NewGen

Pierre Jouvelot

CRI, Ecole des Mines (France)
LCS, MIT (U.S.A.)

Conférence GRECO de Programmation
Paris, 28 Novembre 1991

Qu'est-ce que NewGen

— NewGen (New Generation)
— Outil de génie logiciel
— Aide à l'écriture de logiciels de taille importante
— Développement à l'Ecole des Mines de Paris (Automne 1988)
— Prototype "Public Domain" distribué (Mines, Bull, NASA, Boeing, ...)
— Applications : Paralléliseurs (environ 50 klignes)

NewGen en résumé

— Outil de gestion de structures de données
— Méta-langage de définition de *domaines* : base, somme, produit, liste, ensemble, tableau
— Génération automatique de fonctions de création, manipulation, mise à jour, libération, stockage ...
— Bibliothèque générale (listes, tableaux, itérateurs, ...)
— Intégration dans logiciels existants (externe)
— Modularité (import)
— Efficacité temps/espace

Plan

— Problématique
— Atouts
— Description du langage de spécification
— Fonctions de manipulation (en C)
— Bibliothèques
— Un Premier Exemple
— NewGen “avancé” : tabulation, importation, external
— Retour sur l’exemple
— NewGen et CommonLISP
— Aspects de l’implémentation
— PIPS : Un exemple “vraie grandeur”
— Autres systèmes
— Conclusion

Problématique

— Modèles de Développement Logiciel : *waterfall*, prototypage
— Prototypage : Spécification vs. Implémentation
— Spécification : Exécutabilité vs. Expressivité
— Implémentation : Efficacité vs. Simplicité
— Solution : *Interopérabilité*

Interopérabilité

— Indépendance par rapport au langage cible
— Paradigme uniforme de programmation
— Etat : support pour CommonLISP (orienté spécification) et C (orienté implémentation)
— Compatibilité complète au niveau fichier
— Maintien de la cohérence (persistence)
— Transition souple (conception modulaire)

Atouts

— *Abstraction Fonctionnelle* :
  — Constructions de haut niveau (fonctions)
  — Indépendance de l'implémentation
  — Extensions/redéfinitions possibles
— *Processus Evolutif* :
  — Support pour logiciel multi-passes
  — Gestion de persistence (partage, cycle)
  — Multi-langages
  — Intégration progressive (fichiers, pipes, variables globales)
— *Environnement de Programmation* :
  — Lisibilité inter-langages
  — Réutilisation des bibliothèques
  — Mise-au-point aisée (tests dynamiques possibles)

Langage de Description de Domaines

— `--` pour les commentaires
— Définitions de domaines :
`name = expression ;`
— Prédéfinis : `unit`, `bool`, `char`, `int`, `string`, `float`
— Produit de *membres* :

```
user = name:string x id:int x
       passwd x shell:string ;
```

— Somme de membres :

```
passwd = crypted:string +
         clear:string ;
```

— Enumération (vue comme somme) :

```
passwd_status = {crypted, clear} ;
```

— Les *membres* peuvent être complexes
— Définitions récursives autorisées
— Liste ordonnée de domaines :

```
node = information:int x
       children:node* ;
```

— Ensemble (non ordonné) de domaines :

```
group = elements:user{} ;
```

— Tableau (indexé) de domaines :

```
#define BUFFER_SIZE 100

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

Fonctions de Manipulation : Création

— Tout domaine *domain* définit :
  — un type *domain*
  — un *constructeur* `make_`***`domain`***
  — une valeur par défaut ***`domain`***`_undefined`
— Domaines produit : création à partir des membres :

```
user pierre =
    make_user( "jouvelot",
               110,
               passwd_undefined,
               "/usr/local/bin/ksh" ) ;
```

— Domaines somme : création à partir d'un *tag* et d'une valeur
— Chaque membre *member* d'une somme *domain* a un tag `is_`***`domain_member`*** :

```
char buffer[ 8 ] ;
passwd at_login =
   make_passwd( is_passwd_encrypted,
                crypt( gets( buffer ),
                       "aa" )) ;
```

— Il existe un type `tag`.

Fonctions de Manipulation : Accès

— Call-by-sharing (vs. call-by-value, call-by-reference)

— Un domaine *domain* et un membre *member* définissent un *accesseur* ***domain_member*** :

```
printf( "User %s logged on\n",
        user_name( pierre )) ;
```

— Le tag d'une somme *domain* s'obtient par ***domain***_tag :

```
if( passwd_tag( at_login ) ==
    is_passwd_encrypted ) {
    check( passwd_crypted( at_login )) ;
}
```

— Des prédicats ***domain_member***_p existent pour les sommes :

```
if( passwd_encrypted_p( at_login )) {
    check( passwd_crypted( at_login )) ;
}
```

— Implémentation sous forme de macros.

Fonctions de Manipulation : Modification

— Utilisation de = :

```
passwd_tag( at_login ) =
     is_passwd_clear ;
passwd_clear( at_login ) = "go ahead" ;
```

— Création de partage et cycle :

```
node = info:int x next:node ;

node n =
   make_node( 1, node_undefined ) ;
next( n ) = n ;
```

Fonctions de Manipulation : Opérations I/O

— Tout domaine *domain* définit :
  — une fonction d'écriture `write_`*`domain`*
  — une fonction de lecture `read_`*`domain`*

```
user pierre = read_user( open_db()) ;
fprintf( stderr,
         "Read data for user %s\n",
         user_name( pierre )) ;
```

— Le partage (sharing) est preservé dans la sous-structure.
— Gestion des cycles.

Fonctions de Manipulation : Libération

— Tout domaine *domain* définit `free_`***`domain`***.

```
if( denied_access( pierre )) {
    fprintf( sdterr,
             "Permission denied: %s\n",
             user_name( pierre )) ;
    free_user( pierre ) ;
    restart_top_level() ;
}
```

— Réclamation récursive des structures de données

— Gestion des cycles et partage *dans la sous-structure* (attention au partage transverse)

— Remarque : pas nécessaire en CommonLISP !

Bibliothèques

— A chaque constructeur de type est associée une bibliothèque
— Listes
— Ensembles
— Remarque : tables de hachage

Bibliothèque : Listes

— Constructeurs classiques de Lisp : CONS, CAR, CDR, NIL, ENDP, ...

— Nécessité de typage explicite (listes polymorphes)

— Tout domain *domain* définit la conversion *DOMAIN* :

```
list logged_on = NIL ;

void add_to_users( u )
user u ;
{
    logged_on =
        CONS( USER, u, logged_on ) ;
}
```

— Itérateurs :

```
printf( "Users logged on: " ) ;
MAPL( users, {
    user u = USER( CAR( users )) ;
    printf( "%s ", user_name( u )) ;
}, logged_on ) ;
```

— Mise-à-jour simple :

```
CAR( logged_on ) = pierre ;
```

— Ensemble "extensible" de fonctions (`remove`, `nconc`, `copy`, `find`, `length`, ...)

Bibliothèque : Ensembles

— Problème : gestion d'ensemble (et non multi-ensembles), implémentation efficace
— Support pour chaines, entiers et pointeurs
— Convention d'allocation à l'appelant (solution simple pour appels emboités).
— Opérations triadiques :

```
set_op( result, operand1, operand2 ) ;
```

— Nécessité d'allocation :

```
set logged_on = set_undefined ;

void add_to_users( u )
user u ;
{
    if( set_undefined_p( logged_on )) {
        logged_on =
             set_make( set_pointer) ;
    }
    set_add_element( logged_on,
                     logged_on,
                     u ) ;
```

```
    }
```

— Itérateurs :

```
    printf( "Users logged on: " ) ;
    SET_MAP( u, {
        printf( "%s ", user_name( u )) ;
    }, logged_on ) ;
```

— Ensemble “extensible” de fonctions :

  — `set_intersection`,
  — `set_union`,
  — `set_equal_p`,
  — `set_free`,
  — `set_size` ...

Bibliothèque : Tables de hachage

— Tables dans Unix SV “inutilisables”
— Utilisation fréquente : ensembles, NewGen, ...
— Itérateurs :

```
#define SET_MAP(element,code,set) { \
    HASH_MAP(_set_map_key, element, \
            code, \
    (set)->table); \
    }
```

Un Premier Exemple

— SIMPLE est un petit langage d'expression

— Fichier `expression.tex` :

```
\title{SIMPLE Language Specifications}
\author{Pierre Jouvelot}

\begin{document}
\domain{expression = constant:int +
                     identifier:string +
                     binary +
                     let ;} {
An expression is either an integer constant,
an identifier, a binary expression, or a
nested let construct.
}

\domain{binary = operator:string x
                 lhs:expression x
                 rhs:expression ;} {
A binary expression consists of an operator
and two subexpressions.
}
```

```
\domain{let = bindings:binding* x
              expression ;} {
A let construct includes a binding list and
a body expression.
}

\domain{binding = name:string x
                  value:expression ;} {
A binding binds a name to a value.
}
```

— Fichier `expression.newgen` trié, automatiquement généré

```
binary = operator:string x
         lhs:expression x rhs:expression ;
binding = name:string x value:expression ;
expression = constant:int +
             identifier:string +
             binary + let ;
let = bindings:binding* x expression ;
```

— Fichier `expression.dvi` de documentation automatiquement généré.

Frontal pour SIMPLE

— Syntaxe d'entrée à la Lisp :

```
. 1
. (+ x 1)
. (let ((x 1)) (+ x 2))
. (let ((x 1))
    (let ((y (* 2 x)))
      (+ x y)))
```

— Génération des structures de données NewGen :

```
. make_expression(is_expression_constant,1)
. make_expression(
    is_expression_binary,
    make_binary(
      "+",
      make_expression(
        is_expression_identifier,
        "x"),
      make_expression(
        is_expression_constant,
        1)))
```

— NewGen est compatible avec tous les outils Unix

— Frontal généré automatiquement par Yacc

```
%{

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

expression Top ;
%}

%token LP RP
%token LET

%term INT
%term STRING

%union {
    expression expression ;
    let let ;
    list list ;
    identifier identifier ;
    string string ;
}
```

```
%type <expression> Axiom Expression
%type <let> Let
%type <identifier> Identifier

%type <list> Bindings
%type <string> String

%%
Axiom   : Expression {
            Top = $1 ;
        }
        ;

Expression
        : INT {
            $$ = make_expression(
                 is_expression_constant,
                 atoi( yytext )) ;
        }
        | Identifier {
            $$ = make_expression(
                 is_expression_identifier,$1);
        }
        | LP String Expression Expression RP {
```

```
        binary b =
        make_binary( $2, $3, $4 ) ;

        $$ = make_expression(
             is_expression_binary, b );
    }
```

```
        | Let {
            $$ = make_expression(
                 is_expression_let, $1 ) ;
        }
        ;

Let     : LP LET LP Bindings RP Expression RP {
            $$ = make_let( $4, $6 ) ;
        }
        ;

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

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

```
        }
        ;

String  : STRING {
            $$ = strdup( yytext ) ;
        }
        ;
%%
```

Commande `newgen`

— Commande shell de génération de code
— **`newgen`** prend en arguments :
  — Language objet (**`-C`**, **`-Lisp`**),
  — Fichiers **`.newgen`**

```
% newgen -C expression.newgen
GEN_READ_SPEC order:
expression.spec
% ls
expression.newgen       expression.h    expression.
%
```

— Pour chaque fichier **`foo.newgen`**, on obtient deux fichiers :
  — Déclarations C : **`foo.h`**,
  — Spécifications : **`foo.spec`**
— Remarque : les **`spec`** devraient disparaitre dans une nouvelle version de NewGen
— Fichiers **`spec`** lus à l'exécution, *avant* tout appel de fonctions NewGen.
— Ordre des fichiers **`spec`** donné par **`newgen`**

```
#include <stdio.h>
#include "genC.h"
#include "expression.h"

expression Top ;


main()
{
    gen_read_spec( "expression.spec",
                   (char*) NULL) ;
    yyparse() ;
    fprintf( stdout, "%d\n",
             constant_fold( Top )) ;
    free_expression( Top ) ;
}
```

```
int
constant_fold( e )
expression e ;
{
    int value ;
    tag t ;

    switch( t = expression_tag( e )) {
    case is_expression_constant:
        value = expression_constant( e ) ;
        break ;
    case is_expression_binary:
        binary b = expression_binary( e ) ;
        int lhs = constant_fold(binary_lhs(b));
        int rhs = constant_fold(binary_rhs(b));

        value =
           eval_primitive( binary_operator(b),
                           lhs, rhs ) ;
        break ;
    default:
        fprintf( stderr,
                 "Unimplemented %d\n",
                  t ) ;
        exit( 1 ) ;
```

```
    }
    return( value ) ;
}

int
eval_primitive( op, lhs, rhs )
char *op ;
int lhs, rhs ;
{
    if( strcmp( op, "+" ) == 0 )
        return( lhs+rhs ) ;
    if( strcmp( op, "-" ) == 0 )
        return( lhs-rhs ) ;
    if( strcmp( op, "*" ) == 0 )
        return( lhs*rhs ) ;
    if( strcmp( op, "/" ) == 0 )
        return( lhs/rhs ) ;

    fprintf( stderr, "Primitive %s unknown\n",
             op ) ;
    exit( 1 ) ;
}
```

Aspects Avancés
Tabulation

— Domaines *tabulés*
— Accès global aux objets d'un même type
— Le premier membre *doit* être une chaine (unique par objet) :

```
tabulated user = name:string x id:int x
                 passwd x shell:string ;
```

— Permet une dissociation entre définition et référence
— Notion de *object-id* en programmation persistente
— Unicité des objets (`name` est une clé utilisée à la création des objets)

— Utilisation : déallocation, accès fichiers, ...

```
user pierre, francois, michel ;

list roots = CONS( USER, pierre,
                   CONS( USER, francois,
                                NIL )) ;
list admins = CONS( USER, michel,
                    CONS( USER, pierre,
                                 NIL )) ;

group root = make_group( roots ) ;
group admin = make_group( admins ) ;

free_group( admin ) ;

--> CAR( group_elements( root )) ????
```

— Si `user` est tabulté, pas de libération automatique

— Chaque domaine tabulé *tab* définit *tab*_domain
— Manipulation globale d'objets tabulés en mémoire :

```
TABULATED_MAP( u, {
    fprintf( stdout, "User %s\n",
             user_name( u )) ;
}, user_domain ) ;
```

— Libération explicite (même en CommonLISP) et IO :

```
FILE *db = fopen("user.database","w");

gen_write_tabulated( db, user_domain ) ;
gen_free_tabulated( user_domain ) ;
```

— Remarque : Attention au problème de partage
— Remarque : Tabulation automatique dans une future version de NewGen

Aspects Avancés
Importation

— Définition modulaire de spécifications NewGen
— Spécification multifichiers
— Complétude requise (mais voir *external*)

```
-- network.newgen

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

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

— `newgen` donne l'ordre pour `gen_read_spec`

```
% newgen -C network.newgen \
  workstation.newgen gateway.newgen
GEN_READ_SPEC order:
workstation.spec
gateway.spec
network.spec
```

%

Aspects Avancés Externes

— Compatibilité ascendante ("dusty data")
— Utilisation de NewGen en présence de données non-NewGen
— Contrainte : Compatible avec `char *` en C et pointeur en CommonLISP

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

output_device = laser + daisy + punch ;
```

— Routines de lecture, écriture, libération et copie à fournir par l'utilisateur
— `gen_init_external` à appeler avant toute utilisation.
— Définition de *DOMAIN* pour premier argument de `gen_init_external`

Retour sur l'exemple

— Tabulation des identificateurs
— Définition séparée de `identifier` :

```
-- File identifier.newgen

tabulated identifier = name:string ;
```

— Forme ASCII compacte `external`

```
-- File expression.newgen

import identifier from "identifier.newgen" ;
external compacted ;

binary = operator:string x
         lhs:expression x rhs:expression ;
binding = name:string x value:expression ;
expression = constant:int + identifier +
             compacted + binary + let ;
let = bindings:binding* x expression ;
```

— Appel de `newgen` :

```
% newgen -C expression.newgen \
  identifier.newgen
GEN_READ_SPEC order
identifier.spec
expression.spec
%
```

— Création des identificateurs :

```
    | Identifier {
        $$ = make_expression(
             is_expression_identifier,
             make_identifier( $1 )) ;
    }
```

— Initialisation de `compacted` dans `main` :

```
void compacted_write( FILE *, compacted ) ;
compacted compacted_read( FILE *,
                          char (*)()) ;
void compacted_free( compacted ) ;
compacted compacted_copy( compacted ) ;

main()
{
    gen_read_spec( "identifier.spec",
                   "expression.spec",
                   (char*) NULL) ;
    gen_init_external( COMPACTED,
                       compacted_read,
                       compacted_write,
                       compacted_free,
                       compacted_copy ) ;

    yyparse() ;
    fprintf( stdout, "%d\n",
             constant_fold( Top )) ;

#ifdef DEBUG
    fprintf( stderr,"Bound Identifiers:\n");
```

```
    TABULATED_MAP( i, {
        fprintf( stderr, "%s,",
                 identifier_name( i )) ;
    }, identifier_domain ) ;
#endif

    free_expression( Top ) ;
    gen_free_tabulated(identifier_domain);
}
```

— Support pour externes

```
void compacted_write( fd, c )
FILE *fd ;
compacted c ;
{
    int val = *(int *)(char *)c ;

    fprintf( fd, "%d",
             (int)log2( (double)val )) ;
}

compacted
compacted_read( fd, read )
FILE *fd ;
char (*read)() ;
{
```

```
    int *c = (int *)malloc( sizeof( int )) ;

    fscanf( fd, "%d", c ) ;
    return( (compacted)(char *)c ) ;
}

void
compacted_free( c )
compacted c ;
{
    free( c ) ;
}

compacted
compacted_copy( c )
compacted c ;
{
    int *cc = (int *)malloc( sizeof( int ));

    *cc = *c ;
    return( (compacted)(char *)cc ) ;
}
```

NewGen et CommonLISP

— Intérêt : Facilité de prototypage, développement, spécifications
— Permettre le développement “souple” : LISP fonctionnel, LISP impératif, C
— Intéropérabilité C/LISP limitée en général (*foreign function interface*)
— NewGen : pont entre deux mondes
— CommonLISP : *de facto* standard, plus norme ANSI en préparation
— Similitude de programmation (listes), mais GC
— Compatibilité “fichiers” ou pipes

Changements

— Type NewGen : `defstruct`
— Adaptation à la syntaxe CommonLISP :

```
(setf pierre
      (make-user
         :name "jouvelot"
         :id 110
         :passwd passwd-undefined
         :shell "/usr/local/bin/ksh"))
```

— Modification via `setf` :

```
(setf (user-id pierre) 120)
```

— Le `switch` de C est définit comme une macro :

```
(gen-switch (expression-tag e)
        (is-expression-constant
           (expression-constant e))
        (:default
           (error "~%Incorrect tag")))
```

— `gen-switch` peut aussi créer des liaisons :

```
(gen-switch (expression-tag e)
        ((is-expression-constant c) c)
        (:default
          (error "~%Incorrect tag")))
```

— Pas de libération explicite (sauf pour domaines tabulés)
— Visibilité des fonctions de manipulation via `use-package`

Evaluateur pour SIMPLE

— Création des fichiers Lisp :

```
% newgen -lisp expression.newgen \
  identifier.newgen
REQUIRE order:
identifier.cl
expression.cl
% ls
expression.cl   expression.spec
identifier.cl   identifier.spec
%
```

— `require` pour chargement des fichiers

— Pas d'arguments à `gen-read-spec` : auto-initialisation des fichiers CommonLISP

Top Level

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

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

(defun test (&optional (file *standard-input*))
  "FILE contains the parser output."
  (gen-read-spec)
  (let ((*standard-input* (open file)))
    (eval-expression (read-expression) '())))
```

Boucle d'évaluation

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

(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))))

(defparameter operators
  `((,"add" . ,\#'+)
    (,"sub" . ,\#'-)
```

```
    (,"times" . ,\#'*)
    (,"cons" . ,\#'cons)
    (,"eq" . ,\#'eq)))

(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)))))

(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))))
```

Aspects avancés
Exemples

— `gen-recurse` : Couplage appels récursifs et dispatch :

```
(defun eval-expression (e env)
    (gen-recurse e
        ((expression tag) tag)
        (identifier
          (cdr (assoc
                   (identifier-name i)
                   env
                   :test #'string-equal)))
        ((binary lhs rhs)
          (funcall
            (cdr (assoc
                    (binary-operator b)
                    operators
                    :test #'string-equal))
              lhs rhs))))
```

— Opérations implicitement itérées (sur listes)

— Utilisation des domaines tabulés :

```
(defun gensym ()
  "Generate a brand new identifier."
  (do ((i 0 (+ i 1)))
      ((gen-find-tabulated
           (format nil "gensym-~D" i)
           identifier-domain)
       (make-identifier
           :name (format nil "gensym-~D"
                         i)))))
```

Aspects de l'implémentation

— Outil "léger" :
  — 6 klignes de C, Yacc, Lex et Korn shell
  — 800 lignes de CommonLISP
— Compilateur :

  token.l Lexèmes du langage NewGen

  gram.y Syntaxe du langage NewGen

  build.c Compilation en fichier spec, création dynamique des descripteurs de domaines

  genC.c, genLisp.c Génération de code C et Lisp

  newgen Commande shell

— Run time C/Lisp :

  genClib.c,genLisplib.cl Bibliothèque run-time C et Lisp

  list.c Support de listes en C

  set.c,set.cl Support d'ensembles en C et Lisp

  hash.c Package de hash-coding dynamique (interne et externe – set) par *open coding*

read.l,read.y Parser C de structures de données NewGen(utilisation de macros en Lisp)

— Structure mémoire taggée, avec inlining
— Vérification dynamique de types (`gen_debug`)
— Parcours générique parallèle descripteurs/structures : gestion de partage, écriture, copie, libération
— Un mot supplémentaire pour objets tabulés
— Ecriture compacte sur disque (mais pas binaire)

PIPS : Un exemple “vraie grandeur”

— PIPS : Paralléliseur Interprocédural de Programmes Scientifiques
— Tranformation DO en DOALL (Fortran77)
— Projet de recherche : structure modulaire en phases (50 klignes)
— Prise en compte complète de Fortran :

```
-- Entities

tabulated entity = name:string x type x
                   value x storage ;

-- Expressions

expression = reference + range + call ;
reference = variable:entity x
            indices:expression* ;
range = lower:expression x
        upper:expression x
        increment:expression ;
call = function:entity x
```

```
        arguments:expression* ;

-- Statements

statement = label:entity x
            number:int x
            comments:string x
            instruction ;

instruction = block:statement* + test +
              loop + call + unstructured ;
test = condition:expression x
       true:statement x
       false:statement ;
loop = index:entity x
       range x
       body:statement x
       label:entity ;

unstructured = control x exit:control ;
control = statement x
          predecessors:control* x
          successors:control* ;
```

— Gestion de la persistence par `pipsdbm`

— Prototypes du linker incrémental, prettyprinter

et détection des réductions en CommonLISP

Autres systèmes

- IDL
  - Outil logiciel (North-Carolina U., développé chez Tartan)
  - Génération de structures de données (C, Pascal)
  - Description des phases (*processes*) et des interconnexions
  - GC
  - Forme limitée de sous-typage
  - Assertions (définition d'un langage complet d'assertions)
  - Format binaire
  - Conclusion : industriel, moins abstrait, plus lourd
- OODB (O2, ORION, VBASE, Exodus, Postgres)
  - Extension des modèles orienté-objet aux DB (limitation du modèle relationnel)
  - Manipulation et langage de requêtes intégrés dans un langage classique (CO2, Common-LISP, C, C++)

— Orienté accès interactif (SQL)
— Conclusion : plus puissant que NewGen, accès coûteux (persistence implicite)

— OOL (C++, CLOS, Smalltalk, Trellis)
  — Dépendent d'un langage, pas *upward* compatible (sauf C++)
  — Plus puissant que NewGen : héritage, redéfinition
  — Performances ?
  — Pas de persistence
— Langages Persistants
  — Nécessite des modifications de compilateurs (Pascal/P, PS-algol)
  — Pas de standard dans les primitives
— RPC, XDR
  — Bas niveau (orienté transferts de données)
  — Pas de gestion de sharing ou de cycle

Conclusion

— NewGen : outil de génie logiciel (Ecole des Mines de Paris)
— Abstraction fonctionnelle, Multi-langages (C, CommonLISP), Compatibilité
— Prototype "Public Domain" distribué par ftp anonyme (Mines, Bull, NASA, Boeing, ...)
— Applications : PIPS, PMACS (Bull)
— Futur : extensions aux fonctions (tabulation automatique) :

```
typing = expression -> type ;
```