NewGen: A Language-Independent Program Generator
Report ENSMP/CRI/A-191

Pierre Jouvelot
Rémi Triolet

Jul. 1989, Revised Dec. 90

Abstract

NewGen is a software engineering tool that helps programmers define and implement sophisticated data types. Data *domains* are defined with a high level specification language, called *DDL*, that allows user defined domains to be built over *basic* domains with operators like Cartesian product, union, list or array.

NewGen analyzes a set of specifications and produces a collection of macros, functions, predicates, and pre-defined constants that enable programs to create, initialize, access, update, and delete objects of these types as if the programming language had been especially tailored to manipulate *these* data types.

NewGen promotes the ideas of *functional abstraction* and *evolution process* as ways to ease the prototyping, development and evolution of applications that use such multi-language program generators. Applications that are developed with NewGen data types can share complex data structures while being written partly in C and partly in CommonLISP, the two currently supported programming languages.

NewGen ideas have been validated in practice; this tool is heavily used for two projects that focus on the development of state-of-the-art parallelization strategies for Fortran programs.

Keywords: program generation, C, Common Lisp, data type, type checking, prototyping, language extension.

1 Presentation

The design and production of large software products require the definition and implementation of application specific data structures. NewGen is a software engineering tool that helps in this process.

NewGen provides a high level specification language with which data types can be defined. A user data type is built over basic data types (integer, string, float, ...) and other user data types defined in the same set of specifications or imported from another one. Operators are provided to elaborate complex data types, e.g. cartesian product, union, list, set ...

From a set of specifications, NewGen produces a library of functions and macros to create, initialize, access, update, delete and transfer objects of these types. These functions and macros are written in one of the supported programming languages (C [KR78] and CommonLISP [S84] at the time being) and may be used within a program, as if the language had been tailored to the user's needs.

In the remainder of this paper, we will discuss the related work (section 2), present the major aspects of NewGen (section 3), give a precise definition of the specification language (section 4), define the set of functions automatically provided by NewGen from a specification file (section 5), detail a complete example that shows the power of our tool (section 6) and conclude.

2 Related Work

The very idea of NewGen, i.e. automatic generation of code from high-level specifications, is hardly new. There are numerous commonly used "program generators" in the business market; NewGen integrates most of their characteristics. Moreover, NewGen provides a cleaner and sounder interface to the programmer, without giving up on efficient implementations.

NewGen combines ideas that were previously introduced in two similar tools: GenPgm, developed in 1986 by Rémi Triolet at Ecole des Mines de Paris, and MetaGen, developed at the same time by Pierre Jouvelot, at the Bull Corporate Research Center. The main purpose of MetaGen was to provide the user with a better programming language and a safer execution environment, while the goal of GenPgm was to enable independent programs to exchange complex data through files or streams.

NewGen is closely related to the IDL project [S89] which was developed around the same time. IDL is a program development tool [L87] that allows the specification of abstract data structures and automatically generates appropriate language-dependent (currently, C and Pascal are supported) data types. Accesses to values created with these facilities require an unnecessary knowledge of the internal implementation of these data types. NewGen alleviates this shortcoming by using functional abstraction (see below). IDL does not provide ways to free storage, thus relying on a garbage collector. To have a more efficient implementation, NewGen requires the programmer to explicitly free heap

storage. Nonetheless, to help the programmer avoid as many dangling pointers as possible, this is done at a high level, with sharing within values being automatically managed by Newgen. Note that this explicit memory management scheme partly motivated the introduction of tabulated domains (see below).

Stub generators generate code that perform data transfers via remote procedure calls (RPC) [BN83]. The description of the types to be transferred, even though they are described in a C-like language such as XDR [S87], are rather low level and restrictive. NewGen provides such a facility by allowing any data structure to be transferred over files, pipes or network sockets; sharing is preserved so that isomorphic copies are obtained on the receiving side.

The idea of transferring large chunks of data structures is also present in languages that support persistent objects [CM85], although they are often associated to types and not to values. In NewGen, the programmer is responsible for performing input/output operations (as in some Lisp systems that support this facility via `fasl` formats) on stable storage. The format is language-independent, contrarily to Lisp systems; data can be written from a C program and read in a CommonLISP process.

Object-oriented database systems such as O2 [A89], ORION [KBCGW88] or VBASE [AH87] propose facilities that extend the ones offered by NewGen. In these systems, database manipulation and query languages can be embeddeded into classical programming languages (e.g. C for O2 or CommonLISP for ORION), but they generally require a special preprocessor to manage the object-oriented nature of the language. NewGen only uses the standard Unix C preprocessor `cpp`, thus promoting portability, speeding compilation time and easing debugging (since a new debugger isn't required). Persistency is a by-product of the database layer; each object is persistent. This approach is too costly for the kind of problems NewGen is targeted to, so stable storage accesses in NewGen are specified by the programmer. These database systems are also biased towards interactive usage for queries; NewGen is centered around program development.

3 The Major Aspects of NewGen

NewGen promotes the ideas of functional abstraction, smoothes the evolution of software and provides a safer programming environment.

NewGen Supports Functional Abstraction

NewGen adopts a different philosophy than IDL: *functional abstraction*. From a data type specification, NewGen generates functions to create, manipulate, update and destroy values. This allows the programmer to think in terms of high-level concepts instead of the internal details of the implementation.

Since NewGen is based on the notion of function, which is ubiquitous in programming languages, the style that characterizes programs written with NewGen is the same in the different languages supported by NewGen.

Finally, extensions to a given program are easier to introduce. Since NewGen specifications generate functions, they can, if necessary, be manually tuned or redefined to adapt for changes in the specifications. For instance, some specification could be modified, in an upward-compatible way, by adding more information to the definition of a given domain or a given access function could be required to perform some profiling to discover code hot spots. In this case, programs that have been written in accordance to the old specifications can still be used (after a recompilation); the NewGen-generated functions hide the fact that new information have been introduced.

Data Transfer Eases Software Evolution

The ability to exchange complex data through files or pipe streams is useful when developing large multi-pass softwares such as compilers, where each pass takes one or several complex data structures as input, performs some calculation and finally produces another data structure.

Since large programs are difficult to develop, debug and maintain, it is preferable to write each pass as a separate program, but in this case, input and output data structures must be read and written on files or streams. This is a difficult problem when data structures are made of dynamically allocated memory cells, organized within list, trees, cyclic graphs or other pointer based recursive structures.

This problem is completely solved if the objects that must be read or written belong to a data type designed with NewGen. The write function produced by NewGen is able to map an object of any type on a disk file and the related read function is able to reproduce in memory a similar object, even if memory cells were shared in the original object (circular list, graph, ...). This facility is already provided in CommonLISP, but NewGen extends it to any other language (such as C).

Disk files created by the write function are independent of the programming language. This implies that all passes need not be written in the same language. The most appropriate language can be chosen for each one: C will be preferred for its execution speed and CommonLISP for its programming flexibility.

This pass-based organization allows a prototype to be gradually converted to a more industrial product by rewriting in C, with an imperative flavor, passes initially written in a functional style in CommonLISP. Moreover, as integration goes on, data transfers on disks can be replaced by pipe streams and eventually, when passes are compiled together, to simple global variables. This smooth *evolution process* is one of the major assets of NewGen and a good reason for using this tool in a complex project.

NewGen Provides a Better Programming Environment.

When NewGen is used to design and implement a collection of data types, it generates a set of tools that may be used within a program to declare, create, access and delete objects of these types. It also provides a library of generic

tools to create, access or delete lists, sets and arrays of objects. For instance, macros or functions equivalent to the Lisp functions `mapcar` or `reverse` are provided in the C implementation.

From the user point of view, the programming language is greatly improved, and this leads to at least four advantages. The code is more readable, it is more rapidly developed, easier to debug and can be extended more transparently:

- The code is more readable because of the programming style enforced by the NewGen tools: names of creation and access functions are derived from type names, and this is always done in the same consistent way; all lists have the same structure, with objects linked together with *cons* cells, and so on. Moreover, even across languages, the same style is preserved. This helps the conversion of programmers from one version of a program to another.

- The code is more rapidly developed because the use of predefined functions allows programmers to write less code. Generic programming on lists, sets and arrays leads to the same result. For instance, one call to the creation functions produced by the C version of NewGen executes all the necessary operations to create a new object of a given type: memory allocation, memory overflow check, dynamic type check of arguments and assignment of objects passed as arguments to the new object components (if efficiency is at premium, these checks can be deactivated).

- Programs are easier to debug because a lot of dynamic type checks are done when objects are created and accessed. For instance, the access macros provided by NewGen optionally check at run time that the object passed as argument is of the right type before accessing it. Other functions and macros perform similar checks. The degree of dynamic checking is user-tunable.

As a general matter, NewGen provides a transparent extension of the embedding language and improves code compactness and documentation.

4 DDL is NewGen Domain Definition Language

NewGen types are called *domains*. Domains are defined by a set of *domain definitions* which are analyzed by a language dependent implementation of NewGen to produce the programming tools mentioned in the previous section. Domain definitions are written in a simple high level language called *DDL*. In the following, `[form]` means that `form` is optional, and `[form]*` means that `form` may be repeated zero or more times.

4.1 Domains Can Be Predefined

NewGen provides a set of pre-defined domains that correspond more or less to the basic types supplied by most programming languages: `bool`, `int`, `float`,

`char`, `string`, etc. An extra pre-defined domain called `unit` is provided by NewGen. It is mostly useful with the union operator to create enumerated types. An object of this domain may take only one value, denoted `UU` in this paper. This list could be extended if necessary, for instance if double precision arithmetic were critical for an application. Pre-defined domains are the basic material over which user-defined domains are built.

The set of values that might be taken by an object of a pre-defined domain is equal to the set of values that might be taken by a variable of the corresponding type, in the programming language used by NewGen.

4.2 Users Can Define Domains

More complex domains may now be defined with the *product* operator `x` or with the *union* operator `+`.

A product domain is like a C or CommonLISP structure or a Pascal record. It is a collection of named components called *members*, that can be of different types. A product domain should be defined when related objects need to be encapsulated into a single one. Valid product domain specifications have the following form:

```
product-domain = member [ x member ]* ;
```

Let P be a product domain defined by $P = M_1 \times M_2 \times ... \times M_n$. Let V_i be the value set of member M_i. The value set for P's objects is the Cartesian product of V_1, V_2, ..., V_n.

A union domain is like a C union or a Pascal variant record. A union domain also is a collection of named components called members, but, unlike product domains, each object may hold at most one member at a time. A union domain should be defined when several objects of different types must be manipulated as if they were of the same type. Valid union domain specifications have the following form:

```
union-domain = member + member [ + member ]* ;
```

Let U be a union domain defined by $U = M_1 + M_2 + ... + M_n$. The value set for P's objects is the tagged or disjoint union of V_1, V_2, ..., V_n.

4.3 Domains Contain Members

We now have to define what a member is. The general form of a member is *name:type*, where *name* is the member name, i.e. the name of the component within the domain and *type* is the member type. The type may be simple, structured as a list, a set or an array.

Members Can Be Simple

A simple member has the form `n:d` where the member name is `n`. The type `d` is the name of a predefined domain or of a domain defined in the same DDL specification or imported from another one (see 4.5). The simplified notation `n` is a short hand for `n:n`. The value set of a simple member is equal to the value set of the corresponding domain `d`.

Here is an example of a product domain composed only with simple members:

```
person = name:string x extension:int x login ;
```

The domain `person` has three members: a `string` named `name`, an `int` named `extension` and a `login` named `login`; `int` and `string` are pre-defined domains, and so need no further specifications. `login` is a user-defined domain that has to be defined later or previously for the DDL specification to be handled correctly by NewGen.

As a convenience, the notation

```
union-domain = { member , member [ , member ]* } ;
```

is a short hand for

```
union-domain = member:unit + member:unit [ + member:unit ]* ;
```

and is used to denote enumerated types (like Pascal's `SET` or C's `enum`).

Members Can Specify Lists

A list member has the form `n:d*` where `n` and `d` have the same meaning as in a simple member. The value set of a list member is equal to finite or infinite (NewGen allows circular data structures) sequences of elements of type `d`.

Here is an example of a list domain that could be used to describe directed graphs with node information of type `int`:

```
node = information:int x successors:node* x predecessors:node* ;
```

The domain `node` has three members: an `int` named `information`, a list of `node`s named `successors` and a list of `node`s named `predecessors`. Note that this data type definition is recursive and allows self referencing structures to be created.

Members Can Specify Sets

A set member has the form `n:d{}` where `n` and `d` have the same meaning as in a simple member. The value set of a set member is equal to finite collections of elements of type `d`.

Here is an other way to describe directed graphs:

```
node = information:int x successors:node{} x predecessors:node{} ;
```

In this example, links between nodes are kept in sets instead of lists.

Members Can Specify Arrays

An array member has the form `n : d[i1]...[im]` where `n` and `d` have the same meaning as in a simple member and $\texttt{i}_\texttt{j}$ are compile-time constants. NewGen allows compile-time symbolic constants to be defined with the syntax of the C preprocessor.

The value set of an array member is equal to sequences of length $\texttt{i}_\texttt{1} \times ... \times \texttt{i}_\texttt{m}$ of elements of type `d`; the access to elements is of complexity $O(1)$.

Here is an example of an array domain that could be used to describe a circular buffer of characters:

```
#define BUFFER_SIZE 1000

buffer = first:int x last:int x elements:char[ BUFFER_SIZE ] ;
```

The domain `buffer` has three members: two `int`'s named `first` and `last` that specify positions of elements in the buffer, and an array of `char`'s named `elements` to store the elements.

4.4 Domains Can Be Tabulated

Objects of a given domain have an infinite extent, i.e. they are allocated in heap storage and have to be freed when they are no longer useful (either explicitly in a language like C or automatically in a garbage-collected language like CommonLISP). Thus, objects of a given NewGen domain can be created at will and there is no a priori relationship between them.

However, NewGen allows domains to be *tabulated* in such a way that all the values of a given type are, when created, kept together and are processed as a whole (this can be seen as defining the domain in extension). For instance, it is possible to apply a given function to all the values of a given tabulated domain or to write the whole domain on a file. This facility has deeper consequences as described later.

A user-defined domain can be declared tabulated by prefixing its definition with the keyword `tabulated`. As an example, here is the definition of the tabulated domain of symbols in a compiler:

```
tabulated symbol = name:string x type x scope:int ;
```

Note that the first member of a tabulated domain has to be a string (which means that only product domains can be tabulated). Any tabulated value is uniquely defined by the string value of the first member which serves as a non ambiguous key. Tabulated domains implement dictionaries.

This technique can be used, for instance, to hide the usual symbol tables that occur in many program manipulation systems like compilers. The symbol table is automatically managed by NewGen, once the domain of symbols has been tabulated. Thus, the programmer who manipulates the intermediate representation does not have to worry about what is stored in the symbol table and what is not.

Tabulated domains are also important to precisely manage the deallocation of storage (see below).

4.5 Imported Domains Permit Modular Specifications

The design and maintenance of large softwares require a modular specification. NewGen permits specifications to be split in different files and imported on a domain basis. However, recursive data type definitions have to be written in the same file. To be processed by NewGen, all the specifications of imported domains have to be available. If some of them are not yet defined, the user has to provide some stub domain (e.g., `int` or `unit`).

The following is used to declare an imported domain `imported-domain` located in a specification file `file`:

```
import imported-domain from "file"
```

Here is an example of an imported domain that could be used to describe an Ethernet network of workstations:

```
import workstation from "Include/workstation.newgen" ;
import gateway from "Include/gateway.newgen" ;

network = nodes:node* ;
node = workstation + gateway + repeater:node*;
```

The domains `workstation` and `gateway` define the structure of any workstation or gateway plugged on the network while a `repeater` connects several nodes.

4.6 External Domains Increase Software Reusability

NewGen adapts itself to already existent software and thus allows a smooth transition from hand-coded data types to NewGen generated domains. The only requirements are that external data types be coercible to the least stringent type of the implementation language (e.g., `char*` for C), and that the user provides a set of functions to free, write and read an object of this type.

Domains can be declared external in the following way:

```
external external-domain ;
```

Here is an example of an external domain that describes old printing technology:

```
external punch ;
import laser from "printers.newgen" ;
import daisy from "printers.newgen" ;

output_device = laser + daisy + punch ;
```

The domains `laser` and `daisy` define today's printing technology while the old `tape` punch is managed without NewGen generated functions.

5 NewGen Generates Functional Implementations

From a set of NewGen DDL specifications (possibly split across several files), NewGen generates a file of function definitions (whether they are macros or real functions is not really important here) and a so-called *spec* file that contains a compiled description of NewGen data types.

5.1 Each Domain Defines a Set of Functions

The functions created by NewGen allow the manipulation of objects whose types are given by the specifications. We describe below the functions provided by the C-NewGen version (CommonLISP-NewGen is similar). In the following, `d` is a user-defined domain and `m` one of its members.

Creation and Initialization Functions

The definition of domain `d` introduces the type `d` and the creation function `make_d`. The type `d` must be used to declare domain `d`'s object descriptors and the function `make_d` to actually create and initialize objects. If `d` is a product domain, `make_d` expects as many arguments as there are members in `d`'s definition. If `d` is a union domain, then the first argument is a tag – for each member `m`, NewGen defines the tag `is_d_m` – and the second a value of the appropriate type. `Make_d` returns a descriptor of an object of type `d`.

A special value `d_undefined` is provided for any user-defined domain `d`. This value may be used with the `make_<domain>` functions to create partially initialized objects.

Access and Modification Functions

For each domain and each of its member, an access function `d_m` is generated. This function takes an object of type `d` and extracts the value corresponding to `m`; it returns a value whose type is the type of the member. These access functions can be used in left-hand sides of assignments to modify the objects. This is reminiscent of the way CommonLISP specifies mutation, via the `setf` macro; in this way also, C and CommonLISP programs share the same programming style.

If `d` is a union domain, the function `d_tag` is generated. It returns the tag of an object. Tag equality is `=` so that the `switch` construct can be used; moreover, boolean functions like `d_m_p` are also defined, where `_p` stands for predicate.

In practice, values are usually more often accessed and updated than created or destroyed. We thus accepted to pay a larger run-time overhead at creation time (one function call) than access time. This is why all the access functions are actually macros that directly address substructures of NewGen values. This implies a very efficient implementation of NewGen programs since no function calls are involved in data manipulation.

Input and Output Functions

In order to write and read NewGen data on streams, the NewGen library provides two functions for each domain `d`:

- `write_d` expects a stream and a NewGen value and outputs the value on the stream; the values are recursively written, *except* for values of tabulated domains for which only the key is written.
- `read_d` expects a stream from which a NewGen value is read. The sharing of NewGen pointers within the value previously written is preserved by this operation. If a tabulated value is referenced, it has to be already present in memory.

Since values from tabulated domains are not written when they appear inside another data structure, NewGen provides the `gen_write_tabulated` function to write the entire set of values of the tabulated domain and `gen_read_tabulated` to read a set of tabulated values from a stream.

Deletion Functions

Since C doesn't provide a garbage collector, any useless NewGen value of domain `d` has to be explicitly freed if one doesn't want to experience an eventual exhaustion of memory. The function `free_d` recursively liberates the storage associated to a given value, `except` for tabulated values. These ones are freed only if they are directly passed to `free_d` as an argument.

Library Functions

The NewGen library provides the usual set of basic operators to deal with lists, sets and arrays such as `CAR`, `CDR` or `CONS` which are used to create and manipulate lists (they can be freed by `gen_free_list`). Functions such as `gen_mapl` to apply a function on every list element or `gen_nreverse` to reverse the order of a list are also provided.

The specification file is used at run time to create a core image of the NewGen domain definitions (see below); it is used by generic library functions to "parse" NewGen generated values. It has to be read by the `gen_read_spec` library function before any domain-related function is used.

6 An Extended NewGen Example

The following example presents an interpreter for a very small subset of an expression language, close to Lisp. Although limited in scope, it shows how NewGen helps in merging together the C and Lisp worlds. In particular, this allows different programming styles and tools to be integrated in the same project.

The *slanted font* is used within program fragments to denote macros, functions, types or constants that are automatically generated by NewGen or that belong to the NewGen library.

6.1 The Front-end Program

We use the powerful toolkit provided with the Unix operating system to write the front-end: LEX for the lexical analysis and YACC for the parsing phase. These tools, available from C, will ease the tedious part of the source analysis process.

DDL Specifications — The Expression Syntax.

The abstract syntax for our expression language is quite simple and is given below. NewGen provides a LaTeX preprocessor with which specifications can be written, using the `domain` macro. From this file, both the `.newgen` specification file and a documentation report are extracted. We only give here the newgen version, `expression.newgen`, of this file.

```
-- Specifications for an expression language.

-- The description of identifiers is irrelevant here, and is given in a
-- separate imported file.
import identifier from "identifier.newgen" ;

-- The compacted domain could denote values that are computed by a
-- separate program that doesn't use NewGen.
external compacted ;

-- A user expression is a let construct, which includes a binding list
-- (each binding binds a name to its expression) and a body expression.
let = bindings:binding* x expression ;
binding = name:string x value:expression ;

-- An expression is either an integer constant, an identifier, a
-- compacted value, a binary expression, an associative expression or
-- a nested let construct.
expression = constant:int +
             identifier +
             compacted +
             binary +
             associative +
             let ;

-- A binary has an operator and a pair of expressions.
binary = operator:string x lhs:expression x rhs:expression ;

-- This is an associative, commutative and unitary operator with a
-- set of operand expressions.
associative = operator:string x operands:expression ;
```

The separate DDL file `identifier.newgen` for identifiers is given below:

```
-- Specification of identifiers

tabulated identifier = name:string ;
```

Identifiers are tabulated to enable operations that globally manipulate this domain. For instance, this is required if one wants to implement a `gensym`-like function that creates a fresh new symbol (see below).

From such DDL files, the `newgen` compiler generates two files. The first one, called the "spec" file, is a compacted description of the DDL file that has to be read before the first NewGen-specific value is created by a user program. The second file is specific to the implementation programming language. The C version of NewGen generates a C file to be included (with the `#include` directive) while the CommonLISP version generates a Lisp file to be loaded.

Program Fragments

The lexical analyzer and parser are written in C via LEX and YACC specification files. We only look here at the parser, since it uses NewGen.

The `main` program is straightforward. It mainly:

- initializes NewGen run-time environment . Whenever a spec file is read with the `gen_read_spec` function, a data structure is created that describes the logical structure of NewGen-generated domain. This description is used whenever a NewGen-value of a given domain is created (by a `make` function), freed or printed. Every NewGen object contains a field holding its domain number.
- calls the parser `yyparse` to update the value of the variable `Let`, of type `expression`, that holds the whole user expression.
- writes `Let` on the standard output stream.

```
#include <stdio.h>           /* Unix standard IO */
#include "genC.h"            /* Newgen basic C library */
#include "identifier.h"      /* Newgen-generated header files */
#include "expression.h"

expression Let ;

main()
{
  gen_read_spec("identifier.spec", "expression.spec", (char*) NULL) ;
  yyparse() ;
  write_expression( stdout, Let ) ;
}
```

NewGen must read the specification files at run-time so as to have a full description of all the data types involved. This is necessary to perform dynamic type checking or to "walk" through domains with generic functions. Note that the order in the argument list of `gen_read_spec` is important and is automatically provided by the call to the `newgen` compiler.

The input and parsing of a user expression is performed by the semantic actions of the YACC file. We give below some excerpts of these actions; they

mainly call NewGen-generated functions to create the relevant abstract syntax nodes.

```
Let             :  LEFT_PAR LET LEFT_PAR Bindings RIGHT_PAR Expression RIGHT_PAR
                     {$$ = make_let( $4, $6 ) ;}
                ;

Bindings        :
                     {$$ = NIL ;}
                | Bindings LEFT_PAR String Expression RIGHT_PAR
                     {$$ = CONS( BINDING, make_binding( $3, $4 ), $1 ) ;}
                ;

Expression      :  Int
                     {$$ = make_expression( is_expression_constant, $1 ) ;}
                | LEFT_PAR String Expression Expression RIGHT_PAR
                     {binary b = make_binary( $2, $3, $4 ) ;
                      $$ = make_expression( is_expression_binary, b );}
                | LC String Expressions RC
                     {associative a = make_associative( $2, $3 ) ;
                      $$ = make_expression( is_expression_associative, a ) ;}
                | Identifier
                     {$$ = make_expression( is_expression_identifier, $1 ) ;}
                | LEFT_PAR Expression RIGHT_PAR
                     {$$ = $2 ;}
                | Let
                     {$$ = make_expression( is_expression_let, $1 ) ;}
                ;

Expressions       :
                     {$$ = set_make( set_pointer ) ;}
                | Expressions Expression
                     {$$ = set_make( set_pointer ) ;
                      set_add_element( $$, $1, $2 ) ;}
                ;

Identifier        :  String
                        {$$ = make_identifier( $1 ) ;}
                ;
```

Note that `set_make` accepts an argument that specifies the kind of set to be created: NewGen values (`set_pointer`), strings (`set_string`) and integers (`set_int`). The library function `set_add_element` stores in its first argument the set corresponding to its second to which the third argument has been added.

6.2 The Evaluator

Once analyzed by the front-end, the parse tree of the user expression is transferred (via a file) to a CommonLISP program that will evaluate this expression.

In a similar manner, some definition files have to be loaded:

```
(require "genLisplib")            ; Newgen basic Lisp library
(require "identifier")            ; Newgen-generated header files
(require "expression")

(use-package '(:newgen :identifier :expression))
```

Then, the `main` program initializes NewGen, reads an expression parsed and saved by the front-end in a file and evaluates it.

```
(defun test (file)
  (gen-read-spec)
  (let ((*standard-input* (open file)))
    (eval-let (read-expression) '())))
```

The evaluator is a recursive function that takes the expression to evaluate and an initially-empty environment that binds every name to its value. The environment is expanded whenever a new `let` expression is encountered.

```
(defun eval-let (l env)
  (let ((new-env (mapcar #'(lambda (b)
                             `(,(binding-name b) .
                               ,(eval-expression (binding-value b) env)))
                         (let-bindings l))))
    (eval-expression (let-expression l) (append new-env env))))
```

For each expression, a dispatch is performed according to the particular tag associated to the expression. The NewGen construct `gen-switch` selects the tag corresponding to `e` (e.g., `is-expression-constant`) and, in each clause, binds the given variable to the corresponding value (e.g., `c` is bound to the integer corresponding to `e`).

```
(defun eval-expression (e env)
  (gen-switch e
    ((is-expression-constant c) c)
    ((is-expression-identifier i) (eval-identifier i env))
    ((is-expression-let l) (eval-let l env))
    ((is-expression-binary b) (eval-binary b env))
    ((is-expression-associative a) (eval-associative a env))))
```

For an identifier, the corresponding value is looked for in the environment.

```
(defun eval-identifier (i env)
  (let ((var-val (assoc (identifier-name i) env :test #'string-equal)))
```

```
    (if (null var-val)
        (error "~%Unbound identifier ~S" (identifier-name i))
        (cdr var-val))))
```

To evaluate operations, we use the `operators` and `unitaries` a-lists that bind every operator name to the corresponding CommonLISP function and unitary value:

```
(defparameter operators
  `((,"add" .  ,#'+)
    (,"sub" .  ,#'-)
    (,"times" .  ,#'*)
    (,"cons" .  ,#'cons)
    (,"eq" .  ,#'eq)))

(defparameter unitaries
  `((,"add" .  0)
    (,"times" .  1)
```

The evaluation function for binary and associative expressions is thence straightforward by induction on the subexpressions. Note that all this treatment could be simplified by the use of the `gen-recurse` construct, supported in NewGen, that combines dispatching and recursion [JD89].

```
(defun eval-binary (b env)
  (let ((op (assoc (binary-operator b) operators :test #'string-equal)))
    (if (null op)
        (error "~%Incorrect op code ~S" (binary-operator b))
        (funcall (cdr op)
                 (eval-expression (binary-lhs b) env)
                 (eval-expression (binary-rhs b) env)))))
```

NewGen provides a library of operations on sets. To be compatible with C (and its lack of automatic garbage collector), every operation is a three-address subroutine in which the result is specified as the first argument.

```
(defun eval-associative (a env)
  (let ((op (assoc (associative-operator a) operators :test #'string-equal))
        (result (assoc (associative-operator a) unitaries :test #'string-equal)))
    (if (or (null op) (null result))
        (error "~%Incorrect op code ~S" (associative-operator a))
        (set-map #'(lambda (exp)
                     (setf result
                           (funcall (cdr op)
                                    result
                                    (eval-expression exp env))))
                 (associative-operands a)))
    result))
```

For the sake of the example, identifiers were tabulated. This could be useful if one wanted to add a `gensym` function (with the appropriate extension of the domain of values to include symbols and the introduction of additional operators). A `gensym` function creates freshly allocated symbols and must check that they are unique; it could be written in the following way:

```
(defun eval-gensym (&rest args)
  (declare (ignore args))
  (do ((i 0 (+ i 1)))
      ((gen-find-tabulated (format nil "gensym-~D" i) identifier-domain)
       (make-identifier :name (format nil "gensym-~D" i)))))
```

where the `do` loop is run as long as the proposed gensym'ed string is currently a used symbol name. The library function `gen_find_tabulated` returns the object that corresponds to the given key in the appropriate tabulated domain (or `nil` if it doesn't exist). Note that `make-identifier` automatically updates the domain of identifiers to include the newly created identifier.

7 Conclusion

NewGen generates complex manipulation functions of data structures from high-level specifications. These functions can be implemented in different programming languages and paradigms (C and CommonLISP are currently supported). The *functional abstraction* paradigm that NewGen promotes increases the portability of programs and programmers by enforcing a common programming style. Inter-language communications of values via shared memory, pipes or disk files with preservation of sharing patterns is supported, thus allowing gradual integrations of loosely-coupled modules within large projects.

This tool has been extensively used within the PIPS (Ecole des Mines) and the PMACS projects (BULL Corporate Research Center) that strive to design and implement sophisticated parallelizing compilers for Fortran programs. A clear improvement in programming efficiency and reliability has been noticed, thus supporting a wider use of this kind of software tools.

NewGen is implemented in C and runs under SunOs 4.0 and System V. It heavily uses Yacc and Lex to parse DDL files and transfer NewGen values (the CommonLISP version uses read dispatch macro characters for this purpose). NewGen is available from the authors.

References

[**A89**] Deux, O. *The Story of O2.* Technical Report, Altair, 1989

[**AH87**] Andrews, T., and Harris, C. *Combining Language and Database Advances in Object-Oriented Development Environment.* Proc. of the OOPSLA'87 Conf, SIGPLAN Not. Vol. 22 (12), 1987

[**BN83**] Birrell, A. D., and Nelson, B. J. *Implementing Remote Procedure Calls.* XEROX CSL-83-7, October 1983.

[**CM85**] Cardelli, L., and MacQueen, D. *Persistence and Type Abstraction.* Proc. of the Persistence and Data Types workshop, Scotland, August 1985.

[**JD89**] Jouvelot, P., and Dehbonei, B. *Recursive Pattern Matching on Concrete Data Structures.* SIGPLAN Not. Vol. 24 (11), 1989

[**KBCGW88**] Kim, W., Ballou, N., Chou, H., Garza, J., and Woelk, D. *Integrating an Object-Oriented Programming System with a Database System.* Proc. of the OOPSLA'88 Conf, SIGPLAN Not. Vol. 23 (11), 1988

[**KR78**] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language.* Prentice Hall, 1978

[**L87**] Lamb, D. A. IDL: sharing intermediate representations. *ACM Trans. on Prog. Lang. and Syst. 9*,3, July 1987

[**S84**] Steele, G. L. Jr. *CommonLISP.* Digital Press, 1984

[**S87**] Sun Microsystems, Inc. *External Data Representation Standard.* ARPA NIC RFC 1014, June 1987

[**S89**] Snodgrass, R. *The Interface Description Language.* Computer Science Press, 1989