_obj,EN_NO_TYPE))
                                {
                                        list[Index] = OBJECT(type_obj);
                                        if (is_ptr)
                                        {
                                                SET_INVOKE_MASK(flag,Index);
                                        }

                                        Index++;

#ifdef DEBUG
fprintf(debug,"Call_NewArgs: %s added to list\n",type_dec);
fflush(debug);
#endif DEBUG
                                }
                                else
                                {
                                        fprintf(debug,"ERROR -- ENCORE type %s nonexistent\n", type_dec);
                                        fflush(debug);
                                }
                        }
                        else
                        {
                                fprintf(debug,"ERROR -- %s not an ENCORE type\n", type_dec);
                                fflush(debug);
                        }
                        place = place->next;
                }
        }
        else
        {
                param_descr *param_list;
                param_list = StackFrame;

                do {
                        param_list = param_list->prev;
                        numargs++;
                } while (!param_list->first_in_frame);

                list = (ENObject *) malloc((numargs+1)*sizeof(ENObject));
                param_list = StackFrame;

                do {
                        param_list = param_list->prev;
#ifdef DEBUG
fprintf(debug,"Call_NewArgs: get type for %s\n",param_list->name);
fflush(debug);
#endif DEBUG
                        list[Index] = OBJECT(param_list->type);
                        Index++;

#ifdef DEBUG
fprintf(debug,"Call_NewArgs: %s added to list\n", ENToString(STRING(GET_PROP_VALUE(param_list->type, "name",0L))) );
fflush(debug);
#endif DEBUG
                } while (!param_list->first_in_frame);
```

```
#ifdef DEBUG
fprintf(debug,"Call_NewArgs: Bytes added to list\n");
fflush(debug);
#endif DEBUG
                list[Index] = OBJECT(TYPEBytes);
        }

        Index -=2;      /* Ignore "self" arg */
        actual_list = (ENObject *) malloc((Index + 1)*sizeof(ENObject));
        for (i = 0; i <= Index; i++)    /* reverse order of arg list */
        {
                actual_list[i] = list[Index - i];
        }

        argFlag = INT(ENFromLong(flag));
        argList = ENFromUIDBytes(list, Index + 1);
        INVOKE(self_arg, "NewArguments", 2L, 0L, 0L, 0L, argList, argFlag);
#ifdef DEBUG
fprintf(debug,"Exiting Call_NewArgs\n");
fflush(debug);
#endif DEBUG
}

/*****************************************************************************
 *
 * Function: Add_To_List
 *
 * Arguments: list - a list to add to
 *            entry - an entry to add to the list
 *
 * Returns: a pointer to the updated list
 *
 * Description:
 *
 *    This routine updates the given list of names with a new entry by
 * placing the entry at the beginning of the list.
 *
 *****************************************************************************/
struct symbol_table_entry *Add_To_List (list, entry)
struct symbol_table_entry *list, *entry;
{

  if (entry) {
    entry->next = list;
    return entry;
  } else
    return list;

}

/*****************************************************************************
 *
 * Routine: Add_Symbol_Names
 *
 * Arguments: type - type string for names being added to the table
 *            list - a list of names to add to the symbol table
 *            scope - GLOBAL or current local scope
 *
 * Description:
 *
 *    This routine adds a list of names of the form:
 *
 *         string1,string2,...,stringN
 *
```

```
 * to the current symbol table.
 *
 ************************************************************************/
Add_Symbol_Names (type,list,scope)
char *type;
SYMBOL *list;
int scope;
{
SYMBOL *place = list, *prev = list;

#ifdef DEBUG
fprintf(debug,"Add_Symbol_Names: type = '%s'\n", type);
fflush(debug);
#endif DEBUG

  /* update entries in list of names being added with supplied information */
  while (place) {
    place->symbol_scope = scope;

    /* if there is part of a type string already, then concat */
    if (place->symbol_type) {
      arg_list[0] = type;
      arg_list[1] = place->symbol_type;
      place->symbol_type = Concat_With_Spaces (2,arg_list);

#ifdef DEBUG
fprintf(debug,"Add_Symbol_Names: place->symbol_type = '%s'\n", place->symbol_type);
fflush(debug);
#endif DEBUG

    } else
      place->symbol_type = strdup(type);

    prev = place;
    place = place->next;
  }

  if (prev) {
    /* put list at the beginning of the symbol table */
    prev->next = symbol_table.next;
    symbol_table.next = list;
  }

}

/************************************************************************
 *
 * Routine: Add_Typedef_Names
 *
 * Arguments: list - a list of names to add to the typedef_names list
 *
 * Description:
 *
 *    This routine adds a list of names to the global list of typedef names.
 *
 ************************************************************************/
Add_Typedef_Names (list)
SYMBOL *list;
{
SYMBOL *place = list;

  while (place->next)
    place = place->next;

  place->next = typedef_names.next;
```

```
  typedef_names.next = list;

}

CODE *Allocate_Code_Block ()
{
CODE *new;

  new = (CODE *)malloc (sizeof (CODE));

  if (new == 0) {
    printf ("Error-Allocate_Code_Block-malloc returned 0\n");
    return 0;
  }

  new->changed = FALSE;
  new->use_l_invoke = FALSE;
  new->old_code = 0;
  new->new_code = 0;
  new->type = 0;
  new->actual_type = 0;
  new->en_type = EN_NO_TYPE;
  new->num_exprs = 0;
  new->next = 0;

  return new;
}

SYMBOL *Build_Symbol (string)
char *string;
{
SYMBOL *new;

  new = (SYMBOL *)malloc (sizeof (SYMBOL));

  if (new == 0) {
    printf ("Error-Build_Symbol-malloc returned 0\n");
    return 0;
  }

  new->symbol_name = strdup(string);
  new->symbol_type = 0;
  new->symbol_scope = 0;
  new->code =malloc(1);
  new->next = 0;

  return (new);

}

Call_Query_Parser ()
{

  printf ("Call Query Parser...\n");

}

char *Coerce_Type (target_type,source_type)
char *target_type,*source_type;
{
char *result = 0;

  /* These are the allowable type conversions that are processed by
     the parser */
```

```
/*
  if (!strcmp (source_type,"ENObject"))
        {
        if (!strcmp (target_type,"long") || !strcmp (target_type,"int"))
                result = "ENToLong";

        else if (!strcmp (target_type,"float"))
                result = "ENToFloat";

        else if (!strcmp (target_type,"char *"))
                result = "ENToString";
        }
*/

  if (!strcmp (source_type,"ENInteger") && (!strcmp (target_type,"long") ||
                !strcmp (target_type,"int")))
        result = "ENToLong";

  else if (!strcmp (source_type,"ENFloat") && !strcmp (target_type,"float"))
        result = "ENToFloat";

  else if (!strcmp (source_type,"ENString") && !strcmp (target_type,"char *"))
        result = "ENToString";

  else if ((!strcmp (source_type,"long") || !strcmp (source_type,"char") ||
        !strcmp (source_type,"int")) &&
        (!strcmp (target_type,"ENInteger") || !strcmp(target_type,"ENObject")))
                result = "ENFromLong";

  else if (!strcmp (source_type,"float") && (!strcmp (target_type,"ENFloat") ||
                        !strcmp(target_type,"ENObject")))
        result = "ENFromFloat";

  else if (!strcmp (source_type,"char *") &&(!strcmp (target_type,"ENString") ||
                !strcmp(target_type,"ENObject")))
        result = "ENFromString";

  return(result);
}

char *Concat (arg_count, args)
int arg_count;
char *args[];
{
int size = 0, i;
char *temp;

  for (i=0; i<arg_count; i++)
    if (args[i])
      size += strlen (args[i]);

  temp = malloc (size + 1);

  if (!temp) {
    printf ("Error-Concat-Malloc returned 0\n");
    return(temp);
  }

  temp[0] = '\0';

  for (i = 0; i < arg_count; i++)
    if (args[i])
      strcat (temp,args[i]);

  return(temp);
```

```
}

char *Concat_With_Spaces (arg_count, args)
int arg_count;
char *args[];
{
int size = 0, i;
char *temp;

  for (i=0; i<arg_count; i++)
    if (args[i])
      size += strlen (args[i]);

  temp = malloc (size + arg_count + 1);

  if (!temp) {
    printf ("Error-Concat_With_Spaces-Malloc returned 0\n");
    return(temp);
  }

  temp[0] = '\0';

  for (i = 0; i < arg_count; i++) {
    if (args[i]) {
      strcat (temp,args[i]);
      strcat (temp," ");
    }
  }

  /* trim off the last space added */
  temp[strlen(temp)-1] = '\0';

  return(temp);
}

Display_Symbol_Table ()
{
struct symbol_table_entry *temp = symbol_table.next;

  while (temp) {

    printf ("%10s\t%10s\t%d\n",temp->symbol_type,temp->symbol_name,temp->symbol_scope);

    temp = temp->next;
  }

}

Enter_Scope ()
{

  current_scope++;

}

Exit_Scope ()
{

 Remove_Symbols (current_scope--);

}

int Is_Encore_Type (type)
char *type;
{
```

```
  /* REPLACE - this should really ask Encore itself if the given
     type belongs to it */

  if (type &&
      (Substring(type,"EN") == 0) &&
      (Substring(type,"EN_") != 0) &&
      (Substring(type,"ENTo") != 0) &&
      (Substring(type,"ENFrom") != 0) &&
      (Substring(type,"ENAddArg") != 0) &&
      (Substring(type,"ENBuildArg") != 0) &&
      (Substring(type,"ENGetArg") != 0) &&
      (Substring(type,"ENInit") != 0) &&
      (Substring(type,"ENBegin") != 0) &&
      (Substring(type,"ENNo_Type_Found") != 0) &&
      (Substring(type,"ENCreate") != 0) )
    return(TRUE);
  else
    return(FALSE);

/*** Should this go here?
  if (GetTypeObject(type))
  {
        return(TRUE);
  }
  else
  {
        return(FALSE);
  }
*/

}

int Is_Existing_Encore_Type (type)
char *type;
{

  /* REPLACE - this should really ask Encore itself if the given
     type belongs to it */

  if (type && (Substring(type,"ENObject") == 0) )
/*
      (Substring(type,"ENOperationType") == 0) ||
      (Substring(type,"ENObject") == 0) ||
      (Substring(type,"ENBoolean") == 0) ||
      (Substring(type,"ENType") == 0) ||
      (Substring(type,"ENInteger") == 0) ||
      (Substring(type,"ENBytes") == 0) ||
      (Substring(type,"ENOpDefList") == 0) ||
      (Substring(type,"ENString") == 0) ||
      (Substring(type,"ENUIDBytes") == 0) ||
      (Substring(type,"ENTmpObjHandle") == 0) ||
      (Substring(type,"ENEmbObjectTable") == 0) ||
      (Substring(type,"ENEmbObjHandle") == 0) ||
      (Substring(type,"ENTransStack") == 0) ||
      (Substring(type,"ENTransaction") == 0) ||
      (Substring(type,"ENROProperty") == 0) ||
      (Substring(type,"ENPropDefList") == 0) ||
      (Substring(type,"ENBigReal") == 0) ||
      (Substring(type,"ENReal") == 0) ||
      (Substring(type,"ENFunction") == 0) ||
      (Substring(type,"ENInternalString") == 0) ||
      (Substring(type,"ENInternalBytes") == 0) ||
      (Substring(type,"ENInternalUIDBytes") == 0) ||
      (Substring(type,"ENPropertyType") == 0))
```

```
*/
    return(TRUE);
  else
    return(FALSE);

/*** Should this go here?
  if (GetTypeObject(type))
  {
       return(TRUE);
  }
  else
  {
       return(FALSE);
  }
*/

}

int Is_Global_Encore_Object (name)
char *name;
{

  /* REPLACE - this should really ask Encore itself if the given
     type belongs to it */

  if (Substring(name,"EN_") == 0)
    return(TRUE);
  else
    return(FALSE);

}

/*************************************************************************
 *
 * Function: Look_In_Typdef_Names
 *
 * Arguments: name - a name to look up
 *
 * Returns: TRUE if name is in the typedef_names table or FALSE
 *
 *************************************************************************/
int Look_In_Typedef_Names (name)
char *name;
{
SYMBOL *place = typedef_names.next;
int result = FALSE;

  while (place)
    if (!strcmp (name,place->symbol_name)) {
      result = TRUE;
      break;
    } else
      place = place->next;

#ifdef DEBUG
fprintf(debug,"Look_In_Typedef_Names(%s) returns %d\n",name,result);
fflush(debug);
#endif DEBUG

  return(result);
}

char *Lookup_Encore_Property_Type (type,property)
char *type,*property;
{
```

```
/* REPLACE with Type_TypeCheckGetPropValue */

char *result = 0;

  if (!strcmp (property,"numPositions") ||
      !strcmp (property,"numMembers") ||
      !strcmp (property,"nextFreePos"))
    result = "ENInteger";

  if (!strcmp (property,"operationName") ||
      !strcmp (property,"name"))
    result = "ENString";

  return(result);

}

char *Lookup_Type (name)
char *name;
{
SYMBOL *place = symbol_table.next;
char *result = 0;

  while (place)
    if (!strcmp (name,place->symbol_name)) {
      result = place->symbol_type;
#ifdef DEBUG
fprintf(debug,"Lookup_Type: type of '%s' is '%s'\n", name, result);
fflush(debug);
#endif DEBUG
      break;
    } else
      place = place->next;

  if (!result)
    return ("No_Type_Found");
  else
    return(result);

}

Remove_Symbols (scope)
int scope;
{
struct symbol_table_entry *temp = symbol_table.next, *prev = &symbol_table;

  /* look through the symbol table for all of the names in the current
     scope and remove them */
  while (temp) {

    if (temp->symbol_scope == scope) {

      prev->next = temp->next;

      free (temp->symbol_name);
      free (temp->code);
      free (temp->symbol_type);
      free (temp);

      temp = prev->next;

    } else {

      prev = temp;
```

```
      temp = temp->next;
    }
  }

}

char *strdup(s)
char *s;
{
char *new;

  new = malloc (strlen(s)+1);
  if (!new) {
    printf ("Error-My_strdup-malloc returned 0\n");
    return(0);
  }
  strcpy (new,s);
  return(new);
}

int Substring (s1,s2)
char *s1,*s2;
{
int i,j,k;

  for (i=0; s1[i] != '\0'; i++) {
    for (j=i,k=0; s2[k] != '\0' && s1[j] == s2[k]; j++, k++)
      ;
    if (s2[k] == '\0')
      return(i);
  }
  return (-1);
}

Update_Type (entry,type)
SYMBOL *entry;
char *type;
{

  if (!strcmp (type,"(") ||
      !strcmp (type,"*")) {
    arg_list [0] = type;
    arg_list [1] = entry->symbol_type;
    entry->symbol_type = Concat (2,arg_list);
  } else {
    arg_list [0] = entry->symbol_type;
    arg_list [1] = type;
    entry->symbol_type = Concat (2,arg_list);
  }

}

Write_Out_Query (query)
char *query;
{
FILE *out;

  out = fopen ("./Query","w");

  if (out == NULL)
    printf ("Error-Write_Out_Query-fopen returned NULL\n");
  else {

    fprintf (out,"%s;\0\n",query);
    fclose (out);
```

```
  }

}

Write_Out_Symbol_Table ()
{
FILE *out;
SYMBOL *place = symbol_table.next;

  out = fopen ("./Symbol_Table","w");

  if (out == NULL)
    printf ("Error-Write_Out_Symbol_Table-fopen returned Null\n");
  else {

    while (place) {

      fprintf (out,"%s\0%s\0%d\n",place->symbol_type,place->symbol_name,
               place->symbol_scope);

      place = place->next;
    }
    fclose (out);
  }
}

/*************************************************************************
 *
 * Routine: yyerror
 *
 * Description:
 *
 *    This routine is called from the parser when an error is encountered
 * in the input stream.
 *
 *************************************************************************/
yyerror (s)
char *s;
{

  fprintf (stderr,"%s on line %d\n", s, line_counter);
  fflush(stderr);

}

void
init_term()
{
        term_stack.top = -1;
}

void
push_term(item)
char    *item;
{
        if (term_stack.top == STACK_SIZE - 1)
                {
                fprintf(stderr,"Too many terms\n");
                }
        else
                {
                term_stack.top++;
                term_stack.body[term_stack.top] = item;
                }
}
```

```
char *
pop_term()
{
        if (term_stack.top != -1)
                term_stack.top--;
        if (term_stack.top >= 0)
                return(term_stack.body[term_stack.top+1]);
        return(0);
}

bool
is_empty_term()
{
        if (term_stack.top == -1 || *(term_stack.body[term_stack.top]) == '$')
                return(TRUE);
        return(FALSE);
}

print_term()
{
        int     i;

        for (i=0;i<term_stack.top;i++)
                {
                if (*(term_stack.body[i]) == '$')
                        printf("$\n");
                else if (UID_TYPE(OBJECT(*(term_stack.body[i]))) == T_OBJECT)
                        printf("Type name = %s\n",
                           GET_PROP_VALUE(*(term_stack.body[i]),"name",0L));
                else
                        printf("string name = %s\n",term_stack.body[1]);
                }
}

char *
GetBaseType(cType,isPtr)
char    *cType;
bool    *isPtr;
{
        char    *end_of_type;
        char    *start_of_type;
        char    *type_dec;

        type_dec = strdup(cType);
        start_of_type = type_dec;

        if (Substring(type_dec,"*") == 0 || Substring(type_dec,"[") == 0)
                *isPtr = TRUE;
        else
                *isPtr = FALSE;

        end_of_type = &type_dec[strlen(type_dec)-1];
        while ((*end_of_type < 'a' || *end_of_type > 'z') &&
                (*end_of_type < 'A' || *end_of_type > 'Z') &&
                (*end_of_type < '0' || *end_of_type > '9') &&
                *end_of_type != '_')
                end_of_type--;

        end_of_type[1] = '\0';

        if (!strncmp("extern",start_of_type,6))
        {
                start_of_type = strchr(start_of_type, ' ');
                start_of_type++;
```

```
        }
        if (!strncmp(start_of_type,"EN",2))
                start_of_type+=2;
#ifdef DEBUG
fprintf(debug,"GetBaseType: %s type is %s\n",cType,start_of_type);
fflush(debug);
#endif DEBUG
        return(start_of_type);
}

init_parser()
{
        init_term();
        yy_file_desc = fopen("/tmp/temp.c", "w");

        /* per function basis */
        fwd_decl = "";
        query_functions = "";

        typedef_names.symbol_name = 0;
        typedef_names.symbol_type = 0;
        typedef_names.code = 0;
        typedef_names.symbol_scope = 0;
        typedef_names.next = 0;

        symbol_table.symbol_name = 0;
        symbol_table.symbol_type = 0;
        symbol_table.code = 0;
        symbol_table.symbol_scope = 0;
        symbol_table.next = 0;

        includes = "#include    <stdio.h>\n#include     \"encore.h\"\n";

        current_scope = 0;

        callerType = EN_NO_TYPE;

}

end_parser()
{
        fclose(yy_file_desc);
        system("cb /tmp/temp.c > /tmp/test.c");
        system("rm /tmp/temp.c");
}

init_query()
{
        declarations = "";
        initializations = "";
        StackFrame = 0;
        Coll_List = 0;
}
```

```
%{

#define NUMBER_OF_KEYWORDS 40

/* define a table to be used in looking up keywords */
struct keyword_table_entry {
  char *keyword;
  int token_type;
  } keyword_table [NUMBER_OF_KEYWORDS*2] = {  /* initialize table */
        "AUTO",sym_key_auto,
        "BREAK",sym_key_break,
        "CASE",sym_key_case,
        "CHAR",sym_key_char,
        "CONTINUE",sym_key_continue,
        "DEFAULT",sym_key_default,
        "DO",sym_key_do,
        "DOUBLE",sym_key_double,
        "ELSE",sym_key_else,
        "ENTRY",sym_key_entry,
        "EXTERN",sym_key_extern,
"FALSE",FALSE_TOKEN,
"FLATTEN",FLATTEN,
        "FLOAT",sym_key_float,
        "FOR",sym_key_for,
        "GOTO",sym_key_goto,
        "IF",sym_key_if,
"IMAGE",IMAGE,
        "INT",sym_key_int,
        "LONG",sym_key_long,
"MEMBEROF",IN,
"NEST",NEST,
"NOT",NOT,
"OJOIN",OJOIN,
"PROJECT",PROJECT,
        "REGISTER",sym_key_register,
        "RETURN",sym_key_return,
"SELECT",SELECT,
        "SHORT",sym_key_short,
        "SIZEOF",sym_key_sizeof,
        "STATIC",sym_key_static,
        "STRUCT",sym_key_struct,
"SUBSETOF",SUBSETOF,
        "SWITCH",sym_key_switch,
"TRUE", TRUE_TOKEN,
        "TYPEDEF",sym_key_typedef,
        "UNION",sym_key_struct,
"UNNEST",UNNEST,
        "UNSIGNED",sym_key_unsigned,
        "WHILE",sym_key_while,
        "auto",sym_key_auto,
        "break",sym_key_break,
        "case",sym_key_case,
        "char",sym_key_char,
        "continue",sym_key_continue,
        "default",sym_key_default,
        "do",sym_key_do,
        "double",sym_key_double,
        "else",sym_key_else,
        "entry",sym_key_entry,
        "extern",sym_key_extern,
"false",FALSE_TOKEN,
"flatten",FLATTEN,
        "float",sym_key_float,
        "for",sym_key_for,
        "goto",sym_key_goto,
```

```
        "if",sym_key_if,
 "image",IMAGE,
        "int",sym_key_int,
        "long",sym_key_long,
 "memberof",IN,
 "nest",NEST,
 "not",NOT,
 "ojoin",OJOIN,
 "project",PROJECT,
        "register",sym_key_register,
        "return",sym_key_return,
 "select",SELECT,
        "short",sym_key_short,
        "sizeof",sym_key_sizeof,
        "static",sym_key_static,
        "struct",sym_key_struct,
 "subsetof",SUBSETOF,
        "switch",sym_key_switch,
 "true",TRUE_TOKEN,
        "typedef",sym_key_typedef,
        "union",sym_key_struct,
 "unnest",UNNEST,
        "unsigned",sym_key_unsigned,
        "while",sym_key_while};


/* counter to keep track of lines of input */
int line_counter = 1;

char *strdup ();

%}

DIGIT      [0-9]
LETTER     [A-Za-z_]
EXP        [E][+-]?{DIGIT}+

%%


{LETTER}({LETTER}|{DIGIT})* {

  /* identifier recognized */
  int result;

  /* remember the string */
  yylval.sval = strdup (yytext);

  /* check to see if the token is actually a keyword */
  if ((result = Look_In_Keyword_Table (yytext)) != FALSE)
  {
    return result;
  }
  else if ((Is_Encore_Type (yytext) != FALSE) || (Look_In_Typedef_Names (yytext) != FALSE))
  {
    return sym_typedef_name;
  }
/****** Page's *******/
/*
  else if (Is_Encore_Type(Lookup_Type(yytext)) == TRUE)
*/
  else
  {
        char *symtab_type_field;
        char *full_type_name = 0;
```

```
        char *partial_type_name;
        bool isPtr;

 /* This takes a symbol table type field like */
 /* "extern ENObject foo" and turns it into "ENObject" */

        if (Substring(Lookup_Type(yytext),"No_Type_Found") != 0)
        {
                symtab_type_field = strdup(Lookup_Type(yytext));
                partial_type_name  = GetBaseType(symtab_type_field,&isPtr);
                full_type_name = malloc(strlen(partial_type_name) + 3);
                strcpy(full_type_name,"EN");
                strcat(full_type_name,partial_type_name);
 #ifdef DEBUG
 fprintf(debug,"full name of type of %s is %s\n",yytext, full_type_name);
 fflush(debug);
 #endif DEBUG
        }

 /* Now we properly test if an identifier is of an ENCORE type */

        if (full_type_name && (Is_Encore_Type(full_type_name) == TRUE))
        {
 #ifdef DEBUG
 fprintf(debug,"%s is an en_identifier\n",yytext);
 fflush(debug);
 #endif DEBUG
                return en_identifier;
        }
        else
        {
                return sym_identifier;
        }
  }
 /****** Page's *******/
 }

 {DIGIT}+("L") |
 {DIGIT}+ |
 ("0x"){DIGIT}+ |
 ("0X"){DIGIT}+ {

   /* integer constant recognized */

   /* remember the string */
   yylval.sval = strdup (yytext);

   return sym_constant;
 }

 "'"[^.]"'" |
 "'\\"([ntbrf0]|({DIGIT}{DIGIT}{DIGIT}))"'" {

   /* character constant recognized */

   /* remember the string */
   yylval.sval = strdup (yytext);

   return sym_constant;

 }

 {DIGIT}+"." |
 "."{DIGIT}+ |
 {DIGIT}+"."{DIGIT}+ |
```

```
"."{DIGIT}+{EXP} |
{DIGIT}+"."{EXP} |
{DIGIT}+{EXP} |
{DIGIT}+"."{DIGIT}+{EXP} {
  /* float constant recognized */

  /* remember the string */
  yylval.sval = strdup (yytext);

  return sym_constant;
}

\"((\\\")|[^\"])*\" {
  /* string constant recognized */

  /* remember the string */
  yylval.sval = strdup (yytext);

  return sym_string;
}

"%{" {
  /* query delimiter recognized */

  yylval.sval = strdup (yytext);

  return sym_query_start;
}

"%}" {
  /* query delimiter recognized */

  yylval.sval = strdup (yytext);

  return sym_query_end;
}

"$"       { yylval.sval = strdup (yytext); return LAMBDA; }

"+"       { yylval.sval = strdup (yytext); return sym_op_plus; }

"-"       { yylval.sval = strdup (yytext); return sym_op_minus; }

"*"       { yylval.sval = strdup (yytext); return sym_op_mult; }

"/"       { yylval.sval = strdup (yytext); return sym_op_div; }

"%"       { yylval.sval = strdup (yytext); return sym_op_mod; }

"<<"      { yylval.sval = strdup (yytext); return sym_op_shift; }

">>"      { yylval.sval = strdup (yytext); return sym_op_shift; }

"=="      { yylval.sval = strdup (yytext); return sym_op_eq; }

"!="      { yylval.sval = strdup (yytext); return sym_op_eq; }

"<"       { yylval.sval = strdup (yytext); return sym_op_rel; }

"<="      { yylval.sval = strdup (yytext); return sym_op_rel; }

">"       { yylval.sval = strdup (yytext); return sym_op_rel; }

">="      { yylval.sval = strdup (yytext); return sym_op_rel; }
```

```
"&&"      { yylval.sval = strdup (yytext); return sym_op_and; }
"||"      { yylval.sval = strdup (yytext); return sym_op_or; }
"&"       { yylval.sval = strdup (yytext); return sym_op_bit_and; }
"|"       { yylval.sval = strdup (yytext); return sym_op_bit_or; }
"^"       { yylval.sval = strdup (yytext); return sym_op_bit_xor; }
"!"       { yylval.sval = strdup (yytext); return sym_op_unary; }
"~"       { yylval.sval = strdup (yytext); return sym_op_unary; }
"++"      { yylval.sval = strdup (yytext); return sym_op_inc; }
"--"      { yylval.sval = strdup (yytext); return sym_op_inc; }
"="       { yylval.sval = strdup (yytext); return sym_asgn; }
"+="      { yylval.sval = strdup (yytext); return sym_op_asgn; }
"-="      { yylval.sval = strdup (yytext); return sym_op_asgn; }
"*="      { yylval.sval = strdup (yytext); return sym_op_asgn; }
"/="      { yylval.sval = strdup (yytext); return sym_op_asgn; }
"%="      { yylval.sval = strdup (yytext); return sym_op_asgn; }
">>="     { yylval.sval = strdup (yytext); return sym_op_asgn; }
"<<="     { yylval.sval = strdup (yytext); return sym_op_asgn; }
"&="      { yylval.sval = strdup (yytext); return sym_op_asgn; }
"|="      { yylval.sval = strdup (yytext); return sym_op_asgn; }
"^="      { yylval.sval = strdup (yytext); return sym_op_asgn; }
","       { yylval.sval = strdup (yytext); return sym_comma; }
"."       { yylval.sval = strdup (yytext); return sym_period; }
"->"      { yylval.sval = strdup (yytext); return sym_arrow; }
"?"       { yylval.sval = strdup (yytext); return sym_question; }
";"       { yylval.sval = strdup (yytext); return sym_semi; }
":"       { yylval.sval = strdup (yytext); return sym_colon; }
"("       { yylval.sval = strdup (yytext); return sym_l_parn; }
")"       { yylval.sval = strdup (yytext); return sym_r_parn; }
"["       { yylval.sval = strdup (yytext); return sym_l_sbracket; }
"]"       { yylval.sval = strdup (yytext); return sym_r_sbracket; }
"{"       { yylval.sval = strdup (yytext); return sym_l_brace; }
"}"       { yylval.sval = strdup (yytext); return sym_r_brace; }
"@"       { yylval.sval = strdup (yytext); return sym_at; }
```

```
"`"       { yylval.sval = strdup (yytext); return sym_tick; }

"#"       { yylval.sval = strdup (yytext); return sym_pound; }

"/*"(("*"[^"/"])|[^"*"])*"*/" /* comment found, ignore */;

[ \t\r\014]+  /* whitespace, ignore */;

[\n]       line_counter++;

.          printf ("Error found by lexer on line %d\n",line_counter);

%%

/*************************************************************************
 *
 * Function: Look_In_Keyword_Table
 *
 * Inputs: s - string to look for in keyword table
 *
 * Outputs: token_type of keyword if s found in table
 *          FALSE otherwise
 *
 * Description:
 *
 *    This function looks for s in the keyword_table. If s is found,
 * the function returns the token_type of the keyword that s matched
 * i.e. sym_key_xxxx.  If s is not found, then FALSE is returned. The
 * function uses a binary search.
 *
 *************************************************************************/
int Look_In_Keyword_Table (s)
char *s;
{
int mid, low, high, done = FALSE, result = FALSE, test;

  /* initialize array bound variables */
  low = 0;
  high = NUMBER_OF_KEYWORDS*2-1;

  while (done != TRUE) {

    /* if this test passes, then the binary search failed */
    if (low > high)
      done = TRUE;

    else {
      /* select the midpoint in the array */
      mid = (low+high)/2;

      /* compare the keyword in the table with s */
      test = strcmp (keyword_table[mid].keyword,s);

      /* if the strings are equal, then we're done */
      if (!test) {
        result = keyword_table[mid].token_type;
        done = TRUE;
      }
      else {
        /* if the keyword tested was greater, then search the bottom half */
        if (test > 0)
          high = mid - 1;
        else
          /* search the top half */
```

```
            low = mid + 1;
        }  /* else */
    }  /* else */
}  /* while */

return result;

}
```