Tester les compilateurs

Louis Frécon

Cet article résume l'expérience acquise essentiellement de 1965 à 1995 dans l'ensei–gnement de langages nouveaux, souvent basé sur l'emploi de compilateurs expérimentaux, et comme chef de projet en méta-compilation et compilation lourde (compilateurs PL1/C[1], Ampère 2) comme en conception de langages et de leur environnement.

Pourquoi ?

Les compilateurs, ou plus généralement les environnements de programmation, en tant qu'œuvre humaine, ont nécessairement des défauts : selon des évaluations faites en 1965 et confirmées en 1985, la version 0 d'un programme comporte en général 55 à 65 erreurs par millier de lignes. Même si les compilateurs sont en principe de grande qualité, même s'ils ont été développés à l'aide d'outils garantissant une haute fiabilité, *ils ne sauraient être parfaits, alors même que chacun les utilise en toute confiance.*

La question se pose essentiellement dans le cadre de la recette d'une nouvelle voire première version, et en exploitation lorsqu'une anomalie est détectée, le but étant de vérifier qu'au sens d'un langage donné L :

- tout programme correct au sens de L est correctement exécuté,
- tout programme incorrect au sens de L est correctement détecté, diagnostiqué et rejeté.

Recette d'une nouvelle version

Cette version est-elle acceptable ?

En général, une version récente a surtout été testée sur ce que les compilistes[2] ont considéré comme points sensibles – et plus légèrement ailleurs.

Pour une première ou nouvelle version, la question se pose donc d'évaluer la conformité du produit à la définition officielle du langage L (rapport de définition, standard ou norme), éventuellement modifiée dans un manuel de référence du produit, explicitant restrictions, limitations et extensions du langage traité L' par rapport à la définition officielle de L.

Une première version demandera un jeu d'essais, formé d'une suite de programmes de complexité croissante, comportant

- des programmes simples d'usage général, parfois calqués sur des programmes usuels écrits dans d'autres langages,
- des programmes simples mettant en valeur les spécificités du langage,
- peut-être un ou deux programmes plus lourds.

Pour d'autres versions, on est en droit d'espérer une *évolution non-régressive* : le nouveau jeu d'essais sera formé du jeu d'essais précédent, enrichi de programmes relatifs aux différences annoncées, testant en priorité les limitations et restrictions supprimées ou réduites.

Certification

La certification d'un compilateur est un test assez similaire au précédent, les rapports avec les auteurs étant plus distants.

---

[1] PL1 peut être considéré comme un ancêtre de C, étendu et puissant (il devait remplacer Fortran, Cobol, Algol 60... dans les applications les plus variées) mais fermé. C, « PL1 du pauvre », compense sa minceur initiale par son auto-extensibilité, et la symbiose C/unix.

[2] ceux qui réalisent puis maintiennent le compilateur

Au-delà des premières certifications, le train d'essais sera grosso modo constitué. Cependant, toute nouvelle campagne de test peut manifester des failles amenant à étendre le train d'essais. Ces trains d'essais peuvent devenir considérables : avant même d'aborder les tests « temps réel », le train d'essais Ada 83 comportait plus de 2 000 programmes.

Dépannage/Maintenance

En exploitation, on peut constater diverses anomalies — en principe, de plus en plus rares, et pouvant apparaître dans des situations complexes.

Pour bien remonter de l'anomalie à sa cause, il faut d'abord cerner cette anomalie au plus juste, en simplifiant progressivement l'application révélatrice jusqu'à *manifester cette anomalie dans le plus simple contexte possible.* Cette opération est parfois difficile, si l'anomalie constatée n'est qu'une conséquence plus ou moins lointaine de l'erreur réelle, qu'il s'agira de traquer.

Enseignements

L'analyse des rapports d'anomalie, le test de compilateurs, l'enquête sur l'origine des défauts constatés se révèlent très instructifs en ce qui concerne les langages, leurs formalismes de définitions plus ou moins adéquats, la théorie sous-jacente, la distance entre programme (formellement) correct et programme valide, les dilemmes liberté/sécurité. Elle peut également donner des indications sur l'intérêt et les failles de diverses techniques de compilation.

Ces activités permettent aux concepteurs de langages et aux compilistes de mieux appréhender les forces et faiblesses des langages antérieurs, de leurs formalismes, et des techniques employées pour leur compilation.

Comment ?

Le but est donc de vérifier au sens du langage L (L') considéré stricto sensu

- que tout programme correct est correctement exécuté[3],
- que tout programme incorrect est proprement détecté.

Cette activité suppose une attitude légaliste.

De fait, tout programme est un signifiant (texte) porteur d'un signifié (algorithme). Son exécution correcte suppose qu'il soit correctement compris, puis correctement exécuté.

Nous distinguerons donc dans tout compilateur :

- un *frontal,* chargé d'extraire (et coder) l'algorithme du programme,
- un *exécutant*[4], chargé de mettre en œuvre l'algorithme (i.e. de transformer les données en résultat).

Leurs principales caractéristiques permettront de préciser les failles possibles et ce qu'il y a lieu de tester, des aspects les plus superficiels aux plus profonds.

Test du frontal

Le frontal a une double fonction d'analyse du programme reçu, et de codage de son algorithme. L'évolution récente tend à rendre ce frontal interactif, afin de maximiser les chances de mener à bien l'analyse et le codage de programmes semblant problématiques.

---

[3] d'après C.H.A. Koster, ceci ne constitue que les premiers 30% d'un compilateur

[4] au sens de F.H. Raymond

On attend du frontal :

- qu'il ne refuse aucun programme correct,
- qu'il code exactement l'algorithme des programmes acceptés,
- qu'il refuse les programmes erronés, avec un diagnostic précis et correct de la ou des erreurs.

Test de l'exécutant

Interprète ou Compilateur ?

Le problème est maintenant de vérifier qu'un programme accepté voit son algorithme correctement exécuté.

L'exécutant est dit *interprète* s'il exécute immédiatement, pas à pas, le code produit. Il s'agit en général d'un *code de haut niveau*, en ce sens qu'il est inversible : on peut lui associer, par exemple à l'aide d'un paragrapheur, un programme dans le langage-source L, équivalent au programme initial. En général, comme on n'a pas de machine sachant exécuter un tel code, l'interprète est un programme simulant une machine comprenant L, parfois appelée L-machine, ou machine virtuelle pour L. Le test d'un tel exécutant revient à tester à la fois que la production de code est correcte, et que l'interprète est une L-machine correcte. Cette solution s'impose lorsqu'on entend privilégier la souplesse : langages destinés à la programmation exploratoire, au maquettage, mais aussi langages ambitieux en début de vie, la rédaction d'un interprète permettant d'acquérir par l'expérience une compréhension plus globale de la sémantique.

On parle plutôt de *compilateur* quand le frontal transcode l'algorithme dans un langage X plus proche de la machine disponible que de L. Issu d'un générateur, le code produit est dit *code de bas niveau*, en ce sens qu'on ne peut plus lui associer un programme dans le langage-source L clairement équivalent au programme initial. Le code obtenu peut être exécuté en différé, voire traduit de nouveau dans un langage plus physique. Cette solution, qui substitue aux programmes en L un code-image s'exécutant efficacement sur une machine donnée, s'impose lorsqu'on entend privilégier les performances — logiciels des couches basses ou applications lourdes.

Les codes ne sont parfois que des codes-pivots :

- certains ateliers de génie logiciel utilisent un code interne de haut niveau commun à plusieurs langages de sémantique comparable, chaque langage ayant son propre frontal ;
- certains systèmes multi-cibles utilisent un code interne de bas niveau commun à plusieurs processeurs assez comparables, chaque processeur disposant d'un générateur particulier ; ainsi, P-code, M-code, S-code ou Byte-code peuvent permettre à des compilateurs Pascal, Modula 2, Simula 72 ou Java d'utiliser un même frontal, en ne différant que par les générateurs relatifs à tel ou tel processeur.

L'exécutif

Qu'il y ait interprétation ou compilation, la plupart des programmes exploitent des procédures prédéfinies dans le langage L, servant aux entrées/sorties ou à des traitements standards. C'est dire que tout exécutant s'appuie sur un outillage, ou *exécutif*, regroupant une image efficace des procédures prédéfinies. Cet exécutif est souvent important :

- pour les langages généraux, afin de montrer leur efficacité, en offrant des procédures utiles de tous ordres,

- pour les langages spécialisés, l'offre d'actions synthétiques puissantes impliquant l'existence de modules spécialisés prédéfinis ;
- en 1961, avec son Lisp, McCarthy a inauguré la filière des langages auto-extensibles : on ne réalise au plus près de la machine qu'un noyau nécessaire-et-suffisant du langage basé sur des procédures dites *primitives*, puis on développe dans le langage déjà disponible des mécanismes secondaires plus commodes… On retrouve ce mécanisme en Simula 67. En C, les E/S sont bâties sur les 3 primitives *read, write* et *ioctl*, définies au ras de la machine, et un module permettant un nombre variable de paramètres dans une procédure ; les ordres usuels comme *printf* sont alors définis en C… On a alors affaire à un exécutif mixte : quelques primitives définies au ras de la machine, les autres procédures prédéfinies l'ayant été en L.

En conclusion, le test de l'exécutant regroupe trois vérifications :

- pour un programme accepté, l'image produite est correcte ;
- l'exécution d'un code exécute correctement ses commandes ;
- l'image des procédures prédéfinies invoquée est correcte.

Qualité de l'analyse

L'analyse comprend en général trois étapes :

- l'*analyse lexicale*, qui s'occupe d'isoler les mots du langage ;
- l'*analyse syntaxique*, qui repère les structures du langage ;
- l'*analyse sémantique*, qui s'intéresse au sens des constructions repérées, en explicitant si nécessaire les informations « par défaut ».

Analyse lexicale[5]

Elle s'occupe d'isoler dans le texte du programme les chaînes de caractères qui correspondent à une entité élémentaire du langage (constante, identificateur, ponctuation, opérateur…) – de forme généralement définie par une expression régulière. Après une *segmentation* isolant ces entités lexicales à partir d'un flux entrant de caractères, l'analyseur lexical identifie les lexèmes de ce flux et les range dans des catégories d'entités lexicales. S'il détecte une entité lexicale invalide, il rapporte une erreur.

Généralement, les combinaisons d'entités lexicales sont laissées aux soins de l'analyse syntaxique — un analyseur lexical reconnaîtra les parenthèses comme étant des entités lexicales, sans s'assurer qu'une parenthèse ouvrante « (» est forcément associée à une parenthèse fermante «) ».

De fait, l'analyse lexicale peut connaître des difficultés sources d'erreur :

- problème de segmentation :
    - les Fortrans les plus anciens devaient tolérer des blancs partout, ce qui nuisait à la segmentation, notamment à l'identification des mots-clés ;
    - un compilateur Simula, assimilait toute suite de caractères spéciaux à un symbole unique, si bien que « U :=–W ; » lui était incompréhensible, «:=–» lui étant inconnu ;

[5] les mots des langages traités étant présumés invariables, il n'y a pas d'analyse morphologique ou morpho-syntaxique.

- problème de reconnaissance : Pascal MS admet des constantes entières du style « base#chiffres», où base est un entier à 1 ou 2 chiffres, valant de 2 à 36, et où chiffres est une suite de chiffres pris dans « 0123456789ABC…XYZ » *selon la base exprimée* : ainsi, 20#c* désigne une constante entière en base vingt formée de chiffres pris dans « 0123…HIJ » ;
- problème de classification : PL/1 étant basé sur un important vocabulaire, ne traitait pas en mots réservés ses quelques 200 mots-clés ; ainsi, BEGIN pouvait être :
    - une instruction d'ouverture de bloc « BEGIN ; » ;
    - une étiquette « BEGIN : » ;
    - une interruption localement armée par l'utilisateur « (BEGIN): »
    - une variable « DECLARE BEGIN CHAR(5) ; » ou « BEGIN, END = X+1; ».

A un niveau supérieur, aux confins de l'ergonomie, surgit le problème des tolérances lexicales :

- conflit entre les règles de ponctuation du langage, et ce qui est normal dans une langue proche : l'Université de Rennes a établi que pour les langages à point-virgule, plus de 30% des erreurs venaient d'une absence de « ; » en fin de ligne ;
- pour les programmeurs occasionnels ou débutants, demande pour des mots-clés dans leur langue ; A. van Wijngaarden a proposé de disjoindre le symbole de sa/ses représentation(s), ce qui permet d'associer à une représentation principale diverses représentations secondaires, au risque de nouvelles ambiguités et de nouvelles demandes : si je peux écrire « imprimer x, y, z sur fic4 ; », pourquoi pas « imprime | imprimez | imprimons x, y, z sur fic4 ;» ?

Pour un langage donné, un compilateur quelque peu tolérant sera apprécié, si ce libéralisme n'introduit pas de nouvelles erreurs…

Analyse syntaxique

Depuis Algol 60, elle est en général basée sur l'exploitation d'une grammaire formelle de type Backus-Naur ou d'un type dérivé.

Deux stratégies essentielles :

- dans la *stratégie ascendante*, souvent prônée, on utilise l'analyse lexicale comme guide ; le mot clé « begin » suggère un début de bloc…
- dans la *stratégie descendante*, on sait qu'au plus haut niveau on cherche un programme ou un module ; si c'est un programme, alors l'analyseur lexical doit nous fournir le mot-clé « programme » ; l'analyse lexicale est alors dite indirecte ; elle connaît moins d'ambiguïtés, car elle répond à une demande précisée par un contexte gauche connu (réel ou supposé).

Spécification de la syntaxe

C'est une question cruciale. D'une part, la définition officielle du langage L (rapport de définition, standard ou norme) fournit souvent une grammaire officielle, éventuellement modifiée dans le manuel de référence du produit, explicitant le langage traité L' comme variante de L. Mais cette grammaire officielle est en général simplifiée, mieux adaptée aux besoins de l'utilisateur (programmeur ou formateur) qu'aux besoins de l'analyse.

Rappel

Deux grammaires décrivant un même langage sont dites *fortement équivalentes* si elles en fournissent la même analyse, *faiblement équivalentes* sinon.

Et un langage est dit de classe n (au sens de Chomsky) s'il possède au moins une grammaire de classe n, et si n est maximal pour cette propriété.

Du point de vue de la seule *acceptation* d'un langage, une grammaire peut être très superficielle, comme dans :

expression :	terme, fin d'expression.
fin d'expression :	opérateur binaire, expression ; vide.
terme :	constante ; variable ; «(», expression, «)».

Du point de vue de l'*analyse*, la question est de trouver parmi toutes les grammaires équivalentes de L, une grammaire fournissant la meilleure analyse du programme, au sens de l'algorithme sous-jacent.

Cas de l'expression

Quand le parenthésage n'est pas total et que les opérateurs ne sont pas tous associatifs, on utilise un système de priorités, comme dans :

expression :	terme, fin d'expression.
fin d'expression :	opérateur additif, expression ; vide.
terme :	facteur, fin de terme.
fin de terme :	opérateur multiplicatif, terme ; vide.
facteur :	constante ; variable ; «(» , expression, «)».

Cependant, cette grammaire analyse A–B+C comme A–(B+C), alors que nous nous attendons à (A–B)+C … Une telle erreur n'est hélas détectable qu'au niveau du codage, généralement sur enquête après une exécution constatée fausse (avec C≠0 …) .

De ce point de vue, il faudrait donc avoir :

expression :	expression, opérateur additif, terme ; terme.
terme :	terme, opérateur multiplicatif, facteur ; facteur.
facteur :	constante ; variable ; «(» , expression, «)».

ce qui est correct en termes de génération, mais dont l'analyse descendante est difficile du fait des récursivités gauches.

Cas de l'affectation multiple

Certains langages comme occam autorisent des affectations multiples du style :

variable1, variable2, variable 3 := expr1, expr2, expr3.

Les compilistes ont tendance à supposer une grammaire du style :

affectation : variable, (« , », affectation, « , », expression ;
« := », expression).

Une telle règle de classe 2 est facile à traiter, mais correspond à une structure parenthétique, signifiant de fait :

variable1, variable2, variable 3 := expr3, expr2, expr1.

Cette facilité d'analyse entraînerait à l'exécution un chassé-croisé de valeurs dès que expr3≠ expr1, sauf à la faire suivre d'une remise en ordre.

Ou à considérer la syntaxe de ces affectations comme relevant de la classe 1, dont les structures de recopie fourniraient une analyse correcte de ces affectations.

Ambiguïtés

Un algorithme ne pouvant être ambigu, un programme ne doit pas l'être, ni son analyse. On devra donc bannir les règles comme :

```
liste :  liste, «,», liste ; truc.
```

qui, pour une liste de 1, 2, 3, 4, 5, 6… trucs fournit 1, 1, 2, 5, 14, 42… analyses[6].

Mais on devra utiliser des règles non ambiguës comme :

```
liste :  truc, suite de liste.
suite de liste : «,», liste ; vide.
```

qui fournissent une seule analyse, et accélèrent la détection d'erreurs.

Grammaire exploitée

Les techniques de méta-compilation permettent une utilisation directe ou presque de la grammaire spécifiée.

Sinon, la grammaire exploitée est une réécriture « machine » de la grammaire spécifiée, par exemple en « formes normales de Floyd » du style

$$X \rightarrow TY \mid Z$$

où X, Y, Z désignent des notions quelconques, et T une notion terminale. Transcription qui peut être une source d'erreur.

Résultat attendu

Un *accepteur* se contenterait d'un diagnostic local (erreur constatée) ou global (compilande accepté).

Un *analyseur syntaxique* assure la production d'un arbre, reflétant une analyse succincte du signifiant (programme ou compilande), dont les principaux nœuds coupleront déclarations locales et algorithmes locaux.

L'*analyse sémantique* tente alors de transformer cette représentation encore superficielle en une représentation du signifié (l'algorithme), en se basant sur la théorie du langage. Pour cela, dans cet arbre les attributs implicites sont explicités (l'arbre est « décoré »), et on effectue les vérifications possibles.

Analyse sémantique

Elle correspond essentiellement à la sémantique statique (au sens de l'Ecole de Vienne), c'est-à-dire à la vérification de la cohérence de l'unité analysée au sens de la théorie du langage traité.

Elle explicite les informations par défaut, comme les valeurs ou attributs implicites, avant de procéder aux vérifications.

Elle ne peut aller au-delà de la théorie du langage, qui peut être :

[6] suite des nombres de Catalan ; ce genre de règles se trouvait dans la grammaire de Fortran 77

- lacunaire : de nombreuses situations n'ont aucun sens ; si elles ne peuvent être diagnostiquées, elles autorisent nombre de programmes corrects (en apparence) mais invalides (à l'exécution) ;
- permissive : le programmeur étant présumé avoir toujours raison, de nombreux cas douteux sont régularisés en multipliant les conversions et alignements divers, quitte à obtenir des exécutions inexplicables ;
- rigoureuse : facilitant les contrôles, elle risque d'encombrer les programmes d'interminables préliminaires suivis d'écritures minutieuses.

L'analyse sémantique sera peu ambitieuse pour les langages interprétés, plus conséquente pour les langages compilés, particulièrement fouillée pour les langages destinés aux grosses applications.

Ce dernier point est très clair si on compare la filière sécuritaire Modula 2/ LIS[7]/ Adas à l'évolution d'un langage comme C, langage structuré de bas niveau, et dont l'évolution à partir de 1969 marque un durcissement progressif destiné à faire face à son succès – la compatibilité ascendante laissant des failles nuisant encore aux développements collectifs.

Preuve de programmes

C'est un cas extrême, applicable à des langages axiomatisés comme *occam*, où une analyse sémantique statique tente d'établir logiquement lors de la compilation :

- que le programme est correct si le programme se termine proprement (correction partielle) ;
- que le programme se termine proprement (correction totale si la correction partielle est déjà acquise).

On se contente en général des vérifications partielles ci-après.

Typage

Du fait de la variété des données, cette question surgit dans les opérations, les affectations, les appels de fonction, de procédures, de modules…

Dans les langages à typage faible (APL, Lisp, Prolog…) les données ont un type, et les variables sont du type de leur valeur, qui peut changer à toute nouvelle affectation.

Pour des raisons de sécurité, les langages à typage fort (Algols, Simulas, Modulas, Adas…) imposent que chaque variable (ou argument ou paramètre) ait un type précis, et que les opérations, affectations, appels de fonction, procédures ou modules se fassent dans le strict respect de la concordance des types. Pour cela, le type de chaque variable est déclaré, sauf dans les *langages à inférence de type* (famille ML) où le type d'une variable peut être déduit de ses usages, pour autant qu'ils soient homogènes.

> On peut citer le cas extrême du langage Ampère, reprenant une idée de Michel Sintzoff : utiliser pour les calculs techniques un langage où :
>
> - les variables représentant des grandeurs physiques sont typées selon la nature de ces grandeurs, au sens de l'*analyse dimensionnelle* ;
> - les constantes sont typées et cadrées par les noms d'unités ;
> - à chaque type physique T est associé une signature dimensionnelle [T].
>
> Dans ce système, (+, –, max, min, :=) et les comparateurs sont définis entre opérandes de même signature, tandis que (*, /) admettent des opérandes de tout type, avec un résultat de

[7] Langage d'Implémentation de Systèmes, J. Ichbiah et coll, CII, précurseur d'Ada

signature produit ou quotient des signatures des opérandes. Les contrôles sémantiques sont réalisés par évaluation symbolique, les signatures formant un groupe multiplicatif. Alors, comme [P/Q] = [P]/[Q], la permutation par erreur de P et Q est détectée dès que type[P] ≠ [Q].

Le typage fort étant sûr mais pénible, il est souvent modulé au niveau des emplois par des procédés de conversion d'un type dans un autre, explicites (lourds mais sûrs) ou implicites (commodes mais parfois inattendus).

Les types et conversions implicites sont des commodités génératrices d'incertitudes sur l'appréhension du programme par le compilateur, et qui peuvent rendre énigmatiques l'acceptation voire l'exécution de certains programmes[8]. L'édition d'une table de références croisées est alors un moyen de vérifier la compatibilité entre analyse du compilateur et intention du programmeur.

Aux origines, le typage **Fortran** était implicite : toute variable en I, J, K, L, M, N était entière, les autres étant présumées réelles (virgule flottante). Puis s'est rajouté un système de déclaration de types « si nécessaire ». Si bien qu'une faute de frappe pouvait dédoubler une variable, par exemple en NI et N1, l'une étant déclarée, l'autre pas, l'une étant initialisée, l'autre pas. D'où des programmes « corrects » à exécution aléatoire…

PL1 reprit cette tradition, avec un grand luxe de types définis à l'aide de nombreux attributs, explicites ou déductibles. Ainsi, une variable I non déclarée se voyait dotée d'une quinzaine d'attributs… Présumant que le programmeur avait toujours raison, les conversions pour alignements de type étaient menées selon de nombreuses règles dont l'application réitérée défiait rapidement le bon sens. D'où une proportion élevée de programmes réputés corrects, mais idiots ou d'exécution incompréhensible.

Forme des algorithmes

Si le langage le permet, on vérifie la bonne forme des programmes analysés :

- *boucles propres* (munies d'un test d'arrêt comportant au moins une variable modifiée dans le corps de boucle) ;
- statut des *alternatives*, de même statut que leur dernière branche, les branches précédentes étant toutes conditionnelles, i.e. comprenant au moins un test comportant au moins une variable ;
- *fonctions propres* : certains langages[9] exigent que, contrairement aux procédures, les fonctions, comme en mathématique, soient indépendantes du contexte ; cette propriété vérifiable favorise l'optimisation des appels ; le corps de la fonction ne doit alors employer ni modifier aucun fichier ni variable non-locale ;
- *synchronisation* en cas de programmation concurrente.

La question de la validité des tests des boucles et alternatives peut être liée d'une part à l'évaluation des (sous-) expressions constantes, d'autre part à leur éventuelle utilisation dans un mécanisme de compilation conditionnelle.

Modules, procédures, arguments et paramètres

Modules et procédures sont des unités paramétrées mettant à la disposition du programmeur les concepts de composants réutilisables, « boites noires » à l'extérieur, « boîtes blanches » par leur corps.

---

[8] en Fortran comme en C, diverses compatibilités de type ne sont valables que sous réserve de représentations physiques trop spécifiques pour être reprises dans les normes

[9] hors la mouvance C

Les *paramètre*s assurent la communication et la définition du traitement ; d'un emploi sur l'autre, diverses données ou *arguments*, jouant un même rôle (à l'extérieur) sont connu(e)s sous un même nom ou paramètre (à l'intérieur) pour définir le traitement.

Les principaux modes de passage des paramètres sont autant de façon de réaliser les affectations « argument → paramètre » à l'invocation ou les affectations « paramètre → argument » en retour. Ils ont d'abord été définis par référence à leur réalisation physique. On distingua ainsi le passage :

- *par valeur* : le paramètre est une copie de l'argument donné ;
- *par référence* : le paramètre représente l'adresse de la donnée ;
- *par nom* : le paramètre représente le libellé de la donnée ;
- *en résultat* : le paramètre désigne une donnée locale, retransmise à l'argument en fin d'exécution de l'unité paramétrée ;
- *par mot-clé*, réservé aux unités génériques lourdement paramétrées : seuls les paramètres nommés sont associés aux données externes, les paramètres non explicités utilisant une valeur par défaut.

La plupart des langages n'utilisent officiellement qu'un ou deux modes de passage de paramètres. Mais comme ceux-ci ont des avantages et inconvénients divers, relatifs notamment aux implantations de données et à la sécurité, les compilateurs prennent souvent des libertés avec la théorie annoncée, astuces pouvant entraîner l'exécution erronée de programmes corrects.

Passage par valeur et par référence se sont inspirés du vecteur de transfert utilisé en assembleur ; mais le passage par valeur s'est révélé coûteux pour les données structurées, tandis que le passage par référence se révélait peu fiable s'il était fait « en aveugle ».

L'unique passage par référence de Fortran pouvait se faire « en aveugle », avec par exemple un entier comme argument et un réel comme paramètre.

Si on constatait un fonctionnement normal avec

```
CALL SP(1.0)
```

```
I=1
CALL SP(I+0.0)
```

```
I=1
CALL SP(FLOAT(I))
```

et anormal avec

```
I=1
CALL SP(I)
```

c'est que l'adresse de l'argument était passée sans conversion de cet argument… et que la forme « I+0.0 » est une correction effaçable par une optimisation.

De même,

```
CALL INCR(3)
```

```
SUBROUTINE INCR(N)
N=N+1
RETURN
END
```

est un sûr moyen de corrompre 3 dans la table des constantes, les 3 utilisés après l'appel étant devenus des 4…

De nombreux programmes **Algol 60** validés avaient des exécutions problématiques, car le *passage par nom* demandant une macro-substitution, était remplacé dans de nombreux compilateurs par un passage par référence – qui, en figeant le contexte à l'appel, amenait des divergences dans le cas d'arguments interdépendants (cf. Wirth 80).

Les défauts de ces premiers modes ont entrainé l'apparition d'autres modes comme « const ref », de fait un passage par référence interdisant l'affectation de résultat afin d'avoir la sécurité d'un passage par valeur au coût d'un passage par référence.

La notion de *profil* de fonction, d'opérateur et/ou de procédure permet en principe de régler proprement la double question de la concordance de type et du mode d'échange entre arguments et paramètres.

> Modula 2 a montré comment articuler les modules en une interface publique spécifiant les profils des procédures exportées, et un corps privé contenant la définition des procédures exportées et des procédures locales qu'elles emploient.

Passant du *comment* au *pourquoi* on a tendance, en Ada par exemple, à spécifier les modes de passage des paramètres en fonction du problème général des flux d'informations.

> Typiquement, Ada admet quatre modes de passages pour ses paramètres :
>
> - le mode *in*, mode récepteur de valeur, et mode par défaut,
> - le mode *out* pour passer un résultat à un argument externe,
> - le mode *in out* pour utiliser une variable externe en entrée et la modifier en sortie,
> - et le mode *access* qui demande un pointeur qui ne sera pas modifié.
>
> Les fonctions ne peuvent avoir que des paramètres *in* ou *access* ; il n'y a aucune contrainte pour les paramètres d'une procédure.

Flux d'informations

Pour assurer la cohérence des applications, on examine le cycle de vie des valeurs, chacune étant émise par une *source* en direction d'un ou plusieurs *puits*.

Les sources sont les constantes, les variables ou paramètres initialisés, les valeurs d'opérateurs ou de fonction, les expressions, les arguments en sortie.

Les puits sont les arguments en entrée, les variables-cibles, les paramètres en sortie.

Le système est étendu aux entrées/sorties, fonctions, opérateurs et procédures prédéfini(e)s.

Le contrôle des flux d'informations détecte l'emploi de variables non initialisées, cause fréquente d'exécution aléatoire.

Il peut aussi détecter les variables inemployées. Dans les grands développements (notamment en prototypage évolutif), elles sont la trace d'idées abandonnées, possibles source d'erreur, ou encore d'une optimisation inachevée.

Procédés

L'analyse sémantique constitue souvent une phase autonome, examinant l'arbre produit par l'analyse syntaxique à la lumière de la théorie du langage. Les éléments implicites sont explicités, et les vérifications de cohérence possibles sont effectuées.

Cependant, l'analyse sémantique peut être menée au fil de l'analyse syntaxique, en utilisant des grammaires à 2 niveaux (cf. Cleaveland & Uzgalis ; Koster ; Beney).

La règle de grammaire traditionnelle :

```
affectation :        variable, « ← », expression.
```

généralement flanquée de la « restriction sémantique « où *variable* et *expression* sont de même type », peut être réécrite :

```
affectation :        variable booléenne, « ← », expression booléenne ;
                     variable numérique, « ← », expression numérique ;
                     variable chaîne, « ← », expression chaîne.
```

ou plus généralement :

```
affectation :        variable (type), « ← », expression (type).
----
type :: booléen ; numérique ; chaîne ; rangée de type ; ref type.
```

Dans ce dernier cas, « affectation » est une *hyper-règle* dont l'affixe *type* peut prendre une infinité dénombrable de valeurs (chacune désignant un type concret). Le principe de remplacement uniforme impose alors la concordance entre type de la variable et type de l'expression.

Du point de vue de la compilation, on utilisera plutôt :

```
affectation :        variable (type), « ← », expression attendue (type).
expression attendue (type) :     expression(type) ;
                                 erreur(« expression attendue de type », type).
---
type :: booléen ; numérique ; chaîne ; rangée de type ; ref type.
```

Ce dernier exemple montre que le contrôle sémantique peut compléter à tout moment l'analyse syntaxique lors de la construction de l'arbre résultant, et mener à des messages d'erreur en situation.

Qualité du codage

Le plus souvent, la qualité du code n'est perceptible qu'à l'exécution, qu'il s'agisse de l'exécution erronée ou aléatoire d'un programme accepté, ou plus finement de l'efficience ou de la lourdeur d'une exécution correcte.

S'il s'agit d'un interprète, les phases antérieures sont sommaires, et le code reste suffisamment près du programme source pour être réversible. C'est cependant au moins une mise sous forme normale, s'affranchissant des tolérances lexicales ; on peut pousser plus loin pour une exécution efficace, par exemple en adoptant une écriture polonaise.

Pour une compilation, les phases antérieures sont plus riches, et la qualité du code n'est souvent perceptible qu'à l'exécution – pour se révéler souvent victime d'erreurs préalables non détectées.

Validité du code

Parmi les erreurs constatées :

- mauvaise distinction entre évaluation de gauche à droite et évaluation de droite à gauche :

$$A–B–C = (A–B)–C, \text{ mais}$$

$$A**B**C = A**(B**C) \neq (A**B)**C = A**(B*C) ;$$

- pour les opérateurs non-commutatifs, mauvaise génération liée aux machines ayant des instructions à 2 adresses, du style « op R1, R2 » pour « R2 := R1 op R2 ».

- dans le cas de surcharge d'opérateurs, mauvaise sélection[10].

Sûreté du code

Le code engendré, correct vis-à-vis du programme source, donnera-t-il une exécution valide ? Ici se glisse deux questions :

- *variables non initialisées* : les codes qui les emploient mènent à des exécutions aléatoires ; elles devraient donc être interdites ; si la définition du langage ne les interdit pas, et si l'analyse sémantique ne les détecte pas, il est alors souhaitable de leur affecter à la création une valeur illicite (genre NaN) si cela peut déclencher une interruption à l'exécution ;
- *expressions hors domaines* : pour les indices de tableau comme pour les arguments de fonctions ou procédures prédéfinies, ce que l'analyse sémantique n'a pu détecter devrait être confié à des mécanismes de surveillance[11] type « assert ».

  En Fortran, la déclaration EQUIVALENCE permet la superposition de tables, ce qui désarme la surveillance des indices… et permettait sur IBM 1130 à certains programmes Fortran aux indices erratiques de détruire le système – non protégé.
- *synchronisation* des processus concurrents.

Efficience du code

Le code est souvent médiocre si le langage est neuf mais surtout si sa définition est longue, ce qui reflète souvent une théorie trop faible, et absorbe de grands efforts dès les premières phases de la réalisation.

> Ce fut le cas de PL1, assez peu efficace sur la gamme 360/370 pour laquelle il avait été conçu – sa normalisation fut d'ailleurs problématique.
>
> Par opposition à la définition de Pascal (49 pages dans la version de 1973) ou aux 220 pages de *Programmer en Modula 2*, Pascal MS 3.2 (1986) avait une très longue définition riche en pseudo-extensions, reflétant souvent les particularités des processeurs Intel 80x86 ; à l'arrivée le traitement de l'adresse des variables indicées comme celui des instructions *case* était onéreux[12], et mettait en échec les prédictions de performances des applications.
>
> Une même équipe ayant utilisé ce Pascal, et un Pascal plus standard sur un microordinateur à base de Motorola 6809, constata qu'en moyenne une ligne Pascal produisait 20 octets de code sur IBM PC, et seulement 8 octets sur l'autre machine.

Si un code est de qualité, il n'y a pas besoin d'optimisation. A contrario, les optimisations rajoutées sont source de problèmes – par opposition à une génération au mieux.

> Un compilateur Algol 60 avait prévu deux générations pour la boucle *pour*, selon le type de l'indice ; la génération la plus générale était correcte quoique en principe peu utile, mais la « génération optimale » pour les indices entiers s'arrêtait mal.

Certains compilateurs « avec option d'optimisation » laissent le choix entre un code médiocre et un code faux, du fait d'optimisations imprudentes.

---

[10] les différents codes ont en principe un nom générique suffixé par une « notation hongroise » du profil.

[11] la *programmation défensive* présume faux tout argument transmis à une fonction ou procédure…

[12] cascade d'instructions conditionnelles au lieu d'un branchement calculé.

Génération au mieux

Aspect haut niveau

De nombreux langages autorisent les *constantes nommées*, utilisables pour définir une compilation conditionnelle (démo, débug…) ou une configuration. Alors, l'*évaluation des expressions constantes* à la compilation est un bon moyen d'alléger le code, notamment lorsqu'elles servent à définir des constantes secondaires.

Au-delà, certains compilateurs procèdent à la recherche de sous-expressions identiques ou à des optimisations algébriques permettant la suppression de variables intermédiaires ou de certaines étiquettes devenues sans objet[13].

Aspect bas niveau

Un signe de faiblesse dans la génération de code est l'apparition anormalement fréquente d'un message « expression trop complexe ».

> Ce message apparaissait souvent si on utilisait le compilateur Pascal fourni par l'INRIA pour l'Iris 80, compilateur développé en 1974 par bootstrapping à partir du compilateur Pascal développé pour les CDC 6600 et 6400.
>
> C. Mauger (CCILS) remarqua que ces ancêtres des machines RISC utilisaient des instructions longues de chargement/ rangement, procédant aux échanges entre la mémoire centrale et 8 registres, et des instructions courtes à 3 registres pour les calculs, tandis que l'Iris 80 disposait d'instructions combinant le contenu d'une adresse et un registre. De ce fait,
>
> A+B+C+D
>
> utilisait jusqu'à 4 registres, 4 chargements et 4 additions sur CDC, et pouvait descendre à 1 registre, 1 chargement et 3 additions sur Iris 80. Une allocation de registres plus adaptée rendit le message « trop complexe » rarissime, et d'autres améliorations permirent à C. Mauger de réduire en général le code objet par un facteur 2,8.
>
> Un problème semblable peut se poser pour tous les compilateurs portables, quand le jeu d'instructions de la nouvelle machine diffère du précédent.

Qualité de l'exécution

Elle comporte deux aspects :

- la *qualité du code de l'application*, code spécifique souvent médiocre si le langage est neuf ou si une trop longue définition demande trop d'efforts dès les premières phases de la réalisation ;
- la *qualité de l'exécutif* (code générique faisant partie de l'exécutant ou machine virtuelle que l'utilisateur distingue mal du compilateur), qualité qui dépend de sa programmation, de son code, comme de la machine sous-jacente.

Les fonctions standards peuvent être sujettes à caution, voire la virgule flottante elle-même (comme dans les premiers Intel 486).

Test de fonctions standards

Soit une fonction f à tester, et I(f) un invariant de f, supposé irréductible par le compilateur. Pour tester f, on utilise un programme traquant les valeurs anormales de I(f) en balayant le domaine de f.

---

[13] un optimiseur de haut niveau a ainsi ramené une application Fortran de 45 000 lignes à 25 000…

L'analyse de la première anomalie est importante[14], mais avec l'expérience il arrive que la suite des anomalies puisse s'interpréter comme un spectre typique de l'erreur.

Tester une virgule flottante

Soit à vérifier la *qualité effective d'une virgule flottante*. Pour cela, on examinera par exemple la valeur *calculée* de $x^2-(x-1)\times(x+1) \equiv 1$, avec un programme du style :

```
réel x ;
x ← 1e-10 ;
tant que x< 1e+10 faire ;
        si (x^2 -(x-1)*(x+1) ) ≠ 1 alors {      écrire (x, «→ », (x^2 -(x-1)*(x+1) )) ;
                                                alaligne };
        x ← (-1.2)*x ;
fin_faire
```

A raison de 12 ou 13 valeurs par décade sur 20 décades, ce programme listera les valeurs de x problématiques. *Normalement*, ces valeurs devraient apparaître lorsque la virgule flottante employée peine à distinguer x^2 de x^2-1, c'est-à-dire lorsque x^2 approche l'inverse de la précision. Autrement dit, la *précision effective* est de l'ordre de $1/x^2$, par exemple serait de 10^{-7} pour x de l'ordre de 3200.

On montera peut-être plus haut si l'invariant, exploité directement, est calculé en précision étendue. On « accrochera » plus tôt s'il y a un défaut dans la virgule flottante (par exemple, dans la multiplication) ou dans l'arrondi d'affectation. La distribution des valeurs anormales fournira en général une piste.

Un programme similaire peut être appliqué à l'évaluation d'une fonction $f(x) = x^2-(x-1)\times(x+1)$, vérifiant de plus une passation de paramètre et la valeur retournée.

La même idée permet de tester des procédures d'algèbre linéaire : jusqu'où a-t'on :

$$\forall x : A = \begin{bmatrix} x & x-1 \\ x+1 & x \end{bmatrix} \rightarrow A \times A^{-1} \equiv \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

sachant que la valeur du déterminant est en principe précisément $\Delta(A(x)) \equiv x^2-(x^2-1)$?

Remarque

Du point de vue de la précision de la virgule flottante, la multiplicité des formats n'est pas bon signe :

- le format 32 bits est intrinsèquement médiocre : il multiplie les besoins en précision étendue, qui s'éteignent si la virgule flottante possède au moins 36 bits de mantisse ; c'est donc un format à oublier ;
- le format 64 bits couvre la plupart des besoins ; mais la présence d'un format 128 bits peut encourager un certain laxisme : des algorithmes convergeant en 64 bits sur Iris 80 ne convergeaient pas sur IBM 360, un format 128 bits y accompagnant un format 64 bits peu précis[15].

[14] on pourra notamment consulter Abramowitz & Stegun

[15] un couple 48 bits / 96 bits est probablement meilleur.

Test d'opérations

Les tests doivent porter sur les valeurs comme sur les propriétés. Sauf optimisation, on vérifiera par exemple que :

$$x–(x–a) – a \equiv 0 \text{ pour tout x ;}$$

$$(x +y) \oplus (x \oplus y \oplus x \cdot y) \equiv 0 \text{ pour x, y} \in \mathbb{B} ;$$

$$x–(\sqrt{x})^2 \equiv 0 \text{ pour x} \geq 0 \text{ (et que } \sqrt{\text{x}} \text{ intercepte les x<0) ;}$$

$$max(x, y) + min(x, y) – (x+y) \equiv 0.$$

Tester un sinus

De $sin\ 2x = 2\ sin\ x \cdot cos\ x$ on peut déduire $sin^2\ 2x – 4\ sin^2\ x \cdot cos^2\ x \equiv 0$, soit :

$$sin^2\ 2x – 4\ sin^2\ x \cdot (1– sin^2\ x) \equiv 0.$$

Un programme similaire au précédent vérifiant l'invariance de cette formule devrait trouver un domaine correct inférieur à celui de la virgule flottante, car *sin(x)* est probablement calculé à partir de *sin(y)* avec par exemple $y = x–k\cdot\pi/2$ et $y \in [-\pi/4\ \pi/4]$; soit x = 500 ; $1000/\pi \approx 318$; $500 – 318\cdot(\pi/2) \approx$ 0,487, et l'argument y est 1000 fois moins précis que x ; un avertissement « argument trop grand » devrait donc apparaître aux grands angles quand l'incertitude sur y devient excessive.

Remarques

Les particularités de la virgule flottante se répercutent sur les fonctions standards :

- Soit *sind(x)* la fonction calculant le sinus d'un angle exprimé en degrés. Elle sera plus précise si elle exploite un argument *réduit en degrés* y = x – 90k, 90 ayant une représentation flottante exacte.
- De tous les logarithmes, le plus précisément calculable est celui ayant pour base celle de la virgule flottante, 2 en général, car

  $$x = m \times 2^{exp},\ 1 \leq m < 2 \rightarrow \log_2(x) = exp + \log_2(m)$$

  où $\log_2(m)$ se calcule sans problème dans [1 2[.

  De même, si $x = n,f_2$ alors $2^x = 2^n \cdot 2^f$ avec $0 \leq f < 1$: ainsi, 2^x, représenté par un nombre d'exposant n et de mantisse 2^f, devient la plus simple et la plus exacte des exponentiations.
- $\sqrt{\text{x}}$ est en général calculable directement, et plus précisément que exp(ln(x)/2) ou même $2^{\log_2(x)/2}$. On pourra ainsi tester un logarithme par

  $$log(x)/2–log(\sqrt{x}) \equiv 0.$$

Aspects temps réel

En termes de sûreté, ces tests sont les plus nécessaires, mais ce sont les plus difficiles – et, in situ, les plus dangereux et les plus coûteux[16]. En effet, il faudrait vérifier qu'ils font face correctement à toute situation, quelques soient les données mais aussi quelle que soit la séquence des évènements.

Une première difficulté peut surgir de flous dans la définition des langages.

> En PL/1, une interruption peut voir son traitement modifié dynamiquement ; soit A, B, C trois définitions, activées successivement d'une certaine interruption. L'instruction REVERSE

[16] réaliser des tests in-situ sur une certaine chaîne de production engendrait une perte de 1 million de francs par heure d'interruption ; quand au coût d'une erreur…

permet en principe de revenir à la définition B. Mais la définition ne dit pas si un nouveau REVERSE ramène à la définition A ou à la définition C.

C'est pourquoi on tente parfois d'exploiter à fond tout enregistrement relatif au proche passé d'un incident.

On accorde également la plus grande attention à la *prévention* d'incident.

L'axiomatisation des langages (ex : Roscoe & Hoare) permet aux compilateurs de vérifier avant exécution diverses conditions de bonne fin.

Pour ne pas avoir à démêler les interruptions déclenchées par des alarmes en avalanche, nombre d'automates programmables fonctionnent par scrutation.

A un niveau plus global, le système CHORUS développé à l'INRIA pour la SM 90[17], basé sur un formalisme acteur, a été fiabilisé en décidant que *toute réception de donnée ou de message devait nécessairement se faire en une durée limitée*.

Qualité des diagnostics du compilateur

Elle est d'autant plus souhaitable que « une erreur coûte d'autant plus cher qu'elle est détectée plus tard ».

On attend un diagnostic précis, dans son énoncé comme dans sa localisation, alors même que les compilistes sous-estiment souvent les difficultés de leurs utilisateurs.

Précision de l'énoncé

Pour la précision de l'énoncé, les messages du genre « erreur de syntaxe » sont bien moins éclairants que « absence de point-virgule» pointant au bon endroit.

Le compilateur Simula 67 fourni par la CII pour l'Iris 80 comportait au départ 3 messages d'erreurs, que l'INRIA remplaça après refonte des premières phases par 185 messages.

Localisation de l'erreur

La localisation d'une erreur au caractère près est souvent suffisante.

Le compilateur Fortran fourni par Télémécanique pour son T2000 était particulièrement déviant. Sa refonte à l'INSA de Lyon – destinée à l'étendre en le rapprochant de la norme AFNOR – permit une localisation plus précise des erreurs, qui compensa largement le fait qu'il fallut, pour des raisons propres à cette machine, se limiter à 19 types d'erreurs.

Au-delà de l'erreur

Au-delà d'une erreur, il est bon de poursuivre l'analyse mais de façon assez stricte : un programme peut contenir beaucoup d'erreurs, mais certaines ne seront que des « erreurs induites » par une erreur précédente — ou la décision prise de poursuivre malgré tout.

Toutefois, la réparation d'une erreur ne dispense pas d'un nouveau test : « une erreur peut en cacher une autre », et toute réparation peut être imparfaite.

Organisation de la campagne de test

On suppose que les compilistes ont mené leurs propres tests. L'équipe de test, a priori différente, doit alors se considérer représentative des utilisateurs, directement ou au nom de l'organisme de certification.

[17] machine développée par Ulrich Finder (CNET), et comportant 2 à 16 processeurs hétérogènes.

La matière des tests doit tenir compte des caractéristiques et particularités du langage, comme des failles probables liées aux techniques de réalisation examinées précédemment.

Essais intensifs

Dans le cas où on dispose d'une petite équipe motivée, la campagne de tests peut être strictement planifiée. On commence alors par des tests élémentaires puis plus élaborés, exploitant ce qui est déjà vérifié.

Les programmes de tests se répartissent en 4 catégories.

Les programmes

catégories	**incorrect**	**correct**		
		invalide	**valide**	
			non pertinent	**pertinent**
définition	non-conforme au sens du langage L	accepté malgré un algorithme incorrect	exécution correcte	exécution conforme aux intentions
rôle dans un jeu d'essai	vérification de la nature et de l'emplacement de l'erreur diagnostiquée	vérification d'aspects douteux : variables non initialisées, indices ou arguments hors domaine		exemplaire

Le train d'essais

1. Il comporte d'abord des programmes élémentaires valides, dont les résultats sont connus ou aisément vérifiables (calculette…).

> Ex : édition d'une suite entière connue — un environnement de programmation éditait correctement les entiers, sauf le chiffre des centaines pour les nombres au-delà de 1000.

Les premiers programmes étant simplissimes, avec des objectifs étroits mais fondamentaux, les programmes suivants ne devraient exploiter ce qui a déjà été testé, afin de la compléter. Les derniers peuvent être consacrés aux tests de qualité de l'exécutif.

Définir pour chaque programme de test ses pré-requis (constructions ou aspects supposées sûrs) et ses objectifs (constructions ou aspects objets du test) facilite une organisation progressive du train d'essais, et l'interprétation des anomalies repérées.

Ces programmes doivent progressivement couvrir le langage : version officielle L, version spécifique documentée L'. La tendance sera d'abord à encourager l'exactitude et la rigueur de la documentation – et les difficultés de mise au point des programmes de tests pourront donner de précieux indices[18]. Au-delà, on poussera à restreindre au maximum les restrictions de L' par rapport à L, et à n'encourager que les *extensions rationnelles* de L (i.e. à lever les restrictions que L s'est inutilement infligé) au détriment des extensions exotiques.

2. Une deuxième vague sera formée de programmes incorrects ne comportant qu'une ou deux *erreurs délibérées et situées*, afin de vérifier la qualité de la détection d'erreurs : exactitude du diagnostic et précision de son positionnement.

3. Une autre vague pourra ensuite tester la zone grise formée des *programmes corrects au sens du langage, bien qu'invalides* en tant qu'algorithmes, comportant par exemple des varia-

[18] par exemple, sur la non-détection des variables non déclarées : est-ce licite ? si non, il y a faute ; si oui, la documentation (guide du programmeur...) avertit-elle l'utilisateur ?

bles non initialisées, des indices ou arguments hors domaine etc… notamment si le compilateur propose ce genre de contrôle sans que le langage l'exige officiellement, ou si on veut en établir la nécessité.

4. Une dernière vague orientée utilisateur sera formée de « programmes d'école » pour le langage, comprenant à la fois la reprise de quelques classiques, et quelques programmes démonstratifs mettant en valeur les spécificités du langage.

L'exigence d'*évolution non-régressive* devrait se traduire au fil du temps par une extension progressive du train d'essais.

Un exemple en Javascript

On suppose vérifiées *affectation et addition des variables entières simples, sorties élémentaires* et *boucles for.*

On se propose de vérifier *conditions ; instructions conditionnelles ; affectation et addition des variables entières indicées ; expressions conditionnelles et fonctions récursives.*

Pour cela, on exploite la suite de Fibonacci, suite entière à croissance exponentielle définie par :

$f_0=0$, $f_1=1$, et pour $i>1$, $f_i = f_{i-1}+f_{i-2}$.

calculable à l'aide d'une première boucle et de 3 variables simples figurant respectivement la nouvelle valeur (r) et les deux valeurs précédentes (s et t).

On profite de cette première boucle pour initialiser pas à pas un tableau Fib[] avec chaque nouvelle valeur ; pour i>1, cette initialisation permet de vérifier que l'on a toujours Fib[i] = Fib[i–1]+Fib[i–2], et que sous cet angle l'usage d'une table est sans problème (en ce qui concerne initialisation et rappel, addition et test d'égalité).

Une seconde boucle permet de vérifier que pour tout i on a bien Fib[i] = FiboR(i), où FiboR() est une fonction récursive de style lispien, exploitant une *expression* conditionnelle. Cette fonction pourrait être évidemment remplacée ou doublée par une fonction récursive plus classique utilisant une *instruction* conditionnelle.

Ainsi, la construction de la table a été vérifiée par rapport à un calcul plus simple présumé sûr, puis la table construite a été comparée à une fonction définie indépendamment, si bien qu'on peut présumer que la validité de la table garantit la validité de la fonction – tant que l'exécution est sans problème.

Cette exécution est ici paramétrée par LIM (qu'on pourrait lire… si la lecture d'entier est testée).

Sa valeur (ici 30) peut être augmentée, mais connaîtra deux limites :

- les entiers engendrés, croissant exponentiellement, doivent rester traitables par l'arithmétique,
- la fonction FiboR écrite est simple et pure, mais avec un temps d'exécution fonction exponentielle de l'argument, et la seconde boucle est alors sujette à explosion combinatoire.

Le code source (en environnement HTML) et l'exécution sont donnés ci-après.

Programme test javascript

```
<!DOCTYPE html>
<html>
 <head>
  <meta content="text/html; charset=windows-1252" http-equiv="content-type">
  <h1>Test Fibonacci</h1>
        <h2>prérequis</h2>
        affectation et addition des variables simples ; sorties élémentaires ; boucles for
        <h2> objet </h2>
        <p>conditions ; instructions conditionnelles ; affectation et addition des variables indicées ;
        expressions conditionnelles et fonctions récursives
        </p>
 </head>
 <body>
  <script type="text/javascript" language="javascript">

  function FiboR(n) { return (n<=1) ? n : FiboR(n-1)+FiboR(n-2);}

  var r=1, s=-1, t=0 ;
  var LIM= 30 ;

  var Fib = new Array();

  document.write("Conformité table<BR><BR>");

  for (var i=0 ; i<= LIM ; i++)
  {
   t=s ; s=r ; r=s+t ;
   Fib[i]=r ;
   if (i<2)
                {document.write(i+" ->\tval:\t"+r+";\ttable:"+Fib[i]+"<BR>");}
   else
                {document.write(i+" ->\tval:\t"+r+";\ttable:\t"+Fib[i]);
                 if((Fib[i-1]+Fib[i-2])==Fib[i]) document.write("\t...ok<BR>");
                 else document.write("\t!!!<BR>");
                }
  }
        document.write("<BR>Conformité table/fonction<BR><BR>");
        for (var i=0 ; i<= LIM ; i++)
  { r= FiboR(i);
   document.write("Fib("+i+")= "+r+" ? "+(Fib[i]==r)+"<BR>");
  }

  </script>
 </body>
</html>
```

Résultat test

Test Fibonacci

prérequis

affectation et addition des variables simples ; sorties élémentaires ; boucles for

objet

conditions ; instructions conditionnelles ; affectation et addition des variables indicées ; expressions conditionnelles et fonctions récursives

Conformité table

0 -> val: 0; table:0
1 -> val: 1; table:1
2 -> val: 1; table: 1 ...ok
3 -> val: 2; table: 2 ...ok
4 -> val: 3; table: 3 ...ok
5 -> val: 5; table: 5 ...ok
6 -> val: 8; table: 8 ...ok
7 -> val: 13; table: 13 ...ok
8 -> val: 21; table: 21 ...ok
9 -> val: 34; table: 34 ...ok
10 -> val: 55; table: 55 ...ok
11 -> val: 89; table: 89 ...ok
12 -> val: 144; table: 144 ...ok
13 -> val: 233; table: 233 ...ok
14 -> val: 377; table: 377 ...ok
15 -> val: 610; table: 610 ...ok
16 -> val: 987; table: 987 ...ok
17 -> val: 1597; table: 1597 ...ok
18 -> val: 2584; table: 2584 ...ok
19 -> val: 4181; table: 4181 ...ok
20 -> val: 6765; table: 6765 ...ok
21 -> val: 10946; table: 10946 ...ok
22 -> val: 17711; table: 17711 ...ok
23 -> val: 28657; table: 28657 ...ok
24 -> val: 46368; table: 46368 ...ok
25 -> val: 75025; table: 75025 ...ok
26 -> val: 121393; table: 121393 ...ok
27 -> val: 196418; table: 196418 ...ok
28 -> val: 317811; table: 317811 ...ok
29 -> val: 514229; table: 514229 ...ok
30 -> val: 832040; table: 832040 ...ok

Conformité table/fonction

Fib(0)= 0 ? true
Fib(1)= 1 ? true
Fib(2)= 1 ? true
Fib(3)= 2 ? true
Fib(4)= 3 ? true
Fib(5)= 5 ? true
Fib(6)= 8 ? true
Fib(7)= 13 ? true
Fib(8)= 21 ? true
Fib(9)= 34 ? true
Fib(10)= 55 ? true
Fib(11)= 89 ? true
Fib(12)= 144 ? true
Fib(13)= 233 ? true
Fib(14)= 377 ? true
Fib(15)= 610 ? true
……

Fib(26)= 121393 ? true
Fib(27)= 196418 ? true
Fib(28)= 317811 ? true
Fib(29)= 514229 ? true
Fib(30)= 832040 ? true

Essais extensifs

On suppose ici que l'on dispose d'une équipe nombreuse mais modérément motivée et/ou disponible[19]. On demande alors à l'équipe de test de se considérer comme représentative des utilisateurs : chacun développera *l'application de son choix* (le responsable veillant seulement à ce qu'elles soient diverses) *sans contrainte stylistique*, la force du test reposant ici sur le foisonnement des programmes.

A chaque erreur constatée (à l'analyse ou à l'exécution), le testeur devra d'abord vérifier qu'il a raison contre le produit – au sens de la documentation. Une statistique des erreurs avérées permettra alors au responsable de pointer les problèmes les plus critiques, en les précisant au mieux.

Cette méthode peut être assez efficace pour la mise au point d'un compilateur, si la communication est bonne entre responsable des tests et compilistes, permettant une interaction positive rapide.

Quelques leçons

Sur les langages

En Lisp, riche en exécutions énigmatiques et tests partiellement définis, il est difficile de détecter la moindre erreur à la saisie, à part « parenthèse droite en trop », et la syntaxe un peu plus développée de Logo constitue à cet égard un progrès.

En APL, tout repose sur nombre d'opérateurs ayant une signification unaire et une signification binaire, opérateurs représentés concrètement par 1 à 3 caractères suivant des conventions liées aux claviers. Là aussi il est difficile de détecter une erreur à la saisie, et les conventions spécifiques poussent à la faute.

Les « langages à point-virgule » manifestent souvent une sensibilité excessive : 30% des erreurs viennent d'une absence de point-virgule en fin de ligne (qui pourrait être une représentation secondaire) et d'autres d'un « ; » excédentaire (quand on n'admet pas d'instruction vide).

Le typage faible, l'absence de profilage, les nombreux implicites, les conversions automatiques, ces dispositifs sensés faciliter « la programmation astucieuse » poussent le programmeur à la faute et compliquent la maintenance.

A l'opposé, Modula 2 et les Adas ont montré à la fois l'intérêt de disposer de types de bas niveau, et celui de pouvoir en confiner strictement l'usage.

> Après avoir supervisé la réécriture de Siris 7 (100 000 lignes d'assembleur) puis conçu les langages LIS (Langage d'Implémentation de Système) et Ada 83, Jean Ichbiah remarquait qu'un système d'exploitation demandait tout au plus 10% de code de bas niveau.

Sur le développement des compilateurs

Notre expérience a montré l'intérêt de développer tous les compilateurs dans un langage du type « Compiler-Design Language », dont les contrôles sémantiques ont dû être progressivement durcis à la demande des compilistes, au fur et à mesure qu'on attaquait de plus vastes projets[20].

---

[19] cas limite : une horde d'étudiants

[20] le compilateur PL1/C comportait plus de 44 000 lignes LET auto-documentées engendrant 94 000 lignes C non commentées.

L'utilisation d'un seul et même langage pour tout le projet facilite les inspections croisées, et par là le dénouement de situations difficiles, par opposition à l'emploi d'outils différents dans diverses phases.

Remerciements

Je remercie J. Beney, C. Mauger, J.P. Doucet, M. Maranzana et les nombreux étudiants, doctorants et docteurs sans qui cette expérience n'aurait pas été acquise.

Références

Abramowitz M. & Stegun I., 1964-1999, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables,* Dover Publications, http://people.math.sfu.ca/~cbm/aands

Beney J., Frécon L., 1980, *Langage et système d'écriture de traducteurs*, RAIRO Informatique, vol 14, n°4, pp 379-394

Cleaveland J.C. & Uzgalis R.C., 1977, *Grammars for programming languages*, Elsevier.

Koster C.H.A., 1991, *Affix Grammars for Programming languages* ; in : H. Alblas and B. Melichar (Eds.), Attribute Grammars, Applications and Systems. LNCS 545, Springer Verlag pp 358-373.

Maranzana M., Martin J.F., Frécon L., 1990, *Un Traducteur de PL1/Multics vers C/VE*, TSI, vol.9 , n°5, pp.399-422

Muller J.M., *Elementary Functions, Algorithms and Implementation*, Birkhauser, Boston, 1997/2005

Ochem Q., 2004, *Cours d'Ada 95 pour le programmeur C++*, http://g3c.sourceforge.net/docs/c%2B%2B%20to%20ada.pdf

Roscoe A.W., C.H.R. Hoare, 1986, *The laws of occam programming*, Oxford University Computing Lab., Technical Monograph PRG-53, http://www.cs.ox.ac.uk/files/3376/PRG53.pdf

Sidhoum H., Babout M., Frécon L., *Ampère2, un langage de programmation pour la physique*, The European Journal of Physics, vol.11, 1990, pp.163-171

Wirth N., 1980, *Algorithmes et structures de données*, Eyrolles, trad. de *Algorithms + Data Structures = Programs,* 1976, Prentice-Hall Series in Automatic Computation.

Wirth N., 1984, *Programmer en Modula-2*, PPUR, Lausanne, trad. de *Programming in Modula-2*, 1982, Springer-Verlag, Berlin

Louis Frécon

Professeur d'Université (retraité)

louis.frecon@wanadoo.fr