Spécification d'un interpréteur de mini-langage algotithmique pour l'initiaition à la programmation

D. Deveaux, Michèle Raphalen, J. Revault

SPÉCIFICATION D'UN INTERPRÈTEUR DE MINI-LANGAGE ALGORITHMIQUE POUR L'INITIATION À LA PROGRAMMATION

D. Deveaux - M. Raphalen - J. Revault

Laboratoire d'Application
des Méthodes Statistiques et Informatiques
I.U.T. de Vannes
8 rue Montaigne - BP 1104
56036 VANNES CEDEX
Tél 97 63 26 09

RÉSUMÉ :

L'emploi des outils habituels de développement (éditeurs, compilateurs, etc.) dans l'initiation à l'algorithmique conduit à de nombreuses difficultés pratiques, parfois insurmontables par les étudiants. En réponse à ce problème, cet article présente les spécifications et les premiers éléments d'analyse d'un logiciel "d'aide à l'apprentissage de l'algorithmique" qui est en cours de réalisation à l'I.U.T. de Vannes.

Conçu dans un objectif de **formation professionnelle**, ce logiciel didactique doit servir de prototype pour l'étude et la réalisation ultérieures d'outils modulaires d'aide au développement dans un contexte de recherche de Qualité Industrielle.

SPÉCIFICATION D'UN INTERPRÈTEUR DE MINI-LANGAGE ALGORITHMIQUE POUR L'INITIATION À LA PROGRAMMATION

D. Deveaux - M. Raphalen - J. Revault

Laboratoire d'Application
des Méthodes Statistiques et Informatiques
I.U.T. de Vannes
8 rue Montaigne - BP 1104
56036 VANNES CEDEX

1. INTRODUCTION - POURQUOI CRÉER UN INTERPRÈTEUR

Souvent l'initiation à la programmation se transforme en "parcours initiatique" : les étudiants sont assaillis pendant une courte période par une quantité d'informations fondamentales et techniques nouvelles et il leur est demandé, en plus, de réfléchir et de comprendre.

En effet, comme le remarque J.Arsac, l'algorithmique est une discipline d'abord difficile car il ne s'agit pas de l'apprentissage de simples techniques, mais de l'acquisition d'une méthode de raisonnement s'appuyant sur des techniques [Ars84] [RiR85].

Toutes les expériences pratiques en pédagogie montrent qu'en cette matière, **l'expérimentation** est absolument nécessaire à la compréhension des concepts de base. L'emploi des outils habituels de l'informaticien (langages de programmation classiques) dans cette phase d'initiation oblige à une accumulation d'apprentissages simultanés : algorithmique, système d'exploitation, éditeur de textes, langage de programmation. La simultanéité de ces apprentissages crée la confusion dans nombre d'esprits et amène chez les étudiants des découragements souvent irréversibles.

Ces remarques montrent qu'il faudrait s'orienter, pour la première initiation, vers des outils spécifiques à la pédagogie ; en thermodynamique par exemple, on n'utilise pas un moteur d'automobile comme outil de démonstration, mais des expériences simples qui mettent bien en évidence le phénomène à observer. De même, pour l'initiation à la programmation, il faudrait disposer d'outils d'initiation qui soient orientés vers la mise en évidence des concepts fondamentaux de l'algorithmique, mais qui restent d'abord simple.

Actuellement, le choix et la définition de ces concepts de base ne fait pas encore l'unanimité parmi les informaticiens, ce qui ne facilite pas les choix pour la spécification d'outils. Cependant, on peut admettre que les concepts suivants doivent absolument être **compris et assimilés** par toute personne pratiquant l'informatique :

- Objets et environnement, états de l'environnement,
- notion d'action,
- compositions des actions (séquentielle, alternative, itérative),
- notion de cohérence entre les actions,
- procédures et passage d'arguments entre procédures,
- localisation et visibilité des objets.

2. CRITÈRES DE CHOIX POUR UNE SPÉCIFICATION.

2.1. Le cadre du développement

Notre réflexion s'est faite dans le cadre d'un enseignement d'I.U.T Informatique ; l'initiation doit ici conduire non à une simple utilisation secondaire de la programmation, mais à une véritable formation professionnelle débouchant sur des méthodes de travail strictes, compatibles avec l'industrie. Cette remarque initiale justifie certains choix fondamentaux évoqués ci-après, mais elle ne sous-entend pas qu'il faille des outils d'initiation différents pour donner une culture générale et pour former des professionnels ; au contraire, on peut déplorer que trop d'outils d'initiation existant, qui présentent par ailleurs un intérêt pédagogique certain, ignorent totalement les véritables méthodes de travail de l'informatique et donnent finalement de la discipline une fausse image.

Les enseignants d'informatique de l'I.U.T. de Vannes ont choisi d'intégrer une approche de la **Qualité Industrielle** dans l'apprentissage de la programmation. Cette recherche de qualité conduit à imposer des normes d'organisation et de présentation du travail très strictes ; le respect de ces normes suppose obligatoirement l'emploi **d'outils** qui permettront d'alléger les contraintes supplémentaires ainsi imposées [Afc86] [KeP76].

Il est apparu très rapidement que les outils de Génie Logiciel actuellement disponibles n'étaient pas utilisables dans une initiation, car ils supposent la maîtrise de concepts de base qui font justement l'objet de l'apprentissage. Nous avons donc été amenés à construire des petits outils qui demeurent très simples et faciles d'abord, mais qui habituent les étudiants au concept même d'outillage et préparent l'adaptation à de vrais logiciels professionnels.

Ces logiciels ont été conçus dans l'esprit de la **boîte à outils Unix**, dont certains éléments ont d'ailleurs été repris (make, ar, grep, ...) ; ils comprennent un éditeur structuré de langage algorithmique (L.A.), un traducteur automatique de L.A. en C, des

programmes d'analyse statique de code source (références croisées, etc.) et un gestionnaire de documentation [KeP84] [Dev87]. Ces programmes sont utilisés depuis deux années en pédagogie, et ils ont constitué pour nous des prototypes sur lesquels il a été possible d'analyser plus finement les besoins et les aspects ergonomiques qui facilitent la compréhension.

Dans la même optique, notre Laboratoire s'intéresse aussi au développement d'outils de Génie Logiciel légers (exécutables sur toutes machines) et faciles d'apprentissage. De tels outils pourraient contribuer à combler le trou qui existe actuellement entre la pratique quotidienne de la plupart des informaticiens (qui ne sont pas dans des grands centres) et les ateliers de Génie Logiciel développés par les centres de recherche. Un outillage simple et facile d'emploi, s'appuyant sur les acquis récents de la recherche en Génie Logiciel, mais ne nécessitant pas l'emploi de technologie trop lourde, ne pourra évidemment pas prétendre égaler les caractéristiques de qualité et de fiabilité d'un véritable atelier intégré, mais permettrait à un grand nombre de praticiens qui ne peuvent avoir accès à ces ateliers, d'améliorer notablement leurs conditions de travail.

L'outil d'initiation dont nous avons commencé la réalisation évoluera à court terme vers des logiciels professionnels d'aide au développement.

2.2. Les principaux choix

Le logiciel en développement (nous l'appellerons IML dans la suite) est un interpréteur de langage de description, il peut donc **exécuter** immédiatement un texte source écrit dans ce langage. Comme tout interpréteur, il doit également gérer un ensemble de **commandes de base** demandant le chargement, l'exécution, la sauvegarde, etc., d'un programme. Enfin, pour l'efficacité du travail, il comporte une **fonction d'édition** qui permet des modifications dynamiques du programme en cours.

L'emploi pédagogique et l'approche "qualité" du programme IML amènent à attacher une grande importance à la forme :

- Forme du langage employé qui doit avoir des structures claires, non redondantes et efficaces.
- Forme de la présentation : l'outil impose l'organisation du texte (indentation, sauts de ligne) et l'emploi de commentaires aux points stratégiques.
- Forme de l'interface utilisateur où les difficultés d'ordre techniques sont gommées au maximum. Cet interpréteur est avant tout un **support de raisonnement** d'usage immédiat.

Lors de la conception d'algorithmes, **le raisonnement n'est pas linéaire** ; l'analyse descendante, en particulier, suppose que l'on décrive rapidement une action (par une phrase courte) que l'on détaillera ultérieurement en séquence d'instructions. Pour supporter

ce mode de fonctionnement du développement, il nous a semblé intéressant d'introduire la notion de **commentaire actif** : un commentaire actif se distingue du vrai commentaire par le fait qu'il représente en fait une séquence de code exécutable qui n'est pas encore développée. Sur le plan lexical, ces commentaires actifs auront une représentation différente :

```
/* ceci est un vrai commentaire */
<< ceci est un commentaire actif >>
```

Pour le développement, l'éditeur intégré dans IML a des fonctionnalités de gestion pleine-page permettant le retour sur des commentaires actifs et dispose d'une **fonction d'expansion** qui autorise le remplacement du commentaire actif par la séquence de code correspondante.

Au moment de l'exécution, les commentaires actifs ne sont évidemment pas exécutables directement, mais un mécanisme particulier d'exécution permet le test partiel d'algorithmes incomplètement codés : le texte du commentaire est affiché et la possibilité de modifier manuellement l'environnement est alors donnée avant la poursuite de l'exécution.

La manipulation des structures demande une attention particulière. En effet, au niveau syntaxique, une structure a le même statut qu'une instruction simple (affectation ou appel de procédure). Cette équivalence est difficile à assimiler, car une structure s'écrit sur plusieurs lignes et utilise plusieurs mots clés. Elle peut donc donner l'impression de mettre en jeu plusieurs instructions. Une structure doit donc apparaître comme **une** unité syntaxique, autant dans la forme du texte source que dans la manipulation dans un éditeur structuré. La création d'une structure doit être faite par une seule touche de fonction, sa destruction ne peut être que globale.

Enfin, pour l'efficacité du travail, les erreurs lexicales et syntaxiques sont détectées d'emblée et toutes ensembles (comme dans un compilateur) et non au fur et à mesure de l'exécution. Cet élément est une condition nécessaire pour l'apprentissage de méthodes d'organisation du travail qui deviendront indispensables dans la suite de la formation.

2.2.1. L'exécution - Le but étant de faciliter la compréhension du fonctionnement des algorithmes, l'exécution peut se faire suivant deux modes principaux, le **mode direct** où l'on a une exécution brute ne comportant que les affichages demandés par l'algorithme, et le **mode tracé** où l'interpréteur peut afficher tout ou partie de l'environnement des objets visibles ; ce mode tracé comporte plusieurs options de déroulement et d'affichage (mode pas à pas avec affichage de tout l'environnement ou d'objets sélectionnés, points d'arrêt sur adresses choisies ou modification d'objets). Le mode tracé est en fait très comparable dans son fonctionnement à un débogueur symbolique.

2.2.2. Les commandes de base - En nombre aussi restreint que possible, elles permettent le chargement et la sauvegarde d'un programme, le lancement d'exécution, le recours indirect à des commandes du système sous-jacent. Pour la première version du programme IML, un seul fichier peut être chargé à la fois, mais la structure retenue devra permettre ultérieurement le chargement de plusieurs modules et la gestion d'ordres "include". Compte-tenu des remarques faites sur la forme ci-dessus, il semble judicieux d'imposer lors de la sauvegarde un format standard des textes de programmes ; en effet, la première démarche lors de la recherche de qualité, consiste à définir des normes de présentation des textes (de code et de documentation). Il apparaît d'ailleurs que la contrainte d'une rigueur sur la forme facilite l'apprentissage de la rigueur dans le raisonnement. En la matière, l'effet d'exemple et d'habitude est bien réel.

Les autres commandes permettent de demander les divers modes d'exécution prévus et d'appeler les principales fonctions utilitaires de base (visualisation de textes et de répertoires).

2.2.3. L'éditeur - Les spécifications de cette fonction, notamment l'ergonomie, ont une grande influence sur l'efficacité du produit. A coup sûr, de nombreuses modifications seront nécessaires après la mise en service du logiciel pour l'adapter aux réactions des utilisateurs ; en conséquence, le développement est aussi modulaire que possible et le code est particulièrement documenté. Nous noterons ici seulement quelques remarques de base. Les parties de déclaration et de définition d'actions sont visuellement différentes pour faire sentir d'emblée la différence entre les instructions exécutables et les autres. On a retenu une interface de type "grille d'écran" pour les déclarations (objets et procédures) et de type éditeur pleine page pour la description des actions.

L'éditeur est un éditeur structuré dont les principales caractéristiques sont :

- génération automatique des mots clés par touches de fonction. Dans le cas de structures, l'ensemble du squelette est généré par une seule touche de fonction, avec les champs variables sous forme de commentaires actifs. Aucun mot clé n'est modifiable.
- mise en évidence de la structure du code à l'aide d'indentation,
- demande automatique de commentaires aux endroits judicieux (fonctionnement des procédures, rôle des objets, etc), et ceci **avant la description des actions** dans le langage algorithmique.

La conception de cet éditeur n'est pas limitée au seul langage algorithmique minimum, mais elle doit permettre une évolution vers la fabrication d'éditeurs structurés adaptés aux "vrais" langages de programmation.

3. SPÉCIFICATIONS DU LOGICIEL IML

3.1. Le langage algorithmique minimum (MLD)

Comme nous l'avons indiqué ci-dessus, le but de ce langage n'est pas de fournir un outil exhaustif permettant de traiter toutes les applications, mais de constituer un support minimum aux premières expérimentations en algorithmique. On peut donc le considérer comme un sous-ensemble d'un véritable langage algorithmique opérationnel, dans lequel toutes les notions et éléments non strictement nécessaires à la compréhension des concepts de base auraient été "élagués".

Ce langage minimum a été élaboré à partir d'un langage algorithmique défini dans le cadre d'une association d'utilisateurs (A.U.M.E.R.) et utilisé depuis deux ans en pédagogie à l'I.U.T. de Vannes. Ce langage général est déjà à la base d'un ensemble de prototypes d'outils de Génie Logiciel utilisés avec les étudiants (éditeur syntaxique, traducteur automatique en langage C, gestionnaires de documentation) [Dev86].

Le langage a été entièrement spécifié sous forme de grammaire Backus-Naur (BNF) ; cette grammaire de 63 règles est présentée en annexe, ainsi que deux exemples d'algorithmes écrits en mini-langage. Nous détaillerons ci-dessous les principaux choix réalisés.

3.1.1. Types d'objets - Il n'est pas nécessaire pour comprendre les principaux concepts de disposer d'une grande variété de types d'objets. Un très grand nombre d'applications de démarrage peut être traité avec trois types seulement : booleen, caractere et entier. Il semble également utile de définir la notion de **vecteur** d'objets (tableau à une dimension) appliquée à ces trois types de base. Les objets de type **chaîne de caractères** sont manipulés sous forme de tableaux de caractères, les fins de chaînes étant marquées par un code spécifique (EOS).

L'interpréteur sait manipuler de façon interne des objets de type pointeur sur les trois types définis ; cette fonctionnalité pourra ultérieurement être rendue visible pour des applications plus avancées.

3.1.2. Les actions - Les actions simples sont soit les affectations, soit les appels de procédures (voir ci-dessous). Dans les affectations, le strict respect des types est obligatoire, de même qu'entre les opérandes d'une expression ; des fonctions prédéfinies permettent la conversion de types.

Une action composée est constituée d'une séquence d'actions simples comprise entre les signes '{' et '}'.

3.1.3. Compositions des actions - Les trois modes de composition sont représentés :

-composition séquentielle : les instructions successives d'une séquence sont séparées par un ';'.

-composition alternative, représentée par la structure suivante :

```
teste <<condition>>      <<commentaire>>
 si_oui
  <<action>>
 si_non
  <<action>>
```

une des deux composantes (si_oui... ou si_non...) peut être omise. Il n'a pas semblé nécessaire pour un langage minimum de prévoir une structure spéciale pour le choix multiple.

- composition itérative : en suivant G.Clavel et Knuth [BiC81] [Knu74] nous n'avons retenu qu'une forme de schéma général d'itération :

```
iterer
  <<action>>
  quand <<condition_de_sortie>> faire
   [<<action>>]
   sortie
  <<action>>
fin_iterer
```

ce schéma est moins populaire dans les langages de programmation que les schémas tant_que et Repeter, mais il présente l'avantage de dissocier l'idée d'itération du choix de la condition de sortie ; l'association de ces deux concepts dans les schémas d'itération pose de nombreux problèmes aux débutants. Il s'avère plus simple de montrer ultérieurement comment une boucle iterer "dégénère" en tant_que ou Repeter suivant la position du test de sortie.

Dans le même esprit, nous avons autorisé la définition d'une action liée à la sortie de boucle ; sur un plan strictement fondamental, ce choix nuit à l'homogénéité du langage et constituerait une gêne à la démonstration des programmes ; cependant, dans de nombreux cas pratiques, cette possibilité s'avère très intéressante en augmentant la clarté du programme et en facilitant la définition immédiate d'une action qui vient à l'esprit du programmeur quand il définit le test de sortie.

Par souci de pragmatisme encore, l'occurrence de plusieurs tests de sortie sera autorisée à l'intérieur d'une itération, même si cette pratique n'est pas encouragée.

3.1.4. Les procédures - dans le mini-langage, comme dans son prédécesseur, les déclarations de procédures ne peuvent pas être imbriquées. En effet, si l'imbrication des procédures (comme en Pascal) est une bonne traduction des mécanismes de l'analyse descendante, cette forme présente un certain nombre d'inconvénients :

- Même s'il est le plus facile à expliquer et à justifier, le mécanisme d'analyse descendante n'est pas la seule démarche utilisable. Dans le cas où l'on fait une autre approche, l'imbrication obligatoire des procédures devient une contrainte très difficile à gérer.
- Dès que les programmes prennent quelque importance, l'imbrication des procédures oblige à insérer une grande quantité de codes entre la zone des déclarations et la zone de description des actions d'une procédure, ce qui nuit considérablement à la lisibilité des programmes écrits. Un autre problème de lisibilité est lié à l'empilement excessif des indentations qui, au-delà de cinq ou six niveaux, arrivent à égarer complètement le lecteur.
- Les règles de localisation des objets (visibilité d'un objet par toutes les procédures filles) provoquent des difficultés de représentation chez les débutants et peuvent conduire à des effets de bords car le partage d'objets est souvent surabondant.

L'emploi de procédures toujours indépendantes évite ces problèmes et simplifie considérablement les règles de localisation : il n'existe que deux classes d'objets. Les objets déclarés dans les procédures sont strictement locaux à ces procédures, ceux qui sont déclarés à l'extérieur sont visibles de toutes les procédures. Ultérieurement, lorsque les logiciels écrits seront plus importants (on emploiera alors un vrai langage de programmation), on pourra introduire la notion de modules (ou package) et définir des objets locaux ou partagés entre modules.

Il nous semble que ce choix, en limitant l'échange d'information entre procédures aux objets strictement définis comme partagés (objets globaux et arguments d'appel), est plus conforme aux règles de **visibilité restreinte** définies lors des approches de Génie Logiciel visant à la Qualité.

Les arguments des procédures peuvent appartenir à trois pseudo-classes :

- DONNEE : l'argument est utilisé comme donnée par la procédure, il est garanti que l'argument effectif ne sera pas modifié par la procédure, il doit avoir été initialisé avant l'appel.
- RESULT : l'argument est rendu par la procédure comme résultat, si l'argument effectif avait une valeur dans le programme appelant avant l'appel, cette valeur est ignorée par la procédure et écrasée lors de son exécution.
- D_R : l'argument est à la fois données et résultat, la valeur de l'argument effectif est utilisé par la procédure puis rendu comme résultat.

Cette définition de pseudo-classe ne fait pas apparaître la notion de passage de paramètre par valeur ou par adresse. En fait la description de ces techniques n'a pas sa place dans un enseignement d'algorithmique, et doit apparaître quand on évoque les techniques de compilation ou dans l'apprentissage des langages de programmation. Toutefois, il est toujours possible de simuler les passages d'arguments par adresse en utilisant explicitement des pointeurs.

Il n'y a pas de distinction entre "procédures" et "fonctions" ; les algorithmes paramétrés (comme les appelle G. Clavel) sont obligatoirement définis avec un type (simple ou pointeur) et auraient donc un statut de fonction dans la terminologie habituelle. Il existe cependant deux types spécifiques aux objets "procédures" :

- vide : permet de définir les "vraies" procédures sans retour de valeur.
- PROG : désigne la procédure "programme principal" ; un programme doit contenir une et une seule procédure de type PROG qui définit le point d'entrée par lequel l'interpréteur commence l'exécution. A part celà, le programme principal a la même structure et ne présente aucune différence de statut par rapport aux autres procédures du programme.

Lors de l'exécution, les appels de procédures de type vide ont le même statut qu'une action, les appels des autres procédures ont un statut de facteur dans une expression (voir la grammaire).

3.2. Fonctionnement de l'interpréteur

Le programme IML est construit autour d'une structure de données centrale gérée en mémoire dynamique ; cette structure de données est construite à partir de trois éléments de base :

- des arbres binaires balancés pour le stockage des **symboles** (constantes, variables, procédures).
- des structures arborescentes pour le stockage des actions : une organisation de noeud spécifique est définie pour chaque structure du langage (procédure, teste, iterer, etc.), les séquences d'actions sont traitées comme des listes d'instructions (les listes sont doublement chaînées pour faciliter l'édition).

- des arbres binaires pour le stockage et l'exécution des expressions.

Autour de cette structure, le programme est conçu sous forme de quatre modules presque indépendants : un module de chargement de la structure à partir d'un texte source, un module d'exécution du programme, un module de sauvegarde de programme, un module éditeur. Ce développement modulaire permet un prototypage très rapide du projet.

3.2.1. Module de chargement - Il est constitué d'un analyseur lexical et d'un analyseur syntaxique qui traduisent le texte source dans la structure interne. L'analyseur lexical comporte deux niveaux ; un niveau indépendant du langage détecte les entités lexicales simples (mots, nombres, ponctuations, commentaires, etc), un niveau supérieur gère des tables de définition des mots clés spécifiques du langage traité (cet analyseur lexical est utilisable pour tout langage avec de faciles adaptations).

L'analyseur syntaxique est construit en programmation récursive à partir de la grammaire du langage. Un point d'entrée particulier est prévu au niveau de la règle <<instruction_simple>> pour faciliter l'interfaçage avec l'éditeur qui travaille, lui, en mode ligne.

L'exécution d'analyses lexicales et syntaxiques complètes au moment de la lecture du fichier source permet une détection immédiate de toutes les erreurs triviales, comme dans le cas de l'utilisation d'un compilateur.

3.2.2. Module d'exécution - Il travaille directement à partir des structures arborescentes en utilisant une programmation récursive. De façon interne, tous les objets simples sont traités comme des entiers. L'évaluation des expressions n'utilise pas de processeur à pile, mais travaille directement à partir de l'arbre de représentation interne.

3.2.3. Module de sauvegarde - Il fabrique un nouveau texte de programme à partir de la représentation interne arborescente. Cette technique garantit que la présentation des textes générés est très homogène et que les notations (parenthésage des expressions notamment) sont optimales.

L'organisation de programme adoptée laisse la voie ouverte pour des générations automatiques de programmes dans d'autres langages de programmation que le mini-langage algorithmique. A ce stade, il n'est pas difficile d'imaginer des modules additionnels de génération de langage C ou Ada par exemple.

3.2.4. Module d'édition - Le développement de ce module s'appuie sur les acquis de notre équipe, développés dans des applications précédentes (version locale de Emacs, éditeur syntaxique, gestionnaire d'écrans et de masques de saisie). Il utilise des techniques de multifenêtrage pour la visualisation de diverses zones ou de divers niveaux déjà écrits, les consultations de documentation ou de listes d'objets en cours de travail.

3.3. Spécifications techniques

L'ensemble du logiciel IML est réalisé en langage C ; comme les autres logiciels développés à l'I.U.T. de Vannes, il sera adapté sur compatibles PC (avec au moins 256 Ko), Micro-VAX sous VMS et SUN sous UNIX.

4. ÉTAT DU PROJET, ÉVOLUTIONS

En plus des spécifications, une part importante de l'analyse et du codage est déjà réalisée, soit en développement spécifique au projet IML (définition de la structure de données, analyseurs lexical et syntaxique), soit sous forme de prototypes dans des logiciels développés préalablement : gestion de clavier et d'écran dans Emacs-Vannes, création d'objets à partir de grilles d'écran dans l'éditeur syntaxique, etc...

L'élaboration du logiciel est structurée en étapes dont les points intermédiaires sont fonctionnels et utilisables. Ainsi la première phase du développement porte sur les modules de chargement et de sauvegarde, et permet la validation de la structure de données retenue.

Ensuite, l'installation du module d'exécution aboutira à un produit certes peu confortable (les corrections devront se faire dans un éditeur extérieur), mais utilisable.

La mise en place de l'éditeur qui demande une mise au point plus longue sera faite en dernière phase.

Compte-tenu de l'avancement actuel des travaux, un prototype complet du logiciel devrait être opérationnel avant la fin de l'année 1988. Il pourra alors faire l'objet d'une validation pédagogique, si possible avec plusieurs équipes réparties dans différents centres et s'adressant à des publics variés.

5. CONCLUSION

Le logiciel IML que nous proposons a été conçu pour répondre à certains problèmes de pédagogie de l'algorithmique. Les principales difficultés que nous avons rencontrées dans cette spécification tiennent à la définition du sujet lui-même. En effet, comme nous l'avons déjà mentionné, les avis des spécialistes en algorithmique sont loin d'être convergents sur les concepts et sur leurs modes de présentations : ainsi voit-on coexister des approches procéduriales, par objets ou logiques, très formelles (spécifications algébriques, ...) ou intuitives qui souvent s'affrontent en "guerres de religions", mais qui doivent en fait être les différentes facettes d'une même recherche en cours. Dans ce contexte, il nous a semblé important de réaliser rapidement un logiciel permettant d'améliorer l'apprentissage de l'algorithmique, sans attendre un consensus sur les méthodes (est-ce possible ?). Ce logiciel reste fondé sur une approche procéduriale classique, bien que sa forme et sa présentation donnent un rôle particulier à la définition des objets : il apparaît en effet que l'approche orientée objets fournit aux programmeurs un moyen de structuration exceptionnel.

Nous ne cherchons pas, à travers cet outil, à imposer un modèle de représentation algorithmique ou un modèle pédagogique ; il est très probable que l'expérimentation du

logiciel conduira à des remises en question ou à des extensions des choix que nous avons pu faire ici. L'extension de cet interpréteur à un compilateur de langage algorithmique complet ne nous semble pas être une opération judicieuse. Quand ils ont été introduits (dans les années 1970), les langages algorithmiques apparaissaient comme un outil indispensable pour l'expression de la structure algorithmique que les langages de programmation (essentiellement COBOL et FORTRAN) ne savaient pas alors représenter. Les langages de programmation enseignés actuellement sont structurés et permettent une écriture directe des programmes, même pour la définition des structures de base sous forme algorithmique (les suites d'instructions simples étant exprimées sous forme de commentaires actifs, par exemple...).

Un mini-langage algorithmique comme celui que nous avons défini doit uniquement permettre de comprendre et d'appliquer les concepts d'algorithmique, ce n'est pas un langage de programmation. Il ne nous semble pas utile d'en prolonger l'emploi au-delà de cette période d'initiation (sauf comme support d'outils), la programmation n'est pas un exercice de thème.

Il est plus judicieux de construire, à partir d'IML, un éditeur structuré de "vrai" langage de programmation tel que C, PASCAL ou ADA, qui regrouperait des fonctions d'édition et de contrôle de cohérence. Les recherches à propos de nouveaux langages et de nouveaux outils doivent maintenant s'orienter vers l'aide à la spécification et au raisonnement.

Enfin, ce logiciel constitue pour nous un point de départ, d'une part pour une analyse plus approfondie de la pédagogie de la programmation, d'autre part pour la réalisation d'outils plus évolués dont il va constituer à de nombreux égards un prototype rapidement expérimentable.

BIBLIOGRAPHIE

[Afc86] AFCIQ. *Recommandation de plan qualité logiciel*. 1986 - Recommandation AFNOR - s-FR-B - Paris

[Ars84] J. ARSAC. *La conception des programmes*. 1984 - Pour la Science n.85 - nov - Paris

[Ars85] J. ARSAC. *Les bases de la programmation*. 1985 - Dunod - Paris

[BCC87] M. BIDOIT - F CAPY - F. CHOPPY. ET AL. ASSPRO. *Un environnement de programmation interactif et intégré*. 1987 - T.S.I vol 6 n.1 - Paris

[BaC80] BARRE - COUVERT. *Algorithmique : langage de description*. 1980 - Polycopié d'enseignement - Univ. Rennes 1

[BiC81] J. BIONDI - G. CLAVEL. *Introduction à la programmation*. 1981 - Masson - Paris

[Dev86] D. DEVEAUX. *Favoriser la diffusion des logiciels....* 1986 - Rapport interne A.U.M.E.R. -

[Dev87] D. DEVEAUX. *Editeur syntaxique ELD* : MODE D'EMPLOI. 1987 - Document de cours, I.U.T. de Vannes -

[FBG84] G. FAYET - BIENAIME - GIGOUT - RIELA. *Les outils du Génie Logiciel*. 1984 - I.N.R.A. - rapport interne - Nancy

[Gra86] ANNA GRAM. *Raisonner pour programmer*. 1986 - Dunod - Paris

[KeP76] KERNIGHAN - PLAUGER. *Software Tools*. 1976 - Addison Wesley -

[KeP84] B.W. KERNIGHAN - R. PIKE. *L'environnement de programmation UNIX*. 1984 - InterEditions - Paris

[KeR78] B.W. KERNIGHAN - D.M. RITCHIE. *The C programming language*. 1978 - Prentice-Hall Software Series -

[Knu73] D.E. KNUTH. *The art of computer programming*. 1973 - Addison Wesley -

[Knu74] D.E. KNUTH. *Structured programming with go to statements*. 1974 - ACM computing surveys vol 5 n. 4

[LPS83] M. LUCAS - PEYRIN - SCHOLL. *Algorithmes et représentation des données*. 1983 - Masson - Paris

[LuS75] M. LUCAS - SCHOLL. *Propositions pour une initiation à l'algorithmique*. 1975 - Univ. de grenoble - Grenoble

[Nee86] D. NEEL. *Guide pour l'établissement des spécifications d'un logiciel*. 1986 - C.N.R.S. - PIRSEM (doc interne) - Paris

[Nee87] D. NEEL. *La notion de qualité dans le cas des logiciels*. 1987 - Genie Logiciel - Paris

[RiR85] C. RICHARD - P. RICHARD. *Programmatique. Initiation à la programmation méthodique*. 1985 - Belin - Paris

[Wir76] N. WIRTH. *Algorithms + Data Structures = programs*. 1976 - Prentice Hall -

[You84] S.J. YOUNG. *An Introduction to ADA*. 1984 - Ellis Horwood -

ANNEXE A

Grammaire du Mini-Langage

Expression des règles syntaxiques du "mini-langage de description" en utilisant la forme de Backus-Naur (BNF).

Notes :

- les symboles suivants sont des méta-symboles du formalisme BNF et nondes symboles :

<<...>>	nom de règle
::=	introduit la définition d'une règle
|	séparateur entre termes alternatifs
^	terme vide

- les règles en caractères gras sont des règles décodées par l'analyseur lexical.

```
<<programme>>        ::= <<comment>>
                         <<def_const>>
                         <<decl_var_g>>
                         PROG <<identificateur>>
                         <<bloc>>
                         <<serie_proc>>

<<comment>>          ::= ^ | /* <<serie_car>> */
<<serie_car>>        ::= ^ | <<id_car>> <<serie_car>>
<<id_car>>           ::= <<caract>> | \ <<nombre>> | \n | \t | \f
<<caract>>           ::= [\001 - \176]
<<nombre>>           ::= <<signe>> <<chiffre>> <<serie_chiffre>>
<<signe>>            ::= - | ^
<<chiffre>>          ::= [0 - 9]
<<serie_chiffre>>    ::= ^ | <<chiffre>> <<serie_chiffre>>

<<def_const>>        ::= <<comment>>
                         <<serie_def_cst>>

<<serie_def_cst>>    ::= ^ | <<def1cst>> <<serie_def_cst>>
<<def1cst>>          ::= const <<ident>> <- <<valeur>> fcst <<comment>>
<<ident>>            ::= <<lettre>> <<suite_id>> | _ <<suite_id>>
<<lettre>>           ::= [a - z] | [A - Z]
<<suite_id>>         ::= ^ | <<lettre>> <<suite_id>> |
                         <<chiffre>> <<suite_id>> | _ <<suite_id>>
<<valeur>>           ::= <<nombre>> | <<car>> | <<chaine>>
<<car>>              ::= '<<id_car>>'
<<chaine>>           ::= " <<serie_car>> "
```

```
<<decl_var_g>>      ::= <<commentaire>>
                        <<serie_decl_var>>

<<serie_decl_var>>  ::= ^ | <<decl1var>> <<serie_decl_var>>
<<decl1var>>        ::= <<declvars>> | <<decltabl>>
<<declvars>>        ::= <<type>> <<ident>> <<initvar>> ; <<comment>>
<<type>>            ::= caractere | entier | booleen |
                        pt_caractere | pt_entier | pt_booleen
<<initvar>>         ::= ^ | := <<valini>>
<<valini>>          ::= <<nombre>> | <<ident>> | <<expression>>
<<expression>>      ::= <<expr_simple>> | <<expr_simple>> <<op_rel>> <<expr_simple>>
<<expr_simple>>     ::= <<terme>> | <<terme>> <<op_add>> <<expr_simple>>
<<terme>>           ::= <<facteur>> | <<facteur>> <<op_mul>> <<terme>>
<<facteur>>         ::= <<variable>> | <<valeur>> | ( <<expression>> ) |
                        <<appel_proc>> | <<op_un>> <<facteur>>
<<variable>>        ::= <<ident>> | <<ident>> [ <<expression>> ] |
                        * <<ident>>
<<appel_proc>>      ::= <<ident>> ( <<liste_arg>> )
<<liste_arg>>       ::= ^ | <<arg>> <<suite_arg>>
<<arg>>             ::= <<variable>> | <<valeur>> | <<appel_proc>>
<<suite_arg>>       ::= ^ | , <<arg>> <<suite_arg>>

<<op_rel>>          ::= = | != | > | < | >= | <=
<<op_add>>          ::= + | - | ou
<<op_un>>           ::= - | non
<<op_mul>>          ::= * | / | mod | et

<<decltabl>>        ::= <<type>> <<ident>> [<<valini>>] <<inittab>> ; <<comment>>
<<inittab>>         ::= ^ | := { <<serie_valini>> } | := <<chaine>>
<<serie_valini>>    ::= <<valini>> <<suite_valini>>
<<suite_valini>>    ::= ^ | , <<valini>> <<suite_valini>>

<<bloc>>            ::= Debut
                          <<decl_var_l>>
                          <<serie_instruc>>
                          Fin

<<decl_var_l>>      ::= <<comment>>
                        <<serie_decl_var>>
<<serie_instruc>>   ::= ^ | <<instruction>> <<serie_instruc>>
<<instruction>>     ::= <<instruc_simple>> | <<structure>>

<<instruc_simple>>  ::= ; | <<comment>> | <<com_act>> ; |
                        <<affectation>> ; | <<appel_proc>> ; |
                        retour ( <<expression>> ) ;
<<com_act>>         ::= << <<serie_car>> >>
<<affectation>>     ::= <<variable>> := <<expression>>
<<structure>>       ::= <<struc_alter>> | <<struc_iter>>
```

<<struc_alter>> ::= **teste** <<condition>>
si_oui
<<action>>
si_non
<<action>>

<<condition>> ::= <<expression>>

<<action>> ::= <<instruction>> | **{**
<<serie_instruc>>
}

<<struct_iter>> ::= **iterer**
<<action_a_sortie>>
fin_iterer

<<action_a_sortie>> ::= <<aas>> <<serie_aas>>

<<aas>> :: <<serie_instruc>>
quand <<condition>> **faire**
<<serie_instruc>>
sortie
<<serie_instruc>>

<<serie_aas>> ::= ^ | <<aas>> <<serie_aas>>

<<serie_proc>> ::= ^ | <<procedure>> <<serie_proc>>

<<procedure>> ::= <<entete_proc>>
<<bloc>>

<<entete_proc>> ::= <<comment>>
<<type>> <<ident>> (<<serie_decl_arg>>)
<<comment>>

<<serie_decl_arg>> ::= ^ | <<dec1arg>> <<serie_decl_arg>>

<<dec1arg>> ::= <<class_arg>> <<type>> <<identr>> ; <<comment>>

<<**class_arg**>> ::= **DONNEE** | **RESULT** | **D_R**

ANNEXE B

Exemples de programmes

```
/*
 *    Le programme COMPTE_LG est un exemple de programme simple
 *    Son objet est de compter les lignes d'un texte lu sur l'entree
 *    standard par litc().
 */

/*_____________________ Declaration des constantes _____________________*/

const EOF      <-    <<fin_de_fichier>>  fcst  /* code de fin de fichier   */
const NL       <-    10                  fcst  /* code ascii de fin de ligne */


PROG  compte_lg

/*_____________________________ Code du programme _____________________________*/
Debut
   caractere   c ;                  /* caractere courant        */
   entier      nl ;                 /* compteur de ligne        */

   nl := 0 ;
   iterer
      c := litc () ;
      quand  c = EOF  faire
         sortie
      teste  c = NL
       si_oui
         incr (nl) ;
      fin_iterer
   ecrn (nl) ;
   Fin
/*===FPROG ____________________________________________________________________*/
```

Les fonctions litc(), incr(), ecrn(), ecrf(), ... qui apparaissent ci-contre sont des fonctions prédéfinies du langage gérées directement par l'interpréteur.

```
/*
 *    Le programme TEST_PUIS est un exemple de programme multiprocedure
 *    Son objet est de definir et tester une fonction puissance simple
 */

/*_____________________ Declaration des constantes _____________________________*/

const MAXI    <-   10          fcst  /* valeur maxi de l'exposant  */
const VAL1    <-   2           fcst  /* premiere valeur de test    */
const VAL2    <-   -3          fcst  /* seconde  valeur de test    */

/*=====PROC Programme principal ____________________________________________*/

PROG test_puis          /* teste la fonction puissance _____________________*/

/*
 *----PROC
 */
Debut
  entier      i ;

  i := 0 ;
  iterer
    ecrf ("%d %d %d\n", i, puis (VAL1, i), puis (VAL2, i)) ;
    quand  i = MAXI  faire
      sortie
    incr (i) ;
    fin_iterer
  Fin
/*===FPROC ____________________________________________________________________*/

/*=====PROC puis         : eleve le nombre x a la puissance n _______________*/

entier   puis       (
              DONNEE entier  x ;       /* mantisse                  */
              DONNEE entier  n ;       /* exposant >= 0               */
                    )
/*
 *    La fonction eleve le nombre entier x a la puissance n avec n >= 0
 *----PROC
 */
Debut
  entier      i, p ;

  p := 1 ;
  i := 0 ;
  iterer
    quand  i >= n  faire
      sortie
    p := p * x ;
    incr (i) ;
    fin_iterer
  Fin
/*===FPROC ____________________________________________________________________*/

/*===FPROG ____________________________________________________________________*/
```