SETL:
Un langage de très haut niveau pour le prototypage.

J.P. Rosen
ENST
46, rue Barrault
F-75634 Paris CEDEX 13

Résumé:
Le développement de logiciels par la méthode de prototypage apparait de plus en plus comme un puissant outil des nouvelles méthodes de génie logiciel. Le langage SETL, qui s'appuie sur le formalisme de la théorie des ensembles, est bien adapté aux nécessités de l'écriture de prototypes, grâce a sa puissance d'expression et au niveau très abstrait des types de données qu'il manipule. La validité de SETL en temps que langage de prototypage a éte démontrée dans le projet Ada/ED, premier compilateur Ada validé, qui fut développé avec des moyens beaucoup plus faibles que les autres compilateurs Ada.

I. Le développement de logiciel par prototypage.

Une des choses les plus remarquables dans l'émergence actuelle du génie logiciel est que l'on redécouvre que les méthodes traditionnelles de l'industrie sont applicables (et même nécessaires) au développement de logiciels. Ainsi, pour le développement d'une carte électronique, on commencera par faire une maquette, wrappée par exemple, afin de tester la validité du schéma électronique; la technique de fabrication de cette maquette permettra d'apporter facilement des modifications au prototype au fur et à mesure que des problèmes apparaissent; par contre, certains problèmes (résistance mécanique du circuit par exemple) ne sont pas importants à ce stade du développement. Ce n'est qu'une fois le schéma électronique stabilisé et validé que l'on passera à la phase d'industrialisation.

La méthode de développement de logiciel par prototypage utilise la même approche: construire d'abord une maquette du futur logiciel permettant de se rendre compte de la validité des spécifications et du comportement du futur programme. Puis, une fois tous les problèmes connus (et résolus), réécrire le programme définitif. Si la maquette doit permettre d'exprimer la totalité des spécifications, son but est essentiellement d'acquérir l'expérience nécessaire à l'écriture du programme final; les questions d'efficacité, aussi bien en temps de calcul qu'en espace mémoire, n'ont aucune importance à ce niveau.

Du fait des contraintes totalement différentes entre la maquette et le produit final, il apparaît logique d'utiliser des langages différents; en particulier, un langage de prototypage doit avoir les qualités suivantes:

- *Permettre d'écrire rapidement un algorithme.* La notion même de prototypage impliquant que le coût de la maquette soit faible devant celui du modèle définitif.
- *Permettre une modification aisée du programme.* Il ne s'agit pas là de la simple réécriture de quelques lignes de programme, mais de la possibilité de revenir à n'importe quel moment sur des choix de stratégie d'implémentation sans remettre en cause l'ensemble du programme.
- *Affranchir le programmeur des contraintes de bas niveau.* Bien souvent, le choix du mode de représentation des données optimal dépend de contraintes extérieures, éventuellement liées à la structure de la machine hôte, et n'influe pas sur la logique de la méthode.

Nous résumerons l'ensemble de ces contraintes en disant qu'un langage de prototypage doit être un langage de *très haut niveau*, c'est à dire permettant d'exprimer un algorithme d'une façon aussi proche que possible de l'expression naturelle, indépendemment des contraintes dues aux caractéristiques de la machine.

II. Le langage SETL.

SETL (dont le nom signifie SET Language) est un langage qui a été développé à New York University par une équipe constituée de J. Schwartz, R. Dewar, E. Schonberg et D. Shields. L'idée directrice de ce langage était d'utiliser un formalisme dérivé du formalisme mathématique, et principalement de la théorie des ensembles, pour exprimer des algorithmes.

Le langage SETL est un langage séquentiel impératif: un programme se présente comme une suite d'instructions qui sont exécutées séquentiellement. Voici par exemple un programme qui demande l'époque de la journée et répond en conséquence:

```
program Exemple;

    Print('Quelle époque de la journée sommes-nous?');
    Get('INPUT', Reponse);
    case Reponse of
        ('matin'):  Print('Bonjour');
        ('midi'):   Print('Bonne journée');
        ('soir'):   Print('Bonsoir');
        else        Print('Desole, je n''ai pas compris');
    end case;

end program Exemple;
```

La déclaration des variables n'est pas obligatoire en SETL, car dans l'élaboration d'un prototype, les variables sont généralement très instables et changent souvent. De

plus les variables ne sont pas typées: elles peuvent recevoir n'importe quelle valeur à n'importe quel moment. Par contre, les opérations ne sont définies qu'entre des valeurs de types précis: il n'est pas possible par exemple d'ajouter un entier à un flottant. En quelque sorte, la notion de type est en SETL attachée aux valeurs (ou au contenu) et non aux variables (ou au contenant).

SETL fournit les types usuels: entiers, flottants, caractères et chaînes, booléens. Mais les types principaux du langage sont les ensembles (*SET*), les N-uplets (*TUPLES*) et les correspondances (*MAP*). Il existe également une valeur spéciale, appelée OM (pour Oméga), qui signifie *valeur indéfinie*. Toutes les variables sont initialisées à OM.

Un ensemble est une structure de donnée qui peut contenir n'importe quel type de valeur. Tous les éléments d'un ensemble n'ont pas nécessairement le même type. Une valeur peut figurer au plus une fois dans un ensemble, et les différentes valeurs ne sont pas ordonnées. Les valeurs de type ensemble sont construites par une énumération de valeurs ou d'intervalles (notés A..B) entre accolades, ou au moyen d'*itérateurs* qui seront décrits un peu plus loin.

Un TUPLE est une liste ordonnée de valeurs de type quelconque. Une même valeur peut figurer plusieurs fois (à des emplacements différents), et il est possible d'accéder aux différents éléments d'un tuple par indexation (à partir de 1). La notation T[A..B] correspond à la tranche du tuple T depuis l'élément d'indice A jusqu'à celui d'indice B. La notation T[A..] correspond à la tranche du tuple T depuis l'élément A jusqu'au dernier.

Un tuple est virtuellement infini: toute tentative d'accès à un élément auquel aucune valeur n'a été affectée, en particulier au delà du dernier élément utilisé, renverra la valeur OM. Les valeurs de type tuple sont construites par énumération de valeurs ou d'intervalles (notés A..B) entre crochets, ou au moyen d'*itérateurs* qui seront décrits un peu plus loin.

Une MAP est un ensemble constitué uniquement de tuples à deux éléments. Elle représente une correspondance entre l'élément donné comme premier élément de chaque tuple et celui donné comme deuxième élément. Une map étant un cas particulier d'ensemble se construit comme n'importe quel ensemble. On accède à un élément d'une map en l'"indexant" par une valeur de l'ensemble de départ. Si la valeur d'index n'apparait pas dans l'ensemble de départ de la relation, la valeur retournée est OM. Noter que du point de vue formel, un tuple se comporte comme une map dont l'ensemble de départ est l'ensemble des entiers strictement positifs.

Un *itérateur* sert à décrire une liste de valeur prises succesivement par un *membre gauche* (c'est à dire une structure pouvant apparaître à gauche d'un signe d'affectation). La forme générale d'un itérateur est la suivante:

membre_gauche **in** *structure* [| *condition*]

Le membre gauche prend successivement toutes les valeurs contenues dans la structure pour lesquels la condition (si elle est présente) est vérifiée. La barre verticale se lit *telle que*. Noter que puisqu'un ensemble n'est pas ordonné, l'ordre des valeurs renvoyées par un itérateur sur un ensemble est non déterministe.

Il existe deux predicats liés à la notion d'itérateurs: **exists** et **forall**. Exists revoie vrai (TRUE) s'il existe au moins une valeur de *membre_gauche* verifiant la condition. *membre_gauche* prend alors cette valeur. Forall renvoie vrai si tous les *membres_gauches* de la structure verifient la condition.

Exemples:

L'ensemble des nombres pairs entre 1 et 100:

(x **in** [1..100] **|** x **mod** 2 /= 0)

La liste des nombres premiers entre 1 et 100:

[x **in** [1..100] **| not exists** y **in** [2..x-1] **|** x **mod** y = 0]

La liste des nombres entre 1 et 100 dont tous les diviseurs sont inférieurs à 10:

[x **in** [1..100] **| forall** y **in** {z **in** [2..x-1] **|** x **mod** z = 0 } **|** y < 10]

Lorsque les valeurs servant à construire la structure ne sont pas directement les valeurs renvoyées par l'itérateur, il est possible d'utiliser une formule quelconque utilisant (ou non!) la valeur renvoyée par l'itérateur. Le constructeur de structure prend alors la forme:

expression : *itérateur*

Exemple:

Une correspondance entre des nombres entre 1 et 100 et leur carre:

([x, x**2] : x **in** [1..100])

Les différentes opérations sur ensembles et tuples sont résumées dans les tableaux suivants. S, S1, S2 représentent des ensembles, T, T1, T2 représentent des tuples, M une map, E un élément quelconque. A titre d'exemple, on donne également la définition en SETL de quelques unes de ces opérations:

	Opérations:		
+	Union d'ensembles	S1 + S2	
+	Concaténation de tuples	T1 + T2	
-	Différence d'ensembles	S1 - S2	{x **in** S1 **| not exists** y **in** S2 **|** y = x }
*	Intersection d'ensembles	S1 * S2	{x **in** S1 **| exists** y **in** S2 **|** y = x }
#	Nombre d'éléments	#S #T	
power	Puissance d'ensemble	**power** S	
with	Ajout d'élément	S **with** E	S + {E}
less	Retrait d'élément	S **less** E	S - {E}
arb	Element arbitraire	**arb** S	
domain	Ensemble de départ	**domain** M	{x : [x, -] **in** M }
range	Ensemble d'arrivée	**range** M	{x : [-, x] **in** M }
	Prédicats:		
in	Est membre	E **in** S	**exists** y **in** S **|** y = E
notin	N'est pas membre	E **notin** S	**not** (E **in** S)
subset	Est un sous-ensemble	S1 **subset** S2	**forall** x **in** S1 **|** x **in** S2
OK	Backtracking	**OK**	

Le predicat **OK** merite quelques precisions: sa semantique est qu'il repond True si le fait de repondre True amène à la solution du problème; autrement il repond False. Il est donc en quelque sorte capable de prévoir l'avenir! En fait, il commence par preserver l'état du programme, et renvoie la valeur True. Si le programme exécute par la suite une instruction **failed**, l'état du programme est restaure tel que lors du dernier appel à **OK**, et **OK** renvoie False. Ce mécanisme qui offre à l'utilisateur un système de backtracking automatique est bien entendu extrêmement coûteux dans la pratique. Pour économiser de l'espace, le programme peut exécuter l'instruction **succeed** qui permet d'affirmer que l'on ne faillira pas par la suite, et donc de libérer l'espace associé au dernier **OK**. Malgré tout, préserver *tout* le contexte d'un programme est souvent abusif, aussi les implémentations obligent-elles généralement l'utilisateur à déclarer les variables qui seront à préserver en cas de backtracking.

Opérateurs d'affectation:

:=	Affectation		
from	Retrait d'un ensemble	E **from** S	E := **arb** S; S := S **less** E;
fromb	Retrait premier élément	E **fromb** T	E := T[1]; T := T[2..];
frome	Retrait dernier élément	E **frome** T	E := T[#T]; T := T[1..#T-1];

L'élément à gauche d'une affectation (appelé membre gauche) est généralement une variable. Dans le cas où l'expression de droite est un tuple, ce peut être également un tuple composé uniquement de membres gauches où du signe - . Dans ce cas, l'affectation est multiple par correspondance d'éléments, les positions marquées à gauche par un signe - n'étant pas affectées. Ainsi: [A,B,C] := [10,20,30]; affecte 10 à A, 20 à B, et 30 à C. Cette possibilité offre une excellente lisibilité en permettant d'exprimer par l'affectation la structure attendue des données:

[Selecteur, [Elem_1, Elem_2], Elem_3] := Struct;

Il est ainsi possible de "déshabiller" une structure complexe en affectant simplement ses éléments intéressant à des variables locales.

Il est également possible de faire précéder l'opérateur d'affectation d'un opérateur binaire (comme en C). Ainsi, I +:= 1; est équivalent à I := I+1;

SETL dispose bien entendu des instructions de structuration classiques:

Instruction si:

if *condition* **then** *Liste d'instructions*
(**elseif** *condition* **then** *Liste d'instructions*)
[**else** *Liste d'instructions*]
end if;

Instruction cas:

case *expression* **of**
((*Liste de valeurs*) : *Liste d'instructions*)
[**else** *Liste d'instructions*]
end case;

Remarquer que conformément à la philosophie de SETL, il n'y a aucune contrainte quant au type de l'expression d'une instruction cas: celle-ci peut parfaitement être une chaine de caractères, une variable structurée... De même les

listes de valeurs peuvent être des constantes ensemble, tuple, etc. Il n'est pas nécessaire que toutes les valeurs aient le même type, puisqu'une même variable (utilisée par exemple comme expression) peut contenir des valeurs de types différents.

Instructions de boucle:

loop do
Liste d'instructions
end loop;

loop while *condition* **do**
Liste d'instructions
end loop;

loop for *itérateur* **do**
Liste d'instructions
end loop;

Il ne s'agit là que des trois formes les plus courantes de boucle, d'autres syntaxes existent pour exprimer d'autres formes moins classiques. Noter la possibilité d'un itérateur *quelconque*:

loop for I **in** [1..10] **do** _
loop for x **in** [1..100] **| not exists** y **in** [2..x-1] **|** x **mod** y = 0 **do** _
loop for [x, y] **in** Correspondance **do** _

SETL dispose de procédures, mais qui ne peuvent être imbriquées. Une procédure peut sortir en retournant une valeur au moyen de l'instruction **return** *valeur*, ou sans retourner de valeur au moyen de **return** seul. La différence entre fonctions et procédures vient donc purement de l'utilisation qui en est faite. Comme en C, une fonction peut être appelée comme procédure, et réciproquement (auquel cas la valeur retournée est OM). Les procédures sont bien entendu récursives.

SETL dispose d'une possibilité de macro permettant de cacher des séquences utilisées fréquemment sous un nom commun. Cette possibilité permet de faciliter grandement le passage au modèle définitif en spécifiant de façon claire des fonctionnalités tellement simples à réaliser qu'elles ne méritent pas un sous-programme, mais qui pourront donner lieu à un sous-programme lors de la traduction dans le langage d'implémentation.

SETL dispose également de nombreuses procédures prédéfinies permettant de faire des entrées-sorties, des manipulations de chaînes de caractères, etc.

Bien entendu, ce rapide survol de SETL ne prétend pas présenter tous les outils disponibles; le lecteur intéressé est renvoyé à la bibliographie (en particulier [DEW 82]) pour une description plus complète du langage. Mais de l'avis général de ceux qui ont utilisé le langage, on peut dire que SETL est un langage agréable à utiliser; le formalisme mathématique permet d'exprimer en termes naturels des conditions parfois compliquées; et l'affranchissement total du programmeur des contraintes liées aux représentations de données lui permet effectivement de se concentrer sur les problèmes de définition, sans être gêné par les contraintes d'implémentation.

III. Exemples d'utilisation

Nous allons maintenant donner quelques exemples d'utilisation du langage pour essayer de donner un idée du "style SETL" et de sa facilite d'emploi.

Macros de manipulation de pile:

```
macro Push(x);
    Pile with:= x
endm;
macro Pop(x);
    x frome Pile
endm;
macro TOS(x);                          $ Top Of Stack
    Pile[#Pile]
endm;
```

Utilisation d'enregistrements variants:

On suppose que Instr est un tuple décrivant une instruction d'un langage (Pascal par exemple). Ceci pourrait être une partie de la procédure Execute chargée d'interpréter ce langage:

```
[Type_instr] := Instr;                 $ Récupérer le sélecteur
case Type_instr of
('If'):     [-, Condition, Then_part, Else_part] := Instr;
            if Eval(Condition) = True then     $ Eval est l'evaluateur d'expression
                Execute(Then_part);
            elseif Else_part /= OM then
                Execute(Else_part);
            end if;
('While'):  [-, Condition, Statements] := Instr;
            loop while Eval(Condition) = True do
                Execute(Statements);
            end loop;
('Block'):  Instruction_list := Instr[2..];
            loop for Statement in Instruction_list do
                Execute(Statement);
            end loop;
etc...
end case;
```

Tuples en partie gauche ou droite:

Supposons qu'à un identificateur soient associées un type, une adresse, et une valeur appelée profondeur (d'imbrication statique par exemple). Nous pouvons réaliser une table des symboles au moyen de trois maps faisant correspondre à la chaine de caractères représentant le nom d'un symbole les données correspondantes. Si l'on souhaite manipuler globalement toutes les informations relatives à un symbole, il suffit de définir la macro suivante:

```
macro SYMB_TAB(x);
    [ TYP[x], ADR[x], PROF[x] ]
endm;
```

Cette macro pourra aussi bien être utilisée en partie gauche que droite:

```
[T, A, P ]              := SYMB_TAB('Toto');
SYMB_TAB('Toto')        := ['Integer', 1234, 4];
```

Il est ainsi possible de cacher l'aspect "discontinu" de la table des symboles.

Un grand classique: le problème des huit reines:

```
program Huit_reines;
var  occupation,                          $ Note les lignes, colonnes, diagonales prises
     coordonnees : BACK;                  $ Positions des reines.
                                          $ Ces variables sont "backtrackées"

ligne       := 1;
colonne     := 1;
occupation      := {};
coordonnees  := {};

loop for colonne in [1..8] do
     if OK then                           $ Si cette position mene à la solution...
          mettre(ligne, colonne);         $ ... mettons une dame à cette position
          print(coordonnees);             $ Imprimons la solution
          quit;                           $ Sortie de boucle : on s'arrete à 1 solution
     end if;
end loop;

procedure mettre(L,C);
if   ['ligne', L]    in  occupation or
     ['colonne', C] in  occupation or
     ['diag1', L+C] in  occupation or
     ['diag2', L-C] in  occupation
then
     fail;                                $ La case est en prise
else
     coordonnees  with:= [L, C];          $ Noter les coordonnees de la reine
     occupation        +:= ( ['ligne', L], ['colonne', C], ['diag1', L+C], ['diag2',L-C] );
                                          $ Noter les cases en prise
     if L < 8  then                       $ Si pas denière case,
          loop for new_col in [1..8] do   $ mettre la reine suivante
               if OK then
                    mettre(L+1, new_col);
                    return;
               end if;
          end loop;
          fail;                           $ Car impossible de mettre la reine suivante
     end if;
end if;
end procedure mettre;

end program Huit_reines;
```

IV. Du prototype au programme définitif.

Faire un prototype d'une application est bien; mais il faut ensuite passer à l'application définitive. Or le "fosse sémantique" entre un langage tel que SETL et un langage d'implémentation tel que C par exemple est important, et il peut être nécessaire de produire un prototype intermédiaire pour faciliter cette transition.

Ce produit intermediaire peut lui-même être écrit en SETL, puisque le langage dispose des structures classiques des langages de plus bas niveau. On se restreindra cependant à un sous-ensemble du langage, en évitant par exemple les constructeurs de structure trop élaborés au profit d'une écriture plus classique, avec des boucles explicites par exemple. Bien sûr, cet abaissement du niveau sémantique des structures de programme doit s'accompagner d'une redéfinition correspondante des structures de données.

Pour cela, SETL offre la possibilité d'un *typage progressif* des structures de données [DEW 79]. Tout d'abord, il est possible de déclarer des variables, et une option du compilateur permet d'obtenir la liste des variables non déclarées: ceci permet de "faire le ménage" dans l'utilisation des variables. Ensuite, des clauses spéciales de représentation permettent de limiter les classes d'objet que l'on peut affecter à chaque variable, avec plus ou moins de précision. Imaginons que nous ayons besoin d'une correspondance entre deux types d'objets A et B, dont la représentation définitive est encore à définir. Nous pouvons spécifier:

```
repr
    Corr_A_B : map;
end repr;
```

Par la suite, nous pouvons arriver à la conclusion que nous souhaitons représenter les objets de type A par des chaines de caractères. Nous pourrons alors préciser:

```
repr
    Corr_A_B : map(string);
end repr;
```

Puis nous pouvons décider que les objets de type B sont des entiers. Nous spécifierons alors:

```
repr
    Corr_A_B : map(string) Integer;
end repr;
```

Bien sûr, ce raffinement dans les spécifications permettra au compilateur de rajouter des tests qui diagnostiqueront (statiquement et dynamiquement) la cohérence de l'utilisation qui est faite des variables avec leur représentation. Il est ainsi possible de passer par recodage progressif, éventuellement module par module, d'une description de haut niveau très abstraite à une représentation fortement typée et proche des possibilités du langage d'implémentation. Si de plus l'on a pris soin de cacher dans des macros les points sensibles de l'application (accès à la table des symboles par exemple), l'écriture du programme définitif se réduira à une simple

traduction de syntaxe, qui peut d'ailleurs être fortement automatisée, plus le remplacement de macros par des appels de sous-programmes exécutant les fonctionnalités correspondantes.

V. Un exemple d'utilisation de SETL comme langage de prototypage.

Cette approche de l'élaboration d'une application par prototypage et raffinements successifs a été suivie pour l'écriture par l'équipe de New York University du compilateur Ada/ED, qui fut le premier compilateur Ada validé officiellement par le DoD (département américain de la défense) [DOD 83], [KRU 84]. En fait, l'équipe avait un contrat avec l'armée américaine pour produire le plus vite possible un compilateur opérationnel, afin de permettre la mise au point de la suite de validation elle-même. Aucune contrainte de performance n'était par contre requise, aussi l'approche par prototypage apparaissait-elle comme une solution évidente.

L'intérêt de ce choix est amplement prouvé par le rapprochement de quelques dates: la norme Ada ANSI a été publiée au mois de Janvier 1983, et des changements sont intervenus jusqu'à fin Décembre 1982. Pourtant, Ada/ED était validé dès le mois d'Avril 1983, par une équipe comptant alors quatre membres permanents (qui avaient de surcroit des charges universitaires) et quelques étudiants. Une telle performance n'a bien entendu été possible que par le très haut niveau de la description utilisée. L'arbre syntaxique produit par le frontal était directement exécuté par un interprète. Il n'y avait pas de mémoire! Une map faisait la correspondance entre le nom unique d'une variable et sa valeur. Bien sûr, le coût de ce modèle de haut niveau était important: Ada/ED exécutait environ une ou deux instructions Ada par seconde d'unité centrale sur un Vax 780 ...

Le texte de cette première version du système a été publié et est disponible publiquement [NYU 83a], [NYU 83b]. Il permet de se faire une bonne idée de ce que l'on peut attendre de SETL en utilisation réelle.

Il est apparu rapidement que cette description ne permettrait pas d'obtenir directement un produit utilisable. Le choix a donc été fait d'abaisser le niveau sémantique de l'interprète en le transformant en interprète de machine virtuelle classique (type P-machine), moyennant l'adjonction au frontal d'un générateur de code. Le générateur de code et l'interprète de machine virtuelle ont pu récupérer une large part du code écrit pour l'interprète de haut niveau. En même temps, une autre partie de l'équipe reprenait le frontal pour le faire travailler sur des structures de données de plus bas niveau, mais mieux adaptées au futur recodage prévu en C. Les structures transmises entre le frontal et les différentes parties arrières étaient cependant les mêmes qu'avec l'ancien système, ce qui permit une mise au point indépendante des deux parties, une équipe travaillant avec le nouveau frontal + l'ancien interprète, l'autre avec l'ancien frontal + le générateur de code. Lorsque les deux parties furent suffisamment mûres, la connexion nouveau frontal/générateur de code s'effectua sans problème majeur.

Finalement, lorsque le système fut estimé être mis au point à 95% en SETL, il fut décidé de passer au codage définitif en C. L'abandon du prototype SETL fut décidé au début de Février 1985. A l'heure où ces lignes sont écrites (Juin 1985), il est prévu de valider le modèle définitif dans le courant de l'été, sur Vax/VMS et sur IBM/PC,

Ada/ED devant être ainsi le premier compilateur sur IBM/PC. L'equipe du projet ne compte toujours que quatre membres permanents et quelques etudiants...

Finalement, la validité de l'approche par prototypage (et de l'utilisation de SETL pour cela) se mesure au rendement de programmation. La première version du système a demandé, du début à la première validation, environ cent hommes-mois. L'ensemble generateur de code + interprète de machine virtuelle aura demandé environ trois hommes-ans pour 30000 lignes de SETL, plus quelques hommes-mois pour le recodage en C. Au total, l'effort en hommes-ans représenté par l'écriture du système Ada/ED peut être estimé à environ un dixième de l'effort nécessité par les autres compilateurs Ada sur le marché.

Une anecdote montrera à quel point il peut être dangereux de décider d'une structure de données avant que tous les problèmes soient connus. Il est arrivé qu'au cours du développement d'Ada/ED il soit nécessaire de rajouter un champ dans la table des symboles. Celle-ci était faite de plusieurs maps établissant les correspondances nom d'objet vers propriété. Il suffit donc de déclarer une variable globale de plus, de l'initialiser, et de rajouter son nom dans les macros faisant la lecture et l'écriture de la table des symboles sur fichier. Temps approximatif de la modification: 1/2 heure. Erreurs induites par cette modification: néant.

Le même problème survint récemment dans un compilateur développé par une célèbre entreprise dédiée à la production de produits Ada. Il fallut plus d'un mois pour rattraper les conséquences du rajout d'un champ dans la table des symboles...

VI. Conclusion

Le développement d'un logiciel par prototypage est un moyen efficace de faire baisser les coûts de production de logiciels. Le langage SETL est bien adapté à l'écriture de prototypes grâces à ces structures de haut niveau permettant de décrire un algorithme tout en s'affranchissant des problèmes liés à des contraintes de bas niveau telles que les représentations de données. Il offre cependant des possibilités de typage progressif permettant de passer progressivement à une description de plus bas niveau, facilitant ainsi la transition vers le modèle définitif.

L'exemple du compilateur Ada/ED montre la validité de cette approche dans la pratique, sur une application de taille industrielle.

Cependant, le langage est assez ancien, et, à l'utilisation, nombre d'imperfections sont apparues, au niveau du langage bien sûr, mais aussi au niveau du compilateur SETL. Celui-ci est de plus écrit dans un langage d'implémentation qui lui est propre, ce qui nécessite un certain effort pour le porter sur une autre machine. Pour ces raisons, le langage SETL va être redéfini, et le compilateur sera entièrement réécrit. Par un juste retour des choses, il sera recodé en Ada...

VII. Bibliographie.

[DEW 79] R.B.K. Dewar, A. Grand, S.C. Liu, J.T. Schwartz et E. Schonberg: *Programming by refinement, as exemplified by the SETL representation sublanguage,* ACM Transactions on Programming Languages and Systems, Vol 1, n°1, Juillet 79, pp 27-49.

[DEW 82] R.B.K. Dewar, E. Schonberg, J.T. Schwartz: *Higher level Programming - An introduction to the Programming Language SETL,* Courant Institute of Mathematical Sciences, Juillet 1982.

[DOD 83] United States Department Of Defense: *Reference manual for the Ada programming language,* ANSI/MIL-STD 1815 A.

[KRU 84] P. Kruchten, E. Schonberg: *Le système Ada/ED: une experience de prototype de logiciel utilisant le langage SETL,* TSI, vol 3, n°3, Mai-Juin 1984, pp 193-200.

[NYU 83a] NYUADA: *Semantic actions for Ada,* Courant Institute of Mathematical Sciences, Technical Report n°84, 1983.

[NYU 83b] NYUADA: *An executable semantic Model of Ada,* Courant Institute of Mathematical Sciences, Technical Report n°85, 1983.

[SCH 81] E. Schonberg, J.T. Schwartz, M. Sharir: *An Automatic Technique for Selection of Data Representation in SETL Programs,* ACM Transactions on Programming Languages and Systems, Vol 3, n°2, Avril 81, pp 126-143.