EDIS: UN ÉDITEUR DE SYMBOLES AVEC VÉRIFICATEUR SYNTAXIQUE

THÈSE N° 494 (1983)

PRÉSENTÉE AU DÉPARTEMENT DE MATHÉMATIQUES

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

NORBERT EBEL

Licencié ès sciences
originaire de Bienne (BE)

acceptée sur proposition du jury:

Prof. K. Arbenz, président
Prof. G. Coray, rapporteur
Prof. J.-C. Derniame, corapporteur
Prof. A. Strohmeier, corapporteur

Lausanne EPFL
1983

A Isabelle

Remerciements

Je remercie le Professeur Giovanni Coray de m'avoir guidé tout au long de cette thèse et d'avoir accepté d'utiliser des versions expérimentales de l'éditeur dans son enseignement. Ce fut un grand encouragement pour la poursuite de ce travail.

Je remercie le Professeur Jean-Claude Derniame de s'être intéressé à mon travail lorsqu'il a donné des cours à l'EPFL et d'avoir accepté d'être membre du jury.

Je remercie les Professeurs K.Arbenz et A.Strohmeier d'avoir accepté de présider le jury, respectivement d'en être membre.

Je remercie mes collègues pour leur collaboration à divers titres dans la réalisation de ce travail:

Gérard Dalang a introduit les mouvements lexicaux, Jacques Ambühl a assuré le transport de l'éditeur sur RSX, François Voelkle m'a secondé durant un nombre incalculable d'heures dans tous les méandres de ce travail. Il a également contribué à beaucoup d'idées. Rolf Ingold a contribué avec son travail de diplôme à la partie théorique de ce travail. Il a su mettre le vernis mathématique à ce qui n'était qu'intuition chez moi. Il a également écrit le programme VERIFY qui vérifie que la grammaire remplisse les bonnes conditions. Roy Lanek a réalisé le préprocesseur pour VERIFY et a singulièrement amélioré PARSE. Lorenzo Crivelli a programmé la nouvelle commande X et André Dousse a assuré le transport sur VAX/VMS.

Je remercie mes collègues les plus proches Marcel Berthoud, Jean-Pierre Delèze, Béat Hirsbrunner, Sacha Martynov, Anne Neyrinck, André Schiper et Roland Simon qui ont accepté d'utiliser des versions successives et provisoires de cet éditeur. Leurs critiques m'ont été précieuses.

Table des matières.

Résumé

Cette thèse se présente sous la forme d'un produit logiciel qui permet l'édition interactive de programmes ainsi que leur vérification syntaxique.

Dans le premier chapitre nous donnons un aperçu des motivations qui nous ont conduit à développer un tel produit. Ces raisons sont essentiellement à chercher dans la crise du logiciel.

Dans le deuxième chapitre nous présentons notre choix des fonctions de l'éditeur et dans le troisième chapitre nous présentons certains problèmes rencontrés lors de l'implantation et du transport de l'éditeur. Le chapitre quatre est consacré à l'algorithme d'analyse syntaxique utilisé par l'éditeur, à sa justification théorique et à la présentation succinte des conditions imposées à la rédaction de grammaires. Le cinquième chapitre enfin nous donne une évaluation du produit sous différents aspects.

1 EDIS: UN EDITEUR DE SYMBOLES AVEC VERIFICATION SYNTAXIQUE.

1.0 Motivation

Dans la pratique actuelle de la programmation (1983) une partie importante du temps du programmeur est passée devant un terminal d'ordinateur et plus du 80% de ce temps est passé en interaction avec un programme appelé l'éditeur.

Les compilateurs ont bientôt vingt ans et les premières reflexions sur les systèmes d'exploitation ont été élaborées il y a déjà quinze ans. Il n'en va pas de même des éditeurs. Ces derniers ne purent exister que grâce à un matériel récent.

Dès lors on ne s'étonnera pas de se trouver face à une prolifération de produits et un manque de concepts qui permettraient de classer et d'unifier les vues dans ce domaine.

Après avoir tenté de dégager les aspects les plus saillants des éditeurs actuels, nous proposons ici un produit qui se situe à mi-chemin entre les éditeurs de caractères les plus "traditionnels" et les éditeurs dirigés par la syntaxe.

Le public visé par ce produit "utilitaire" est constitué de rédacteurs de programmes amenés à créer, à corriger et à modifier ces derniers. L'éditeur prend en charge ces opérations tant au niveau des symboles que des caractères.

Par ailleurs la vérification de la syntaxe est prévue pour des entités

allant de l'instruction jusqu'au module. Cette vérification syntaxique est effectuée grâce à un algorithme compact, dirigé par table. Cette paramétrisation offre donc une souplesse accrue au produit puisqu'il permet de vérifier des programmes écrits dans différents langages.

1.1 Eléments historiques sur la programmation et les langages

Les premiers ordinateurs sont entièrement codés à la main, en langage binaire. Dès 1950 apparaissent des langages d'assemblage qui, prenant en charge l'attribution des adresses aux variables du programme, permettent une expression symbolique. C'est là une des premières abstractions dans l'histoire de la programmation.

Les applications visées par les ordinateurs à cette époque sont essentiellement le calcul numérique; d'elles est né FORTRAN, un assembleur très perfectionné permettant l'expression de formules mathématiques de manière presque courante ainsi que l'édition des données scientifiques au cours des entrées-sorties (chiffres) à l'aide de modèles compliqués (les FORMAT). Notons que le travail de traduction effectué par le compilateur FORTRAN n'est alors pas encore considéré comme une application typique de l'ordinateur mais comme une aide marginale.

On éprouve la nécessité d'un langage indépendant de la *machine*, en fait du constructeur, avec lequel on puisse décrire des données et des algorithmes de manière plus structurée.

En 1959 on définit COBOL, premier langage de haut niveau, soutenu par une autorité publique, le Département de la Défense des Etats-Unis. A la même époque ALGOL 60 est conçu par un groupe international d'universitaires. C'est aussi le début de la prise de conscience, chez les programmeurs, de la nécessité d'un langage de programmation comme moyen de communication d'algorithmes. Le langage n'est plus uniquement un moyen pour commander la machine. Le signe tangible en est l'existence de deux niveaux du langage dans la définition d'ALGOL 60: le langage de publication

(*reference-language*) et les langages d'installation (*implementation-language*).

Cette petite différence sera le signe de deux attitudes fondamentalement opposées dans la pratique de la programmation pendant les vingt années qui suivent.

D'un côté, il y a, dans l'esprit du programmeur, identité entre le listage d'un programme, les détails d'implémentation et l'algorithme. A lui seul, le langage des organigrammes n'est pas à même de représenter des algorithmes complexes. Son utilisation induit une attitude de "codeur" (*code-hacker*) qui empêche le détachement, pousse à la recherche de l'économie de temps d'exécution et d'espace mémoire, plus généralement d'économie de ressource physique au détriment de la ressource la plus apte à l'abstraction: le cerveau humain. Cette attitude économiste est anarchique mais elle reste le fait de maints programmeurs individuels. Elle sévit jusque dans nos hautes écoles.

De l'autre côté, la recherche puis l'apparition de nouveaux langages de programmation, simples sans être élémentaires, plus mathématiques, de la famille ALGOL et ses successeurs, permet l'expression et la recherche d'algorithmes compliqués de manière synthétique. Qu'on pense à un compilateur, à l'analyseur d'un langage de commande d'un système d'exploitation, à un algorithme heuristique ou encore à un contrôleur de périphérique, programmes écrits dans un langage de haut niveau: leur expression est maîtrisable par un étudiant. Plus tard on verra des langages comme MODULA-2 et enfin l'espoir controversé des années 80: ADA.

L'apparition de nouveaux problèmes à résoudre comme le contrôle de processus ou la maîtrise de la complexité de grands systèmes va engendrer chez les uns la maîtrise toujours plus grande du même langage traditionnel, de la même machine, va affermir la conviction qu'un langage souple et sans contraintes - manipulé avec bon sens et discipline - permet d'affronter tous les problèmes. Ne sont-ils pas venus à bout de tant de *bugs*, n'ont-ils pas tout appris sur le tas. Pourquoi certains font-ils une telle croisade (lisez guerre de langues): "mettons des machines BASIC à dispostion des jeunes et

quand ils seront plus érudits ils sauront déchiffrer le nouveau sanscrit: APL...".

Chez les autres on assistera à l'invention de nouveaux outils ou concepts tels que les notions de ***tâche*** ou de ***module*** dans les langages de programmation. L'effort sur le plan linguistique est considérable et loin d'être terminé: qu'on songe à l'effort entrepris dans le cadre du langage ADA. En plus des aspects actuellement essentiels dans un langage de programmation on cherche à insuffler une ***nouvelle philosophie de la programmation*** grâce à la notion de ***package*** notamment.

Ces deux mondes vont s'ignorer et se mépriser pendant longtemps. D'un côté les "linguistes" ne cessent de proposer des nouveaux langages avant d'avoir achevé l'écriture des anciens compilateurs. Langages qu'ils assortissent de leçons de morale qui ont pour tête de chapitre: ***programmation structurée***, ***décomposition modulaire par raffinements successifs***. De l'autre côté on trouve des gens dont le sérieux est ***attesté par la production de grandes quantités de résultats*** sur les imprimantes de nos ordinateurs.

Ces éléments résumés ici, constituent en fait la trame de fond sur laquelle viennent se dessiner la plupart des recherches actuelles en informatique dans le domaine des outils d'aide à la programmation. Ces domaines recouvrent notamment: la définition de nouveaux langages de programmation, la génération automatique de programmes, les preuves de programmes, les outils intégrés de production de logiciel, la théorie du génie logiciel.

Par rapport à de nombreuses tentatives dans ces domaines notre position reste très sceptique, non seulement déçue en regard des maigres résultats obtenus jusqu'ici, mais aussi réservée par rapport aux chances d'aboutir dans la plupart des démarches.

Dans de nombreuses publications qui touchent aux outils d'aide à la programmation, l'auteur présente son produit comme ***une base d'un ensemble d'outils de programmation***. Lui-même ne s'est pas chargé de l'introduire dans l'hypothétique système intégré de production de logiciel, mais, plus modeste, il nous le propose

comme une première pierre. Or il n'existe pas de théorie générale des outils de production de logiciel; on finit simplement par croire à leur existence, tant on trouve d'auteurs, qui ont ***apporté une pierre à l'édifice***. Nous savons actuellement construire des compilateurs, des éditeurs, (même syntaxiques), des metteurs au point, mais chacun de ces produits a nécessité un développement particulier. Ils sont conjointement utilisés avec intelligence et bonne volonté par des programmeurs, mais il est bien regrettable qu'ils ne soient pas intégrés: leur intégration est précisément un problème où le changement d'échelle dans la complexité demanderait de nouvelles conceptions.

Les ordinateurs sont des outils merveilleux, mais infiniment compliqués à commander, les machines actuelles sont certes beaucoup plus rapides, les mémoires beaucoup plus grandes que celles des années passées, mais les programmes ne sont pas beaucoup plus intelligents, au sens où on l'entend dans les comportements d'un homme. Pour nous, l'intelligence est cette faculté que possèdent les humains de s'adapter à de nouvelles situations en utilisant l'analogie avec des situations vécues antérieurement. L'intelligence n'est pas seulement l'aptitude à la manipulation symbolique. C'est pourtant là qu'excellent les ordinateurs.

1.2 Situation vers la fin des années 60: la crise du logiciel

La crise du logiciel peut être caractérisée comme un malaise que ressent le créateur en informatique quand il veut réaliser les produits

- dans les délais demandés ou prévus,
- avec un coût prévu,
- avec les caractéristiques précises du cahier des charges,
- qui soient fiables,

- qui puissent subir une évolution,

- qui soient portables.

Pour les clients il devient impossible d'obtenir dans des délais précisés à l'avance, un produit informatique déterminé. Le phénomène est d'autant plus agaçant que le prix du matériel diminue alors que le prix du logiciel ne cesse d'augmenter. Il faut de plus en plus de temps pour faire aboutir un projet informatique de qualité. Les exigences que l'utilisateur a en 1980 envers un produit informatique sont beaucoup plus grandes qu'en 1950. Une bonne part du coût du logiciel est imputable à ce que l'on nomme pudiquement la ***maintenance***, laquelle est en fait nécessitée essentiellement par des omissions commises soit au moment de la définition du produit logiciel, soit au cours de sa réalisation. C'est de cette situation, et pour y faire face, qu'est née la nouvelle branche technologique: le ***génie logiciel***. Son but, c'est la maîtrise industrielle du complexe et de l'abstrait, en vue d'une production massive de produits logiciels de qualité à coût minimal. Ses moyens: des méthodes industrielles largement inspirées de celles pratiquées dans d'autres secteurs, une technologie très spécifique, et une automatisation poussée de la ***chaîne de production***.

Nous affirmons pour notre part que la crise du logiciel provient essentiellement de deux phénomènes:

1. Un programme est interprété par une machine et non par des êtres humains.

2. Il n'existe pas de méthodologie adéquate pour la programmation des systèmes des années 80.

Nous allons développer ces deux points.

1. Un programme est un document qui a des spécificités qui le rendent différent d'autres produits de l'ingénierie: il ressemble bien à des instructions données à un être intelligent, mais il doit être précisé dans ses moindres détails puisqu'il n'est pas ***interprété*** (au sens humain) mais ***exécuté***

(par une machine, à la lettre). Pour instruire avec précision un ordinateur il faut donc des formalismes comme les langages de programmation. Le niveau d'abstraction du langage de programmation favorise l'expressivité mais ne change rien à l'affaire: il est certes plus facile de programmer en PASCAL qu'en assembleur mais la même rigueur y est exigée. L'ambiguïté qui est permise dans la communication humaine est sévèrement proscrite en programmation. Cependant la plus grande facilité qu'offrent les langages de haut niveau par rapport à leurs prédécesseurs a engendré chez beaucoup un vain espoir de l'avénement d'un langage avec lequel il ne serait plus pénible de programmer, avec lequel il serait aussi aisé de créer qu'avec une langue naturelle. C'est oublier toute l'ambiguité et les sous-entendus permis dans une telle langue.

Un programme, qu'on pourrait comparer à un plan d'architecte ou de bâtisseur de ponts, est très facile à modifier. Sa ***réalisation***, son exécution comme on dit, peut se dérouler plusieurs fois. Par contre, lorsqu'on a achevé le plan d'une maison, on la construit. On ne peut se permettre de recommencer plusieurs fois la construction en modifiant le plan jusqu'à ce qu'on soit satisfait. Si c'est le cas, toutes les maisons construites devront être habitées. Un programme, par contre, se modifie très facilement. Son concepteur ne livrera que la dernière version au client. La volatilité, la légèreté du matériau d'un programme en fait un objet presqu'aussi léger que le raisonnement.

Comme tout formalisme, le formalisme d'un langage de programmation sert à la communication. Il sert au programmeur pour communiquer ses instructions à l'ordinateur, mais il est trop rarement perçu comme un moyen de communication entre êtres humains. Il n'a pas été concu pour cela mais c'est précisemment la communication qui est le plus important dans ce formalisme.

Prenons un exemple dans un domaine familier: On imaginerait

difficilement un mathématicien affirmer qu'il a trouvé un théorème et dont personne ne comprendrait la démonstration: aussi ambitieuse soit-elle, on exigerait de l'auteur qu'il révèle ses trucs. Il en va autrement dans la programmation: on dit qu'un programme ***marche*** si on ne connaît pas trop de cas où son comportement est inattendu. C'est un peu comme avec la sorcellerie et la médecine.

2. L'imagination en informatique joue un rôle important tant chez le réalisateur que chez le client. Le saut est sans cesse franchi entre ce qui est faisable en principe (le théoricien dirait calculable) et ce qui est faisable en réalité (avec des contraintes de temps et de moyens), ceci notamment en raison de la volatilité du programme. Le programmeur n'hésite jamais à penser à des extensions, des améliorations, des modifications d'un programme : il ne lui en coûtera "que" la conception. Aussitôt qu'il aura imaginé une amélioration, il lui suffira d'amener quelques petites modifications. Pourtant les effets d'échelle sont rarement pris en compte; en effet la complexité d'un programme n'augmente pas linéairement mais bien exponentiellement. Les programmes actuels sont beaucoup plus complexes que ceux des années 50, mais ils sont réalisés dans des formalismes relativement semblables. L'intelligence des systèmes ne s'est guère accrue (ce sont toujours des automates), mais en trente ans l'ambition des réalisateurs est devenue plus grande.

Si le premier compilateur PASCAL est l'oeuvre, à peu de choses près, de deux individus (Wirth et Ammann), c'est que le problème de la compilation est bien connu, bien dominé et qu'une approche méthodique peut encore être le fait d'un individu maîtrisant bien une technique. Le langage est petit et des compilateurs pour ALGOL 60 existent depuis bien des années. On connaît donc les problèmes et l'idée originale réside dans l'écriture d'un autocompilateur. Plus tard, ce compilateur va devenir populaire à cause de l'impact du langage PASCAL: les concepteurs n'avaient pourtant pas cette

ambition!

En conclusion on peut dire que les méthodes actuellement proposées pour la programmation conviendraient (enfin) à la production du logiciel des années 50. Mais existe-t-il des méthodes suffisamment puissantes pour les projets d'envergure actuelle?

Pour notre part nous sommes convaincu que la révolution méthodologique est encore à faire en génie logiciel. Il lui faut pour cela un support linguistique, un environnement de programmation totalement adapté à son langage, et enfin une culture unilatérale propre au langage. C'est ce qu'on voit poindre avec MODULA-2 et surtout avec ADA. [BOOCH 83]

1.3 Solutions proposées pour faire face à la crise

Nous présenterons ici plusieurs approches qui ont été tentées pour faire face aux difficultés inattendues qui ont été rencontrées dans la programmation de produits informatiques.

1.3.1 Méthodes tirées de la gestion et de la conduite de projets informatiques

On tend à augmenter la part de la *conception* relativement à l'effort de *réalisation*. Les équipes de développement sont organisées selon divers schémas; notamment:

1. Le *chief programmer team*, [BOEM 78]. Elle consiste à utiliser un *chef programmmeur* et un *bibliothécaire*. Ce dernier gère les bibliothèques de textes et la documentation. C'est lui qui effectue les modifications des modules. Cela a beaucoup

d'influence sur l'attitude du programmeur: ce dernier demande des modifications avec plus de prudence et perd un peu la tendance générale des programmeurs à vouloir tout faire eux-mêmes. La documentation restera plus longtemps à jour du fait de passer obligatoirement par une même personne. Le rôle du chef programmeur est d'être le directeur technique du projet. C'est d'abord un excellent programmeur, en général plus créatif et plus productif que les autres. Il écrit les interfaces de tous les modules principaux. Il dirige et conseille les autres programmeurs. Une telle équipe est définie pour un projet.

2. [BROOKS 75] propose ***une équipe de spécialistes*** pour organiser la cellule de programmation. Chaque membre a une affectation liée à sa spécialisation et qu'il garde jusqu'au bout. Le chef écrit tout le code et la documentation. Un autre est le bibliothécaire, un autre a charge des outils de programmation ou de test. Un autre écrit les tests et modules nécessaires à l'évaluation et à la simulation.

3. [WEINBERG 71] propose ***une équipe démocratique***. Elle n'a pas de ***leader permanent***. A chaque phase du développement, celui qui est le plus compétent pour cette phase devient le leader. Cette forme de travail dépend beaucoup de la cohésion de l'équipe. L'équipe n'est pas dissoute à la fin du projet.

1.3.2 La Mathématisation

L'idée consiste à bénéficier de tout l'outillage mathématico-logique pour faire des ***assertions*** au sujet des éléments de programmes ou objets dans les programmes. Ces assertions sont des ***prédicats*** distribués dans le programme et on cherche à faire (à la main ou automatiquement) des ***preuves de programmes***. L'interprétation du langage de programmation est axiomatisée. Il existe alors non seulement un système formel pour la syntaxe du langage de programmation, mais aussi un langage d'assertions pour la

sémantique des instructions du langage. Les ***propriétés*** à l'entrée des modules sont utilisées comme ***hypothèses*** pour déduire les assertions valables à la sortie des modules, par l'intermédiaire de règles de déduction correspondant à l'axiomatique du langage de programmation. La terminaison des boucles doit également être démontrée.

Le reproche que l'on fait habituellement à cette approche est que le texte des assertions est aussi volumineux que le texte du programme, qu'il doit être donné dans un formalisme aussi strict qu'un programme et que par conséquent les occasions de faire des erreurs (de forme et de fond) sont aussi grandes dans les assertions que dans le programme. Il est pour le moins amusant d'imaginer la réaction chez le programmeur à qui on propose une telle méthode: mis à part les qualités qui sont exigées d'un programmeur, il devrait être un bon logicien! Rappellons ici qu'un langage de programmation est un système formel et exige une rigueur bien plus méticuleuse que le langage mathématique, langage de communication entre êtres humains. Si une démonstration en mathématiques ne vise qu'à convaincre le lecteur et constitue donc une argumentation, un programme au contraire doit être en plus ***exécuté à l'aveugle*** par un automate.

A notre sens ce genre d'approche relève d'une vision hilbertienne de la technique. Elle poursuit toujours la même illusion: celle de pouvoir ***automatiser les processus de création***, grâce à une mathématique et une logique adéquate. Elle croit pouvoir ***calculer*** les programmes à partir de peu: le cahier des charges. Elle espère pouvoir ***prouver*** (automatiquement) l'adéquation d'un programme à ses spécifications. Elle révèle encore un fait plus grave : le peu d'estime qu'ont les scientifiques de leur créativité. Il faut que la création d'objets technico-scientifiques soit bien méprisable ou que ces objets soient bien dérisoires pour qu'ils cherchent à ***automatiser*** cette activité ! En fait ce qui est dérisoire est la qualité et la quantité de logiciel produit par ces méthodes.

Plus prometteuses semblent les approches qui partent des ***post-conditions*** du module pour ***calculer*** le programme. C'est

l'espoir de [DIJKSTRA 76] et [LIVERCY 78] de pouvoir ***déduire*** des programmes à partir des propriétés qu'ils sont censés avoir. Mais on est loin du compte et les recherches promettent d'être longues encore.

1.3.3 Les outils de production de logiciel

Nous allons présenter un certain nombre d'outils d'aide à la production de logiciel qui sont considérés comme indispensables actuellement.

1. Les ***systèmes de compilation séparée*** font la vérification des liens au niveau des sources et provoquent une recompilation automatique, qui peut être différée au moment de l'édition de liens. Ces systèmes sont chargés de gérer une bibliothèque de textes sources et d'objets des unités déjà compilés ainsi que les informations concernant les relations entre ces unités. Le système de compilation séparée est responsable de vérifier que les contextes de compilation sont complets (tout identificateur utilisé est défini dans l'unité ou dans son contexte) et qu'il n'y a pas d'ambiguïté. Un tel outil fait partie du cahier des charges du projet ADA.

2. Les ***éditeurs de textes*** ou de programme, éditeurs syntaxiques ou éditeurs dans un langage de spécification. Nous reviendrons plus loin en détail sur ce sujet (voir section 1.3.5).

3. Les ***paragrapheurs ou formatteurs***. Ils permettent de mettre en page selon un format uniforme des sources sans modifier la fonctionnalité des programmes qui leur sont soumis. Ils peuvent en partie suppléer la faiblesse du langage de programmation en rajoutant des commentaires par exemple en fin de procédure ou à la fin des structures de contrôle, comme en PASCAL.

4. Les ***systèmes de fichiers*** de programmes et les dictionnaires de données regroupent toutes les données utiles à un programme ou à un ensemble de programmes de la même

application. Ils peuvent regrouper des fichiers de nature diverses tels que sources, binaires, éléments du cahier des charges, dossier d'analyse, de programmation, mode d'emploi, journal des interventions et des décisions au cours du développement du projet.

5. Les *metteurs au point*, *videurs* et *traces* sont des programmes qui permettent la surveillance de l'exécution ou l'analyse de l'état des variables après un arrêt fatal du programme. Ils agissent de préférence au niveau symbolique du source.

6. Les outils de mesure du temps d'exécution et les *profileurs de programmes* permettent de collecter des informations statistiques en vue d'optimisations.

7. L'*indexeur* permet d'obtenir la liste des identificateurs d'un programme avec le numéro de la ligne de déclaration, le genre de l'objet, les numéros des lignes de références avec le genre de référence. Etant donné le volume considérable d'une telle table de références, il convient souvent d'avoir un accès interactif pour fouiller sélectivement une telle table.

1.3.4 L'exhortation à modifier les pratiques des programmeurs: les éléments linguistiques et le style de programmation

1.3.4.1 L'abstraction

Un courant de pensée existe en informatique qui, loin de chercher à trop instrumentaliser des solutions au problème de la crise du logiciel, propose une vieille technique intellectuelle: l'abstraction. Outil classique en ingénierie ou plus généralement pour approcher les problèmes complexes. Un problème est dit *complexe* lorsqu'il est composé de beaucoup d'éléments fortement liés et dépendants et dont les relations internes n'apparaissent pas clairement a priori. L'abstraction consiste alors à inventer un concept, à donner un nom.

En mathématiques ce ***nom*** est censé évoquer l'ensemble des propriétés des objets tandis qu'en informatique ce nom est censé évoquer l'ensemble des actions (pour les noms de procédures) ou l'ensemble des informations possibles (pour les noms de type et de variable). L'abstraction permet de séparer conceptuellement le défini de la définition. Ainsi on peut focaliser son attention tantôt sur les ***propriétés externes*** (dont on oublie les détails ou qu'on ne connaît même pas), tantôt sur la ***réalisation concrète*** de l'objet abstrait.

Chez le mathématicien apparaît un ***dédoublement de la personnalité*** correspondant aux attitudes différentes qu'il prend dans le cours de son activité créatrice. Il peut même s'agir de deux personnes différentes.

Cette pratique est vieille en mathématiques. Le mathématicien n'est pas seulement un manipulateur de vocabulaire inventé pas ses pairs: il devient vite l'inventeur de nouveaux concepts, notions généralisant ou analysant des propriétés connues. Il donne des définitions, des lemmes. Ce faisant il peut aller plus vite, il peut ***oublier*** temporairement le contenu précis de ses définitions. Il existe simplement un contrat tacite avec le lecteur: c'est qu'il puisse substituer en tout temps le défini par la définition.

En programmation une des premières abstractions fut l'existence de noms de variables, abstraction de la mémoire et de noms de sous-routines, abstraction d'algorithmes. Un appel à une sous-routine était censé être équivalent à la substitution textuelle de la sous-routine moyennant changement des noms de variables. En réalité l'utilisateur d'une abstraction se fait toujours un modèle de sa réalisation. Une étape importante de l'informatique va être la généralisation de l'abstraction aux données par l'introduction de la notion de ***type de donnée*** dans certains langages et plus tard la notion de ***type abstrait*** et de ***module***.

Une des difficultés pour la mise en oeuvre de l'abstraction en informatique provient de l'existence simultanée dans un langage de programmation ou dans l'esprit du programmeur des deux niveaux : le défini et la définition. Ce fait, souvent négligé par les

promoteurs de l'abstraction, doit retenir notre attention. Tant qu'on ne peut garantir l'*opacité* de l'objet abstrait, on n'est pas sûr d'avoir gagné la partie. Prenons un exemple: la définition d'une procédure est dans la plupart des cas immédiatement suivie de sa réalisation (son corps). Le programmeur, loin de se faire un modèle (abstrait) aura tendance à se référer au code et à penser en termes d'implantation. Par contre si la définition (l'abstraction) est bien séparée de la réalisation, on peut dire que l'effet d'abstraction est réalisé. Cette séparation était souvent réalisée en FORTRAN à cause de la possibilité de la compilation séparée. Elle a disparu dans les langages "expérimentaux" comme PASCAL et il n'est pas étonnant qu'elle réapparaisse en force avec la notion de *module* en MODULA-2 et de *package* en ADA.

La conception descendante généralement préconisée, qui consiste à concevoir des couches de machines abstraites, va pouvoir jouer. Nous soutenons, pour notre part, que cet effet est pleinement obtenu seulement si les éléments d'abstraction sont supportés par le langage.

1.3.4.2 Le style

Un dernier point concerne le style de la programmation. On inclut ici aussi bien *la calligraphie du texte* que l'attitude pragmatique, caractéristique de l'auteur. Comme dans toute pratique linguistique la liberté combinatoire offre un nombre important de possibilités pour réaliser la même chose. Il n'existe pas de solution unique pour résoudre un problème. Non seulement il existe plusieurs algorithmes différents, mais un même algorithme peut être exprimé de manière variée. Même un programme donné peut s'écrire de différentes manières. On observe les mêmes phénomènes dans une langue naturelle. C'est là qu'interviennent les éléments de style.

Dans les démarches visant à promouvoir la bonne programmation, le style joue un rôle important. Dans la mesure où le texte d'un programme est un véhicule de communication entre divers

programmeurs et est un support concret au processus de dévelopement et de rédaction du programme, l'expression va avoir un certain nombre de caractéristiques typiques des langues naturelles. Le style en programmation relève d'une catégorie esthétique et on va donc juger un programme également en termes esthétiques (c'est aussi vrai en Mathématiques). Cette pratique plus ou moins discrète doit être reconnue ouvertement à notre avis. On trouve encore fort peu d'ouvrages sur le sujet et il faut mentionner ici les livres de [LEDGARD 79] et de [SCHNEIDER 81].

D'habitude les scientifiques n'aiment pas s'aventurer sur un terrain aussi *subjectif*. Pourtant le style de la programmation, et plus généralement le choix d'un langage de programmation, détermine de manière frappante les habitudes et les outils de pensée du programmeur. Voici en résumé un certain nombre de points de style:

1. La clarté du texte par opposition à la malice ou à la virtuosité dans la programmation.

2. La simplicité des algorithmes plutôt que la recherche de l'économie des ressources (temps ou mémoire).

3. Le choix des identificateurs (des noms explicites plutôt qu'abrégés).

4. L'insertion de commentaires permettant de documenter les intentions du programmeur.

5. L'indentation des structures de contrôle, le format du programme.

6. L'utilisation exclusive d'instructions de contrôle structurées: boîtes à une entrée et une sortie

Enfin trois points qui relèvent plus de la pragmatique mais que nous incluons ici:

7. Le comportement robuste du programme vis-à-vis de

l'utilisateur (comportement défensif).

8. Dans les procédures

   - l'utilisation de paramètres protégés (valeurs),

   - l'accès discipliné à peu de variables globales,

   - l'interdiction des *effets de bord*,

   - l'utilisation de variables temporaires locales aux procédures.

9. Le souci de la portabilité

   - par une utilisation de la version standard du langage,

   - en évitant des constantes dépendantes de la machine,

   - en localisant et en identifiant les partie dépendantes de la machine.

En conclusion nous soutenons que le style est une discipline essentielle à la programmation. Mais comme le style en littérature, il est difficile à enseigner, personnel, subjectif. Pourtant son importance est le signe que la programmation est un art [KNUTH 68] et non une science.

1.3.5 Les éditeurs de haut niveau

Parmi les outils d'aide à la programmation il en est sur lesquels nous voulons revenir car notre démarche y est liée, même si nos choix sont différents: il s'agit des *éditeurs de haut niveau*.

Nous aborderons ici les éditeurs généraux de structure et les éditeurs dirigés par la syntaxe.

1.3.5.1 Les éditeurs généraux de structure

Etant donné que la plupart des applications visées possèdent une structure intrinsèque (les documents textuels sont découpés en chapitres, sections, paragraphes, etc) la philosophie des éditeurs de structure est d'exploiter l'ordre ***naturel*** pour diriger l'édition. La représentation la plus courante est une hiérarchie d'arborescence. Les opérations les plus usuelles sur ces structures sont alors le parcours de l'arborescence, l'insertion d'un nouveau site (noeud intermédiaire), l'éclatement, respectivement l'étiolement, la permutation de noeuds, etc. Une caractéristique des éditeurs de structure est la facilité avec laquelle on peut avoir des représentations différentes du programme. Etant donné que la matière à éditer (le texte) est représenté de manière structurée il est possible à l'éditeur de ne montrer que le squelette de l'objet, avec plus ou moins de finesse. Non seulement il sera possible de naviguer dans l'arborescence de la structure mais encore il sera possible de "prendre distance", de voir l'objet avec plus ou moins de détails. On peut considérer l'éditeur comme muni d'un zoom et la régularité de l'effet est alors d'autant meilleure que le découpage aura été fait avec régularité...

1.3.5.2 Les éditeurs dirigés par la syntaxe

Il s'agit d'un cas particulier des éditeurs de structure, la structure étant figurée ici par la syntaxe de l'objet, en général du langage de programmation. L'objet à éditer reste un programme ***abstrait*** c.à.d. une arborescence syntaxique. Un paragrapheur permet de voir le texte habituel à l'écran. On appelle cela un ***décompilateur***. L'idée ici consiste à ne pouvoir éditer que des programmes ***syntaxiquement corrects***. L'éditeur offre la possibilité d'obtenir des maquettes syntaxiques dans lesquelles les ***trous*** peuvent être remplis par de nouvelles constructions, maquettes syntaxiques ou instructions (terminaux du langage).

Dans le cas de MENTOR, [DONZEAU 75], le système est muni d'un langage de manipulation d'arbres (MENTOL). L'objetif de MENTOR est de pouvoir manipuler le programme pour lui faire subir des transformations (pour la portabilité de produits cf. [DONZEAU 81])

ou le calcul d'assertions au sujet de programmes (annoncé comme une des motivations du système MENTOR, mais reléguée aux calendes ... de l'intelligence artificielle).

On peut définir des procédures MENTOL qui sont exécutées sous MENTOR. Chaque utilisateur est donc muni d'une bibliothèque standard ou personnelle de procédures MENTOR qui peuvent constituer des commandes de plus haut niveau que le langage de base de MENTOR.

Le système MENTOR existe avec différents environnements et pour différents langages. L'environnement MENTOR-PASCAL [MELESE 81] est un ensemble de procédures MENTOL qui spécifient comment doit être décompilé un programme PASCAL, les liens qui existent entre la syntaxe abstraite (en termes d'arbre) et la syntaxe concrète (celle du langage). Toutes ces spécifications sont introduites à l'aide du langage METAL [KAHN 83] dans un environnement METAL-MENTOR.

Les auteurs affirment qu'il faut environ une semaine pour devenir manipulateur d'arbre et qu'une fois passé ce cap, les utilisateurs ne reviennent pas volontiers en arrière à un éditeur habituel.

Pour les utilisateurs plus familiers avec le formalisme des symboles de Nassi-Schneidermann [NASSI 73], la manipulation se fera dans le système de support à la programmation d'IBM [FREI 78].

Le cas extrême d'éditeur dirigé par la syntaxe porte le joli nom de ***synthétiseur de programmes*** [TEITELBAUM 80, 81]. Pour commencer une session l'utilisateur frappe la touche ***retour de chariot*** et obtient la maquette suivante:

```
/* commentaire */
nom : PROCEDURE OPTIONS(MAIN);
      declarations
      instructions
```

END nom;

L'utilisateur peut positionner le curseur sur le mot **commentaire** et introduire le texte du commentaire. Ensuite il positionne son curseur sur le non-terminal **declaration**. L'utilisateur peut frapper *.fx* pour une variable fixe, etc.

.fx donne

DECLARE (liste-de-variables) FIXED [attributs];

Le curseur se place sur **liste-de-variables** et le nom des variables doit être introduit. Au niveau des instructions l'utilisateur peut choisir parmi treize maquettes possibles. L'utilisateur continue ce jeu jusqu'à remplacer tous les non-terminaux par des terminaux.

Parmi les avantages des éditeurs syntaxiques on peut citer:

[NOTKIN 79]: "L'arbre syntaxique contient toute l'information nécessaire pour engendrer une représentation exécutable. Les commandes de l'éditeur visent à manipuler l'arbre syntaxique. Les problèmes de fautes d'épellation ou de mots-clefs oubliés tombent car c'est l'éditeur qui les insère et non l'utilisateur. L'avantage de cette forme d'édition est que les programmes sont édités en termes de constructions linguistiques et que l'utilisateur ne peut écrire de programmes syntaxiquement incorrects. Les compilations *juste pour corriger les points-virgules* sont supprimées. (...) La productivité est augmentée."

[MORRIS 81]: "Les éditeurs syntaxiques permettent une plus grande productivité que les autres éditeurs. Ils réduisent l'effort de frappe en offrant des abréviations pour les éléments qui apparaissent souvent comme les mots-clefs, se chargent de la mise en page. (...) Le cycle traditionnel de l'édition-compilation peut être réduit à une seule session...".

[MORRIS 81] cite parmi les principaux désavantages les suivants: Ils sont de gros consommateurs de ressources: la vérification

syntaxique consomme de la puissance de calcul et l'arbre syntaxique occupe de la mémoire. Le second désavantage est une conséquence des contraintes syntaxiques précises que les éditeurs imposent au texte. Pour transformer un programme correct P en un autre P', lui aussi correct, la séquence de transformations la plus courte peut faire passer le texte par des états syntaxiquement incorrects. Par exemple, considérons la transformation de l'instruction:

```
WHILE a > b DO a := a-c;
```

en l'instruction:

```
IF a > b THEN a := a-c;
```

Dès qu'un caractère des mots réservés est touché, les règles de syntaxe sont violées. Pourtant il semblerait naturel de relaxer temporairement ces règles à un degré permettant une édition commode sans faire sauter l'intégrité syntaxique.

On peut bien sûr faire la part des choses de ces divers arguments et il pourrait sembler que ce soit essentiellement une question de goût de préférer ou non les éditeurs syntaxiques. Remarquons qu'un certain nombre d'avantages qu'on leur reconnaît (gain de temps à la frappe, diminution des fautes d'orthographe, correction syntaxique) ***ne sont pas des qualités propres aux éditeurs syntaxiques***. Nous proposons dans ce travail un éditeur, qui n'est pas précisément un éditeur syntaxique, mais qui possède la plupart de ces qualités.

Il nous semble toutefois utile de faire un reproche beaucoup plus fondamental, s'adressant au principe même de l'édition syntaxique. Il s'agit de l'accent mis, par ces éditeurs, sur la syntaxe au détriment des vraies structures, celles qui représentent l'intention véritable du programmeur.

Les linguistes font la distinction entre ***structure profonde*** et ***structure de surface*** de la langue naturelle. La syntaxe d'un langage de programmation se situe à notre avis au niveau de la structure de surface. C'est ce qu'on peut observer lorsqu'on est

gêné par des points-virgules oubliés.

A ce titre, toute tentative qui viendrait alléger la tâche du programmeur, que ce soit en rendant la compilation plus tolérante, en introduisant des automatismes dans l'éditeur ou encore en allégeant les langages est la bienvenue. Mais, à notre avis, l'utilisation d'un éditeur ***dirigé par la syntaxe*** constitue une approche insuffisante au problème du génie logiciel. Le problème qui est posé est la maîtrise de la complexité et pour cela il faut apprendre à structurer un problème. Or la structure syntaxique d'un langage de programmation est une structure de surface. Pour le programmeur l'utilisation d'un éditeur qui force l'attention sur cette seule apparence peut aller à l'encontre de la programmation structurée. Par exemple il aura moins de raisons qu'avec un autre produit de préparer son texte sur un manuscrit avant d'aller à la console.

Enfin imaginons un éditeur de prose qui demande interactivement le verbe, le sujet, le complément. Cela paraîtrait surprenant à toute personne autre qu'un maître de grammaire. Peut-être, les éditeurs syntaxiques sont-ils appréciés par les maîtres de syntaxe.

Il reste à rendre justice aux approches syntaxiques dans la mesure où elles constituent un noyau de primitives solides avec lesquelles peuvent être construites des opérations de plus haut niveau. Lorsque ces opérations seront exprimées en termes d'objets et de concepts du programmeur, voire de l'utilisateur d'un système informatique et non en termes d'arborescence, on pourra voir poindre un espoir dans cette direction. Du coté des ressources, l'arrivée des processeurs 32 bits et des ordinateurs personnels pourraient, à l'avenir, redonner de l'actualité à l'édition syntaxique. Elle serait alors plus transparente à l'utilisateur. Ce dernier n'aurait pas conscience d'utiliser un éditeur syntaxique, pas plus qu'un programmeur n'a, en général, à se soucier des instructions de la machine.

1.4 Notre approche

Nous avons été longtemps tenté d'incorporer dans EDIS, la plupart des éléments qu'on peut trouver en 1982 et que nous critiquons: édition *orientée par la syntaxe*, vérification sémantique, gestionnaire de projet, gestionnaire de modules, de versions, édition des liens vérificatrice, etc.: c'était reconnaître le moment important qu'est l'édition dans la production du logiciel. Ce point de passage semble le lieu idéal où le concepteur de système d'aide à la programmation peut imposer sa méthodologie de travail. En effet certaines idées ont été temporairement implantées et EDIS a été utilisé par des classes de programmation.

Voici ce que nous avons observé: toute contrainte qui empêche l'utilisateur d'un éditeur de travailler à sa guise le détourne de l'éditeur. L'utilisateur veut bien accepter certaines options supplémentaires si elles lui facilitent la tâche (p.e des abréviations pour des mots-clefs, des mouvements par symbole), mais en aucun cas il ne tolère d'être empêché de travailler de manière simple. L'utilisateur ne veut pas avoir à se battre avec l'éditeur pour obtenir un service qui lui paraît évident.

De cette confrontation entre l'idéal du concepteur d'outils et les véritables besoins de l'utilisateur, il est résulté un retour à la conception traditionnelle du travail de programmation: toutes les tâches de gestion sont reléguées hors du champ de l'éditeur. On suppose que le programmeur se rend devant l'éditeur avec un manuscrit de programme pour en faire la saisie. L'éditeur va contribuer à simplifier la saisie en offrant un certain nombre de facilités: liens avec l'environnement, listes des fichiers, éditions de symboles du langage. Mais l'outil n'imposera pas la méthode, il se bornera à proposer, aux instants propices, les fonctions supplémentaires qu'il détient. Il permettra ainsi de travailler selon plusieurs modes et de passer de manière continue de l'un à l'autre.

La saisie est orientée mais pas contrôlée par le langage: à chaque touche du clavier est associé un mot-clef ou un identificateur

courant. Un minimum de mise en page est automatisée: des blancs sont insérés avant et après chaque symbole pour le délimiter, si nécessaire, une indentation a lieu en début de ligne. Une fois un élément de programme saisi (procédure, module de définition ou instructions isolées), l'utilisateur peut en demander la vérification syntaxique. L'éditeur analyse le texte et signale à l'utilisateur d'éventuelles erreurs qui pourront être corrigées immédiatement.

La vérification syntaxique, habituellement faite par le compilateur, est prise en charge par l'éditeur sur demande explicite de l'utilisateur.

En conclusion:

Notre éditeur cherche à intégrer, dans un compromis souple, les trois aspects complémentaires d'un texte structuré: un ensemble de symboles muni des relations syntaxiques, une grille bidimensionelle de caractères affichables et une unité dans un système de fichiers traités par les mêmes utilitaires (compilateur, éditeur, gestionnaire de fichiers).

2 DESCRIPTION DE L'EDITEUR DE SYMBOLES

2.1 Matériel et interfaces de l'éditeur avec l'environnement

Dans cette section nous décrirons le matériel et le logiciel pour lesquels l'éditeur a été implanté et le matériel pour lequel on a pu envisager une implantation.

2.1.1 L'ordinateur

L'ordinateur pour lequel l'éditeur a été principalement développé est un micro-ordinateur TERAK, possédant un LSI/11 comme microprocesseur avec une mémoire de 48K octets, 2 unités de disquettes simple densité, d'une capacité chacune de 0.25 Mbytes et un terminal. Le système d'exploitation est le système PASCAL UCSD. Il comprend un gestionnaire de fichier, un compilateur PASCAL qui produit du code à interpréter et un éditeur à écran, pleine page. Notre éditeur de symboles s'est largement inspiré de cet éditeur au niveau des fonctions courantes et de son architecture.

2.1.2 Le système d'exploitation UCSD

Le système d'exploitation UCSD est constitué de deux éléments: l'interprète de P-code ainsi que les contrôleurs de périphériques écrits en assembleur qui simulent une machine virtuelle, d'une

part, le reste du système, écrit en PASCAL UCSD, qui constitue l'analyseur des commandes et le gestionnaire de fichiers, d'autre part. Le langage incorpore les extensions caractéristiques suivantes:

- le type ***string*** (chaîne de caractères de longueur variable),
- des routines de bas niveau de recherche et de déplacement de caractères. Elles ne sont pas essentielles pour le programmeur courant, mais fortement utilisées par le programmeur système,
- une segmentation du code, permettant une utilisation plus rationelle de la mémoire limitée,
- des entrées-sorties mieux définies pour l'interactif.

2.1.3 Le clavier

Le clavier est le seul canal de l'utilisateur vers l'ordinateur. Il s'agit d'un clavier alphanumérique courant de terminal. Les touches permettent la distinction des lettres majuscules et minuscules. Quelques touches supplémentaires produisant des caractères non affichables sont disponibles telles que TAB, ESC, ETX, BS, DEL, RET ainsi que les touches permettant les mouvements du curseur.

Il n'est pas possible depuis l'ordinateur de reconnaître la frappe simultanée de plusieurs touches. Quand c'est le cas, le circuit du clavier s'en charge comme c'est courant dans les claviers actuels avec les touches SHIFT et CTRL. Pour engendrer quelques fonctions supplémentaires il peut donc être nécessaire d'appuyer sur la touche CTRL en même temps que sur une autre touche.

2.1.4 L'écran

L'écran est alphanumérique. Il a 24 lignes de 80 colonnes. La vitesse

de transmission voisine les 9600 bauds. Pour une utilisation agréable de l'éditeur une vitesse minimale de 2400 bauds est conseillée. Un écran de moins de 24 lignes ou de moins de 80 colonnes constitue une sérieuse limitation pour l'utilisation.

L'écran doit posséder un minimum de fonctions dites ***intelligentes***. Toutes ne sont pas nécessaires car l'éditeur est programmé de telle sorte qu'il se débrouille avec un nombre limité d'entre elles. Voici une liste des fonctions souhaitables:

- positionner le curseur à la position supérieure gauche,
- positionner le curseur à une coordonnée quelconque,
- mouvoir le curseur dans les quatre directions d'un caractère sans détruire le caractère qui est affiché,
- effacer tout l'écran,
- effacer l'écran depuis la position courante du curseur,
- effacer la ligne courante,
- effacer la fin de la ligne courante,
- mouvoir le curseur vers le bas avec un ***scroll implicite***.

En revanche on n'a pas utilisé les fonctions plus ***intelligentes*** qui existent sur certains terminaux, comme par exemple:

- insérer un caractère blanc dans la ligne courante,
- détruire la ligne courante avec remontée du texte qui suit,
- insérer une ligne blanche,
- mouvoir le curseur vers le haut avec un ***scroll implicite***.

2.1.5 Les fichiers

Sur le système UCSD chaque disquette peut contenir un maximum de 77 fichiers de texte. Chaque fichier possède des blocs de tête qui conservent des renseignements sur la nature du fichier (comme, par exemple, le langage) et le mode de fonctionnement de certaines commandes de l'éditeur vis-à-vis de ces fichiers. Ces renseignements constituent le ***profil utilisateur*** (détail 2.3.3), ne sont utilisés que par l'éditeur et sont ignorés par les autres utilitaires du système d'exploitation (compilateur, gestionnaire de fichier). L'éditeur doit pouvoir lire un fichier en mémoire, introduire un autre fichier dans son tampon d'édition, écrire le tampon d'édition dans un fichier ou sauvegarder le fichier édité à la place de l'ancien fichier. Il donne en outre la liste des fichiers accessibles sur disquette, ce qui est une facilité fort agréable puisqu'elle évite à l'utilisateur de sortir de l'éditeur pour connaître la liste des fichiers. Le fait que le fichier édité réside entièrement en mémoire en limite la taille (sur TERAK environ 15000 caractères).

2.2 Commandes de l'éditeur du point de vue de l'utilisateur

2.2.1 Ce que voit l'utilisateur à l'écran

L'éditeur de symboles est une éditeur à écran, pleine page. La première ligne de l'écran est une zone dans laquelle apparaît le menu ou les messages de l'éditeur à l'utilisateur. Tout le reste de l'écran est une fenêtre sur la portion gauche du fichier. Les lignes du fichier d'édition peuvent avoir une longueur quelconque mais seuls les 80 premiers caractères apparaissent. On voit le caractère "!" en dernière position de la ligne quand celle-ci déborde de l'écran. La fin de la ligne constitue un caractère. Il est donc possible

de couper une ligne en deux en introduisant un caractère de fin de ligne ou au contraire de fusionner deux lignes en supprimant la fin de ligne.

Le curseur apparaît dans la fenêtre et c'est par rapport à lui que la plupart des commandes sont éxécutées. Le curseur peut être déplacé dans toutes les directions à l'aide des différentes commandes de mouvements. Dès que le curseur menace de sortir de l'écran, une nouvelle portion du fichier est affichée. La longueur d'un mouvement dépend de la nature du fichier et de la commande. Il existe trois sortes de mouvements: les grands (ligne, page), les mouvements par symbole du langage ou par mot en PROSE, et les mouvements par caractère.

2.2.2 Les deux modèles du fichier d'édition

Nous allons maintenant voir que l'éditeur peut être spécialisé et offre un comportement adapté, selon que l'utilisateur fait de l'édition de texte ou de l'édition de programmes.

Le fichier d'édition peut être perçu comme un quart de plan constitué de ***caractères*** dans lequel on peut balader le curseur horizontalement et verticalement. On obtient ces mouvements avec les touches haut ou bas et SP ou BS respectivement.On dira que ceci constitue le modèle textuel du fichier.

Mais on peut aussi considérer le fichier comme constitué de ***symboles*** d'un langage de programmation ou de ***mots*** en PROSE. Les mouvements se font alors de symbole en symbole respectivement de mot en mot. Ces mouvements sont obtenus par les touches gauche et droite. On dira que ceci constitue le modèle symbolique du texte.

2.2.3 L'éditeur en mode PROSE

C'est le mode qui convient pour faire du traitement de texte.

L'utilisateur peut faire de la *saisie au kilomètre* sans se soucier des fins de lignes. Les mots sont tassés et si la place vient à manquer vers la marge droite, le mot est repoussé à la ligne suivante. On obtient ainsi des paragraphes avec une mise en page *en drapeau*. Si on est amené à corriger un paragraphe (suppression ou insertion de mots, correction de l'orthographe) une commande permet la mise en page d'un paragraphe. C'est utile si le texte est destiné à être traité par un programme de mise en page. Sinon on peut demander la justification immédiate du paragraphe: des blancs supplémentaires sont insérés entre les mots afin d'obtenir également une justification à droite.

Pour la saisie de texte français il est également important de pouvoir saisir des lettres accentuées. Comme la plupart des claviers d'ordinateur actuels n'ont pas de lettres accentuées c'est réalisé avec deux caractères, comme sur les machines à écrire: pour introduire un "*é*" on frappe sur la touche */*, puis sur la touche "*e*". L'éditeur reconnaît les deux caractères et introduit le code particulier du "*é*" dans le fichier. Sur TERAK, le jeu de caractères affichables est programmable et on peut représenter 2 x 96 caractères: les caractères accentués ont été préprogrammés. A l'écran, le */* est remplacé par un "*é*".

2.2.4 L'éditeur de programmes

Comme déjà évoqué dans l'introduction, les éditeurs syntaxiques existants présentent plusieurs désavantages: la complexité de l'implantation, une interface contraignante avec l'utilisateur, souvent aussi la pauvreté des fonctions pour le traitement de texte. Des solutions partielles consistant à superposer au texte une arborescence ont l'inconvénient de présenter deux vues séparées d'un programme et rendent l'implantation compliquée. Avec le modèle textuel d'un programme, la conception de l'éditeur en est grandement simplifiée.

Les seuls éléments qui font d'EDIS un éditeur de programmes sont des éléments qui fonctionnent avec un contexte local. Nous allons

les présenter ici:

- Pour chaque ligne de texte, l'éditeur reconnaît les éléments lexicaux en procédant à une analyse lexicale de gauche à droite chaque fois que le curseur se déplace sur une nouvelle ligne.

- La saisie du programme est facilitée par la préprogrammation des touches. A chaque touche de l'alphabet est associé un mot-clef ou un identificateur courant du langage. Comme la plupart du temps les mots-clefs sont entourés d'espaces dans les langages de programmation, ces espaces sont insérés automatiquement quand c'est nécessaire.

- Au début de chaque ligne une règle d'indentation est appliquée pour déterminer la marge gauche. L'incrément d'indentation peut être choisi par l'utilisateur et fait partie de son profil. Il n'est guère possible d'établir des règles d'indentation qui satisfassent tout le monde, c'est pourquoi l'indentation peut être facilement modifiée par les touches de mouvement gauche ou droite dont l'effet est de translater la ligne entière.

2.2.5 La vérification syntaxique

Lorsque l'utilisateur a terminé la saisie d'un programme complet, ou de quelques procédures il lui est possible de vérifier la syntaxe de l'unité choisie. Les éventuelles erreurs lui sont signalées et il peut tout de suite les corriger: le cycle traditionnel ***édition-compilation*** est allégé.

La solution proposée est un compromis heureux, valable pour un petit ordinateur. Faire faire à l'éditeur tout le travail du compilateur est absurde. On a préféré se limiter à une analyse syntaxique dirigée par table sans vérification de compatibilité de type, etc. Nous reviendrons plus en détail sur ces choix dans le chapitre concernant l'implantation. Ce dispositif s'avère utile dans un contexte différent de la programmation, où le document saisi

doit présenter une structure riche.

2.3 Nature des commandes

2.3.1 Comportement de l'éditeur

Il est essentiel que l'utilisateur ait l'impression de contrôler l'éditeur et non l'inverse. La manifestation la plus évidente de cet impératif est la possibilité de détruire l'effet d'une commande donnée à l'éditeur. Un utilisateur serait gêné s'il ne pouvait revenir en arrière lorsqu'il a donné une commande erronée. On a donc veillé à pouvoir presque toujours révoquer la commande en cours. Malgré cela le langage de commande présente un modèle relativement simple.

La syntaxe des commandes est la suivante: un facteur numérique, la commande, le ou les paramètres de la commande suivi de ETX ou ESC. Chaque mouvement est obtenu par la frappe d'une seule touche.

Chaque commande de mouvement peut être préfixée par un facteur numérique de répétition, la touche "/" représentant le facteur illimité. Mais on peut aussi simplement appuyer sur la touche en observant le déplacement du curseur jusqu'à ce qu'il soit à l'endroit désiré. Les mouvements ont un effet immédiat, visible.

En second lieu existe la commande de recherche. Elle permet de rechercher une sous-chaîne connue littéralement ou au contraire de rechercher une séquence approximative de symboles. Dans ce dernier cas, les séparateurs de symboles sont donnés à quelques blancs près. La commande de recherche permet aussi de compter le nombre d'occurrences d'une sous-chaîne, ou d'une séquence de symboles. Les mêmes abréviations qu'en insertion sont autorisées lors de la saisie de la chaîne à rechercher.

Enfin existent les commandes qui peuvent modifier le fichier.

Toutes ces commandes sont invoquées par une touche alphabétique. La zone de dialogue signale alors qu'on est entré dans le mode de la commande. On en sortira par la frappe de la touche ETX (confirmation de la commande), ou ESC (annulation de la commande). Entre-temps toutes les touches ont un sens qui dépend de la commande. Le comportement est assez intuitif et consistant pour l'ensemble des commandes. Nous allons détailler les commandes en les regroupant par catégorie en suivant une session à l'éditeur.

2.3.2 Profil du terminal

Le but du prologue est la paramétrisation de l'éditeur par des caractéristiques du terminal et de l'utilisateur. Le profil du terminal est déterminé à partir d'un fichier décrivant les caractéristiques du terminal. Il s'agit là de rendre l'éditeur le plus indépendant possible des différentes sortes de terminaux envisageables. Le profil de l'utilisateur est également déterminé à partir d'un fichier. Le profil détermine le comportement de certaines commandes et la valeur de certains paramètres (langage par défaut). L'utilisateur choisit le fichier à éditer, respectivement à créer. A ce stade il peut connaître la liste des fichiers accessibles dans son environnement. S'il s'agit d'une création, le profil de l'utilisateur est celui déterminé à partir du fichier de profil. Si la session vise à l'édition d'un fichier existant, le profil de l'utilisateur est celui trouvé dans le fichier édité. A ce stade l'utilisateur se trouve au niveau des commandes de l'éditeur. Il peut consulter ou modifier son profil, effectuer des mouvements, modifier le fichier d'édition, communiquer avec l'environnement pour déposer le fichier d'édition ou introduire un fichier, demander une vérification syntaxique ou sortir de l'éditeur.

2.3.3 Consultation et modification du profil de l'utilisateur

Pour les langages de programmation le profil de l'utilisateur

comporte:

a) le choix du langage, qui détermine l'analyse lexicale, nécessaire pour les mouvements par symbole,

b) l'utilisation de mots-clefs (inhibés, en majuscules, en minuscules),

c) la valeur de l'incrément d'indentation (éventuellement nul),

d) le mode de recherche dans les commandes de recherche et de substitution.

et pour la PROSE:

e) la marge gauche,

f) la marge droite,

g) l'indentation en début de paragraphe,

h) le fonctionnement de la commande de paragraphage (inhibée, justifiée, en drapeau).

Outre ces paramètres modifiables explicitement par l'utilisateur, des renseignements liés au fichier sont donnés tels que:

a) la taille du fichier et la place libre,

b) les dates de création et de mise à jour du fichier,

c) le nom du fichier,

d) la version du fichier,

e) les valeurs actuelles des tampons de recherche et de substitution.

2.3.4 La navigation dans le fichier

Dans un éditeur à écran, les mouvements sont certainement les

commandes les plus utilisées. Il convient donc de définir soigneusement et de manière cohérente le jeu qui est offert à l'utilisateur. Souvent les mouvements interviennent également dans d'autres commandes telles que la destruction ou la copie. Leur définition doit être la même qu'au niveau principal de l'éditeur.

Une bonne solution consisterait à définir les mouvements selon plusieurs dimensions indépendantes, par exemple:

direction: avant / arrière
fonction: mouvement / destruction / copie
grain: caractère / mot / ligne / paragraphe.

La composition de ces diverses dimensions s'obtient alors soit par la frappe simultanée de différentes touches, ce qui suppose un matériel particulier, soit par l'existence de multiples touches qui pourront être attribuées de manière cohérente par l'utilisateur. Avec les terminaux actuels, nous ne pouvions prendre de telles options. Nous nous sommes limités au choix suivant:

- mouvement au début et à la fin du fichier,
- mouvement au début de la page,
- avance et recul d'un écran,
- mouvement au début et à la fin de la ligne,
- avance et recul d'un symbole,
- avance et recul d'un caractère,
- mouvement vertical.

Ces commandes peuvent être préfixées d'un facteur numérique. Pratique pour compter des lignes, c'est très peu utilisé. C'est une solution peu naturelle qui remonte aux éditeurs de lignes: il est souvent tellement plus simple de laisser son doigt sur la touche voulue le temps nécessaire! Une condition est bien sûr l'arrêt instantané du mouvement du curseur dès que le doigt est levé.

2.3.5 La modification du fichier

Dans toutes les opérations de modification du fichier il apparaît essentiel

- de pouvoir observer la modification immédiatement. Elle ne doit en particulier pas se dérouler à l'abri du regard de l'utilisateur,

- de pouvoir défaire le plus simplement possible la commande de modification en ramenant le fichier à l'état antérieur.

Les commandes de modification se regroupent dans les catégories suivantes:

a) destruction,

b) insertion,

c) substitution, modification,

d) copie.

Nous discuterons, à titre d'exemple, les commandes de destruction, de substitution et d'insertion.

2.3.5.1 Destruction

C'est une commande très souvent utilisée, qui demande un certain apprentissage. Elle est aussi le prélude à la commande de copie ou de déplacement de texte. Pour déterminer la portion à détruire, préalablement à l'appel de la commande, on place le curseur à l'une des extrémités de cette position du texte. Après appel de la commande, on déplace le curseur, qui peint l'écran en noir (en fait écrit des *blancs* sur la zone détruite). Une fois le curseur parvenu à l'autre extrémité du texte à détruire on confirme la commande. Le texte effacé cède la place à la suite du texte.

A tout instant il est possible de renoncer à la commande et tout se

rétablit sous les yeux de l'utilisateur. Un repentir, en cas de confirmation involontaire, est encore possible pour rétablir l'ancienne situation à l'aide de la commande de copie. Afin de maintenir autant que possible la cohérence du texte un algorithme d'insertion de blancs décide si deux symboles doivent être séparés ou non par un blanc.

On observera qu'il n'existe pas de commande de transfert ni de commande de copie comprenant les paramètres délimitant les zones concernées. Pour effectuer un transfert ou une copie on doit utiliser consécutivement les commandes de destruction et de copie. Les paramètres des commandes sont plus simples mais c'est inhabituel.

Il existe deux autres commandes de destruction, une pour détruire des grandes portions de texte, une pour la correction très locale de caractères. Les deux commandes ont une possibilité de repentir. Leur emploi est plus rare; en outre la syntaxe est différente.

2.3.5.2 Insertion de texte

L'insertion peut avoir lieu soit en fin de fichier, soit au milieu d'un texte. Dans ce dernier cas le reste de la ligne est affiché à droite de l'écran afin de rappeler à l'utilisateur le contexte. Les symboles sont alors introduits soit par la frappe abrégée par touches pour mots-clefs, soit caractère par caractère. Lors de la frappe abrégée des blancs séparateurs sont insérés automatiquement quand c'est nécessaire. En cours d'insertion il est possible d'effacer les derniers caractères introduits au moyen de la touche *BACKSPACE*.

A l'introduction d'une fin de ligne, le curseur vient se positionner sous le premier caractère de la ligne précédente, à moins que celle-ci ne commence par un certain mot-clef, auquel cas la marge gauche subit une indentation. Il en va ainsi, par exemple, en PASCAL à la suite de **BEGIN, REPEAT, WHILE, IF, ELSE, VAR, TYPE**. Lorsqu'on frappe un mot-clef qui termine une structure, le curseur vient se placer plus à gauche. Il en va ainsi lors de la frappe de **ELSE,**

UNTIL, END. On devine que ce schéma fort simple ne convient pas toujours. Pour cette raison, on peut *modifier* en tout temps l'indentation de la ligne, à l'aide des touches gauche ou droite sans quitter le mode insertion.

Pour la saisie de texte, les problèmes sont différents. L'indentation sur mot-clef n'a plus de sens. C'est au contraire la notion de paragraphe qui devient importante. Un paragraphe est défini comme une portion de texte délimité par une ligne vide aux deux bouts. En ***saisie au kilomètre***, la mise en page se fait ***en drapeau***. On peut observer que le mode Insertion, qui offre peu de possibilités de corrections convient assez mal à un utilisateur faisant beaucoup de fautes de frappe. A chaque faute remarquée il faut terminer l'insertion, revenir en arrière, corriger l'insertion et repartir. L'autre solution consiste à effacer tous les caractères frappés depuis l'erreur.

2.3.5.3 Substitution de texte

Mis à part la saisie de texte, la modification du texte est une activité importante en édition. Les deux moyens les plus courants sont, d'une part, la destruction de texte suivie d'insertion, d'autre part, la modification de ce qu'on voit à l'écran. Cette manière artisanale convient parfaitement à l'utilisateur car il voit immédiatement ce qu'il fait et n'a pas besoin de planifier une modification compliquée. Lorsqu'on doit faire successivement plusieurs modifications analogues cette manière de faire devient cependant fastidieuse. Il convient alors de disposer d'une commande de substitution. Nous avons veillé à simplifier au maximum la syntaxe de cette commande qui est souvent la plus compliquée dans les éditeurs et donc la plus rebutante.

Les deux paramètres de la commande de substitution n'ont pas de délimiteurs, mais un terminateur non affichable (ETX), la frappe par mot-clef est possible, une chaîne vide signifie l'ancienne chaîne.

Dans cette commande, la recherche se fait en ***mode symbole***

(ignorant les séparateurs et identifiant les minuscules et majuscules) ou en *mode littéral*.

2.3.6 Ecriture et lecture d'un fichier sur disque

A tout instant il est possible de déposer le contenu courant du fichier d'édition sur un fichier sans quitter l'éditeur. De manière symétrique il est possible d'insérer le contenu d'un fichier dans le fichier d'édition. Ces deux commandes ont comme paramètre le nom d'un fichier. Lorsqu'on donne ce nom, il est possible de connaître la liste des fichiers éditables.

2.3.7 Vérification syntaxique

Dans un langage de programmation la correction syntaxique des énoncés est exigée. Dans une instruction conditionnelle en PASCAL, par exemple, il faut non seulement que la condition précède les énoncés conditionnels, mais encore il faut la présence du séparateur **THEN**. Pour un débutant ou simplement pour un programmeur peu soigneux, cet aspect apparaît comme fort rébarbatif. Il est la plupart du temps ressenti comme un obstacle inutile qui a été placé là par on ne sait qui. Nous ne voulons pas décrire ici les bonnes raisons qui imposent de placer cette exigence, mais l'utilisateur est souvent conduit à adopter une attitude fataliste qui consiste à accepter le passage réitéré de son programme par le compilateur. Nous avons voulu offrir la possibilité à l'utilisateur de vérifier la syntaxe d'éléments de programme dans l'éditeur. Le grain de l'unité est à choix: il va de l'*instruction*, en passant par la *procédure* jusqu'au *module* ou *programme complet*.

L'utilisateur place le curseur au début de l'unité syntaxique qu'il désire vérifier. Il appelle la commande de vérification syntaxique. Le menu qui apparaît alors est le choix parmi les unités syntaxiques vérifiables. A la première erreur l'utilisateur est averti par

l'affichage d'un message en français. Il peut alors poursuivre l'analyse syntaxique ou effectuer la correction et recommencer son analyse. Dans le cas où aucune erreur de syntaxe n'est découverte, il lui est possible de connaître la liste des identificateurs non déclarés dans l'unité syntaxique. Il s'agit soit des identificateurs externes, soit des identificateurs mal orthographiés ou non déclarés. Cette vérification sémantique quelque peu sommaire a rendu souvent service, si l'on songe que son implantation est fort simple; il est bien sûr hors de question d'introduire toute la vérification sémantique d'un compilateur dans un éditeur.

2.3.8 Sortie de l'éditeur

Lorsque la session à l'éditeur est terminée, que l'utilisateur ait consulté ou modifié un fichier, il demande à sortir de l'éditeur avec mise à jour explicite. Présisons que l'éditeur ne travaille pas sur le fichier mais sur une copie en mémoire. S'il va écrire dans un fichier, généralement en détruisant l'ancienne version, il importe que l'éditeur avertisse de ce qu'il va faire en nommant le fichier qui va être détruit. Après confirmation de la part de l'utilisateur, l'éditeur écrit le fichier d'édition sur le fichier désiré et en donne confirmation. Il est alors possible soit de terminer la session, soit d'en recommencer une autre en restant dans l'éditeur. (Surtout utile pour un petit ordinateur à disques lents pour éviter le chargement).

3 IMPLANTATION DE L'EDITEUR

3.1 Historique d'EDIS et l'incidence de PASCAL UCSD

Dans cette section nous rappellerons brièvement les caractéristiques du système d'exploitation UCSD sur lequel nous avons implanté l'éditeur de symboles EDIS, ainsi que les particularités de cet éditeur qui ont dicté certaines politiques d'implantation. Le système UCSD est un système d'exploitation mono-utilisateur simple pour micro-ordinateur; il apparaît au programmeur comme s'il était entièrement accessible en PASCAL. Le système offre une segmentation nécessaire au fonctionnement du compilateur et de l'analyseur de commandes. Le noyau du système d'exploitation est le programme principal dont tout programme - l'éditeur en particulier - n'est autre qu'une procédure segmentée. Tout programme peut donc avoir accès assez facilement aux tables du système.

L'éditeur de symboles doit:

- offrir un temps de réponse immédiat à la plupart des commandes (donc éviter de charger du code),
- garantir des temps d'accès également rapide à tout le fichier (donc placer le fichier d'édition entièrement en mémoire; il faut impérativement réserver de la place à cet effet),
- reconnaître des symboles pour les mouvements lexicaux,

l'indentation et la vérification: *l'analyseur lexical* est donc une procédure résidente de l'éditeur,

- permettre une vérification syntaxique; la grammaire doit être chargée en mémoire.

Ces points ont déterminé la politique d'implantation que nous décrivons dans la prochaine section.

3.2 Politique d'implantation

L'implantation, sur un micro-ordinateur, d'un éditeur avec beaucoup de fonctions, dont certaines sont complexes, a déterminé deux choix: la segmentation du code de l'éditeur et la représentation linéaire, non structurée, du fichier d'édition.

a) Le segment de code principal comprend:

- les routines de service de très bas niveau,
- les routines d'analyse des commandes et d'affichage,
- l'analyse lexicale.

Un segment est chargé sur la pile et traite:

- l'initialisation de l'éditeur (calcul de la taille du tampon) ainsi que la lecture et l'écriture du fichier,
- l'affichage et la modification du profil de l'utilisateur,
- la vérification syntaxique,
- le reste des commandes de l'éditeur à savoir:
  - la navigation,
  - l'insertion,
  - la destruction,

- la modification, la recherche.

b) Le fichier est représenté en mémoire comme une zone contiguë de caractères affichables séparés par des caractères de fin de ligne. Les espaces sont condensés (le nombre de caractères blancs en début de ligne). Quelques pointeurs servent à identifier la fin actuelle du fichier, la position courante du curseur et le premier caractère qui est affiché à l'écran. Aucune structure n'est superposée en permanence à cette représentation fort simple; c'est au contraire par des calculs incessants qu'on arrive à afficher le contenu modifié de l'écran, ainsi que les mouvements du curseur. Pour permettre un bon temps de réponse certaines routines de bas niveau sont écrites en assembleur (pour l'implantation sur d'autres systèmes qu'UCSD). Ces routines existent comme procédures standard en PASCAL UCSD. Si le processeur est suffisament rapide, la version PASCAL-UCSD suffit. Nous allons les présenter dans la section suivante.

3.3 Primitives assembleur

Pour déplacer de grandes zones de mémoire rapidement il convient de disposer de procédures de déplacement.

- **moveleft, moveright** prennent comme paramètres l'origine, la destination et le nombre de caractères à déplacer.

- **fillchar** est utile pour initialiser une zone complète avec un même caractère.

- Pour la recherche d'un caractère donné en mémoire **scan** prend comme paramètres l'origine, la direction de recherche, la distance maximale de recherche et le caractère recherché.

- Pour la comparaison de chaines, (en identifiant minuscules et majuscules), la procédure **cmpc** prend pour paramètres les deux origines et la longueur.

- Pour l'analyse lexicale, la constitution, la détermination d'un identificateur, et dans le cas de l'analyse syntaxique, l'identification éventuelle d'un mot-clef se fait grâce à la procédure **ids**. Elle a comme paramètres le curseur dans le tampon; elle délivre le curseur à sa nouvelle valeur, l'identificateur, et la valeur du *mot-clef* selon une table qui lui est fournie.

3.4 Architecture de l'éditeur

3.4.1 Tampon d'édition

A l'initialisation de l'éditeur une zone contiguë est allouée sur le tas. Sa taille va dépendre de la version du système. Elle est environ de 15000 octets sur UCSD. Cette zone contiendra le tampon d'édition et le tampon de copie, qui peuvent se recouvrir partiellement. Voici l'organisation de ce tampon:

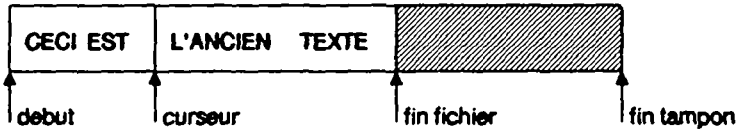

Fig. 341a

En début d'insertion la partie droite est déplacée vers la fin du tampon:

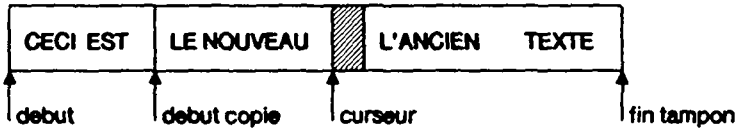

Fig. 341b

En fin d'insertion la partie droite est recollée et la partie insérée devient le tampon de copie.

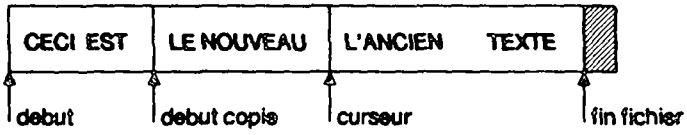

Fig. 341c

Pour la destruction l'organisation est la suivante. Au début on a la même situation qu'en (341a). Les mouvements n'ont aucun effet dans le tampon et c'est seulement l'affichage qui témoigne de l'intention de l'utilisateur. A la fin de la destruction deux situations sont possibles:

a) la destruction est refusée. Le tampon n'ayant pas été modifié, on se contente de réafficher. Par contre le tampon de copie est maintenant défini comme dans la figure (341c),

b) la destruction est acceptée. La zone à détruire est alors d'abord copiée vers la fin de la zone. Puis la fin du fichier est tassé. Le tampon de copie est défini vers la fin de la zone. Voici la situation:

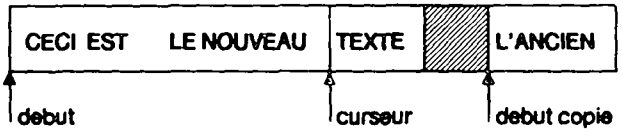

Fig. 341d

Chaque fois que des zones sont retassées un algorithme détermine si un blanc doit être inséré pour maintenir la cohérence des symboles.

3.4.2 Lecture et écriture du fichier

Une fois le nom du fichier déterminé et le fichier ouvert, ce dernier est lu par bloc sur UCSD, par ligne dans les autres systèmes. Une représentation interne lui est donnée, qui correspond à celle sur disque en UCSD. A la fin de chaque ligne il y a un caractère de fin de ligne, au début de chaque ligne les blancs sont représentés de manière compacte. Le fichier subit encore quelques vérifications et modifications. Les fichiers à éditer peuvent provenir de sessions d'édition antérieures ou être des sorties de programmes ou d'autres éditeurs. Leur représentation n'est donc pas forcément la même et il convient de nettoyer le fichier de tout caractère parasite. D'autre part, s'il s'agit d'un texte de programme en PASCAL ou en PORTAL, les commentaires peuvent porter sur plusieurs lignes. Comme cela est incompatible avec la philosophie que nous avons adoptée -qui veut qu'un symbole ne puisse porter sur plusieurs lignes et qu'une fin de ligne est un symbole-, les commentaires portant sur plusieurs lignes sont transformés en commentaires limités à une ligne par l'insertion de fin et début de commentaires. Cette vérification est effectuée à la lecture du fichier, et à l'écriture si une modification a eu lieu pendant la session.

3.4.3 Paramétrisation des commandes de l'éditeur

Il convient de bien distinguer d'une part les commandes à l'éditeur et la manière d'obtenir ces commandes au clavier et d'autre part les fonctions de contrôle de l'écran. La plupart des commandes de l'éditeur sont obtenues par la frappe d'une touche alphabétique. Mais certaines commandes sont obtenues par la frappe de certaines touches spéciales, comme les mouvements du curseur dans le fichier ou la fin d'une commande avec paramètres (ETX, ESC). Cependant, si tous les claviers d'ordinateur sont à peu près standard sur la distribution des touches alphabétiques (elles sont au moins toutes présentes), il n'en va pas de même pour les touches

spéciales. Quand elles existent leur frappe envoie des caractères ou séquences de caractères qui peuvent varier d'un terminal à l'autre. Lorsqu'elles sont absentes il convient alors de décider quelle combinaison de touches va constituer la commande en question. On suppose au moins l'existence de la touche CTRL.

La paramétrisation permet ainsi de reconnaître pour chaque commande de l'éditeur une séquence de un à trois caractères. Cette paramétrisation est courante sur le système UCSD. Elle est prise en charge lors de l'initialisation du système et est disponible dans les tables du système. Pour les autres systèmes on a eu recours à la lecture d'un fichier spécifique, appelé profil du terminal.

3.4.4 Paramétrisation du pilotage de l'écran

Ici le problème est différent. L'utilisateur n'a pas le choix. Le terminal dispose d'un jeu bien défini de fonctions. Il s'agit de décrire lequel et comment chaque fonction a priori prévue dans l'éditeur doit être pilotée. L'éditeur est programmé de telle sorte qu'il se débrouille avec le jeu offert. Un problème est apparu avec l'adressage du curseur. La variété extrême de la séquence à envoyer ne permet pas une paramétrisation a priori et par table. Il faut prévoir les manières de représenter cette fonction et les numéroter. Ce numéro figure alors dans le fichier décrivant le profil de l'écran. Sur le système UCSD, le problème est connu; il est résolu brutalement en configurant le système une fois pour toutes pour un terminal donné.

3.4.5 Algorithme de recherche et de substitution

Lorsqu'on recherche une chaîne, elle est enregistrée dans un tampon. Il en va de même de la chaîne à substituer. Les contenus de ces tampons sont disponibles pour une recherche ou une substitution ultérieure. Ils peuvent être consultés par la commande

de mise à jour du profil de l'utilisateur. Ces tampons restent définis lorqu'on édite plusieurs fichiers consécutivement.

3.4.6 Mots-clefs, Indentation

En mode programme la frappe des touches alphabétiques en majuscule provoque, comme on sait, la saisie d'un mot-clef ou d'un identificateur standard. Ces mots-clefs existent dans une table et c'est aussi cette table qui est utilisée par l'algorithme d'indentation.

3.4.7 Insertion automatique de blancs

L'entité fondamentale que l'utilisateur a en tête avec cet éditeur est le symbole quand il s'agit de programme, respectivement le mot quand il s'agit de texte. Cela lui est suggéré par les mouvements gauche et droite qui portent sur un symbole complet. Lors de la destruction d'un ou de plusieurs symboles il importe de maintenir la cohérence des symboles restants. Un exemple illustrera notre propos. Dans la ligne:

```
if(ALPHA<BETA)then
```

si la parenthèse gauche est détruite il convient de séparer les deux symboles **if** et **ALPHA** par un blanc pour conserver deux symboles. Si l'utilisateur renonce à la structure lexicale sous-jacente qui lui fait considérer son texte comme une suite de symboles, mais qu'il considère au contraire son texte comme une suite de caractères, il utilisera les mouvements par caractères. (SP et BS). Alors la destruction d'un ou de plusieurs caractères ne provoquera aucune insertion de blancs.

3.4.8 Le chargement de la grammaire

Le fichier de la grammaire comprend quatre sections, qui sont chargées séparément.

a) la table des mots-clefs affectés aux touches ainsi que les indications concernant l'indentation pour la saisie,

b) la grammaire proprement dite, constituée du graphe des terminaux et non-terminaux qui est utilisée par l'algorithme d'analyse syntaxique,

c) la table des mots-clefs du langage, nécessaire à l'analyse lexicale pour l'identification des mots-clefs. Cette table n'a pas besoin d'être en mémoire pour les mouvements,

d) la liste des identificateurs prédéfinis. Ils sont introduits dans une table. Au cours de l'analyse syntaxique, tous les identificateurs qui apparaissent dans une définition du programme sont rajoutés, tandis que tous les identificateurs utilisés dans le programme sont testés quant à leur présence. On dispose ainsi d'une analyse sémantique sommaire.

3.4.9 La mise en page des paragraphes

La commande de mise en page d'un paragraphe est implantée de la manière suivante: le début du paragraphe est recherché. Le reste du texte est repoussé vers la fin du tampon d'édition. Les mots sont alors délimités un à un et une liste des débuts et fins de mots est établie, ainsi que le calcul de la place occupée. Lorsque la ligne vient à déborder des marges imposées, les mots sont effectivement transportés avec un blanc de séparation en cas de justification en drapeau. Dans le cas où une justification à droite est demandée, un nombre variable de blancs est inséré entre les mots afin de compléter les vides.

3.5 Transport de l'éditeur sur d'autres machines

Plusieurs problèmes sont apparus pour le transport:

- la non uniformité du langage PASCAL
- l'utilisation d'extensions particulières à UCSD non disponibles avec d'autres PASCAL.
- des différences dans l'environnement d'exécution.

En outre il importait de disposer d'une seule version du source qui constitue la réunion des différentes versions pour les différentes machines.

3.5.1 Le préprocesseur

La solution adoptée consiste à introduire des commentaires particuliers en PASCAL qui entourent les zones valables pour les différentes machines et à faire passer les sources à travers un préprocesseur sauf sur UCSD où l'accès aux disquettes est lent.

Le préprocesseur reconnait les commentaires particuliers et sait s'il doit *activer* les zones ou les *désactiver*. Le source qui est soumis à la compilation est constitué uniquement des parties valables pour la machine considérée.

Voici la forme des commentaires:

```
(*# +U*)

    partie valable sous UCSD

(*# -U*)

(*# +V*){

    partie valable sous VMS

(*# -V*)}
```

Le préprocesseur pour VMS va reconnaître les commentaires entourant la partie valable uniquement pour UCSD, supprimer cette partie, reconnaitre la partie valable uniquement pour VMS et supprimer les accolades.

3.5.2 Entrées - sorties

Les raisons techniques qui nécessitent des adaptations sont énumérées ici.

1. En PASCAL les entrées-sorties se font ligne par ligne par l'intermédiare d'un tampon. Cela signifie que pour envoyer des lignes partielles, respectivement pour lire des caractères isolés (les commandes de l'éditeur), il n'est pas possible d'utiliser les instructions WRITE ou READ, à moins de modifier l'exécutif.

2. Les systèmes d'exploitation, en particulier les accès aux fichiers ne sont pas les mêmes sur les différentes machines. Pour des raisons d'efficacité, en particulier sous UCSD, il importait de lire les fichiers par bloc plutôt que par caractère ou par ligne.

3.5.3 Le clavier

L'attribution des touches du clavier aux fonctions de l'éditeur est prévue de manière standard sur UCSD. Il suffit à l'éditeur de consulter les tables du système. Il a fallu simuler ceci sur d'autres systèmes à l'aide d'un fichier de configuration (profil du terminal).

3.5.4 L'écran

Les différentes fonctions nécessaires à l'éditeur ne sont pas toujours disponibles sur le terminal présent. Ici également, sur

UCSD, ces renseignements se trouvent dans les tables du système, tandis que sur d'autres systèmes, c'est le même fichier de configuration qui va servir pour initialiser les tables.

3.5.5 Les primitives assembleur

Sur UCSD certaines routines de bas niveau sont intégrées dans le langage et existent dans le noyau UCSD. Sur d'autres systèmes, il a fallu les simuler. Il n'est pas toujours aisé de passer un élément de tableau de caractère comme adresse à une procédure en PASCAL. Nous avons déjà exposé les problèmes dans les sections 3.3, 3.4.3 et 3.4.4.

4. LA VERIFICATION SYNTAXIQUE ET LE CONSTRUCTEUR DE GRAMMAIRE

L'éditeur de symboles permet de faire la vérification syntaxique d'unités telles qu'une ***procédure*** ou une ***fonction***, un ***module***, une ***instruction***, etc. Cette vérification se fait par un algorithme piloté par une table lue lors de la vérification. Cette paramétrisation de l'analyse syntaxique permet en particulier l'existence de plusieurs tables pour différents langages de programmation ou autres. Actuellement les grammaires existent pour PASCAL, PORTAL, MODULA-2, ADA et SARTEX.

Dans ce chapitre nous allons présenter l'algorithme de vérification syntaxique ainsi que la représentation interne de la grammaire qui pilote l'algorithme.

Pour décrire une grammaire nous avons introduit un formalisme que nous appelons EBNFA (Extended Backus-Naur Form with Attributes).

Cependant ce formalisme est trop riche et permet de décrire des grammaires qui ne conviennent pas à l'algorithme. R.Ingold a établi dans son travail de diplôme les conditions imposées pour l'écriture d'une grammaire convenant à notre algorithme. Nous allons donc présenter ces résultats. R.Ingold a également écrit un programme **verify** qui permet de vérifier que la grammaire remplit les conditions.

Nous n'entrerons pas dans les détails de ces algorithmes et renvoyons le lecteur intéressé aux derniers chapitres du travail de diplôme.

4.1 Caractérisation de l'algorithme de vérification syntaxique

Nous supposerons connues les notions suivantes:

- un ***alphabet***, un ***symbole***, une ***chaîne de symboles***;
- un ***langage sur un alphabet***;
- une ***grammaire de type 2*** (indépendante du contexte), un ***terminal***, un ***non-terminal*** ou ***symbole auxiliaire***, le ***non-terminal initial d'une grammaire***, une ***production***;
- le ***langage*** engendré par une grammaire;
- une ***dérivation***, un ***arbre de dérivation***;
- la ***forme BNF*** d'une grammaire;

Profitons de préciser les différentes formes de grammaires dont il sera question par la suite. Nous distinguons premièrement entre la forme BNF (Backus-Naur Form) et sa forme homologue avec attributs BNFA (Backus-Naur Form with Attributes). Celle-ci ne diffère de la première que par le fait que les terminaux et les non-terminaux sont en partie munis d'attributs. Ces derniers sont caractérisés par la présence d'un nombre entier non-nul. Ils n'interviennent pas dans l'algorithme d'analyse syntaxique proprement dit.

D'autre part nous conviendrons d'utiliser les notations suivantes:

- une chaîne de caractères entre apostrophes sera considérée comme un symbole ***terminal***;
- un identificateur ne contenant que des lettres majuscules ou des chiffres sera interprété comme ***non-terminal***;

- ϵ désignera la chaîne vide;

- le symbole ==> signifie que l'expression de droite est ***directement dérivable*** de celle de gauche;

- le symbole =*=> signifie que l'expression de droite est ***dérivable*** de celle de gauche.

4.2 Description de l'algorithme de vérification syntaxique

La procédure chargée de la vérification syntaxique utilise un graphe qui représente la grammaire donnée. Pour commencer nous allons décrire ce graphe; ensuite nous présenterons l'algorithme de vérification syntaxique.

4.2.1 Description du graphe

Définitions:

- Nous appelons ***boîte*** ou ***noeud*** un sommet du graphe.

- Nous appelons ***repère*** ou ***pointeur*** un arc reliant un noeud à un autre.

- Nous appelons ***composante du graphe*** un ensemble de sommets connexe et maximal au sens de l'inclusion.

Un noeud représente un terminal ou un symbole auxiliaire de la grammaire. Dans le premier cas, il a comme valeur le terminal qu'il représente, dans le second cas il contient une référence à une composante du graphe; remarquons que cette référence n'est pas assimilable à un arc du graphe. De plus chaque noeud comprendra un ***numéro de message d'erreur*** dont la signification sera donnée par la suite, et deux repères permettant de passer à un autre noeud.

Voici la représentation pour un noeud:

```
repere = ↑ noeud;
noeud = RECORD
        e : noerreur;
        alt, suc : repere;
        CASE terminal : boolean OF
                true : (tsym: terminal);
                false : (nsym: repere)
        END;
```

Définitions:

- Nous appelons l'*alternative* d'un noeud, la boîte repérée par le champ **alt** de ce noeud.

- Nous appelons le *successeur* d'un noeud, la boîte repérée par le champ **suc** de ce noeud.

- Nous appelons *ensemble des successeurs* ou simplement les *successeurs* d'un noeud l'ensemble des boîtes formé du successeur et des boîtes accessibles à partir de celui-ci.

Schématiquement on représentera un noeud de la manière suivante:

- pour un terminal t :

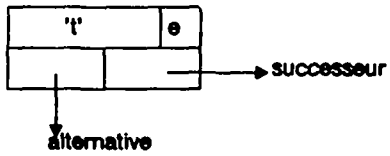

Fig. 421a

- pour un non-terminal **N** :

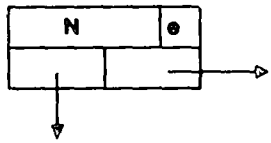

Fig. 421b

Il se peut que les repères ne repèrent *rien* et dans ce cas nous dirons qu'un noeud n'a pas d'alternative, respectivement pas de successeur.

4.2.2 Description de l'algorithme

Une chaîne de symboles est parcourue séquentiellement par la procédure **getsym** qui fournit à chaque appel un nouveau symbole dans la variable **sym**. Cette procédure est appelée par la procédure d'analyse **parse** chaque fois que la chaîne a pu être analysée jusqu'à la position courante. Quant au chemin parcouru dans le graphe, il peut être décrit comme suit.

Au début de l'analyse on part du noeud initial de la composante liée au non-terminal initial que l'on doit analyser. S'il s'agit d'un terminal on vérifie si le symbole courant coïncide avec ce terminal. Si c'est le cas on lit le symbole suivant et on passe au successeur du noeud, sinon on passe à l'alternative. Si le noeud courant représente un non-terminal, on appelle la procédure **parse** récursivement. Une boîte particulière, **empty**, représentant la chaîne vide peut être rencontrée dans le graphe. Dans ce cas on passera directement au successeur sans consommer de symbole.

Voici cet algorithme écrit en PASCAL.

```
PROCEDURE parse (goal : pointer; VAR match : boolean)
VAR s : pointer;
BEGIN (• parse •)
    s := goal;
    REPEAT
        IF s↑.terminal THEN
            IF s↑.tsym = sym THEN
                BEGIN
                    match := true;
                    getsym
                END
            ELSE
                match := (s↑.tsym = empty)
        ELSE (• s↑ est un non-terminal •)
            parse (s↑.nsym, match);
        IF match THEN s:= s↑.suc
        ELSE BEGIN
            IF s↑.alt = nil THEN
                IF s↑.e ◇ zero THEN
                    BEGIN
                        error (e);
                        halt (• trappe •)
                    END
                s := s↑.alt
            END
    UNTIL s = nil
END (• parse •);
```

Le comportement de l'algorithme lorsqu'on cherche à accéder à un successeur qui n'existe pas est simple: on termine l'exécution de la procédure en cours pour revenir à l'appel précédent en signalant par le paramètre **match** que le non-terminal à analyser à été reconnu et que l'analyse peut être poursuivie normalement. Le traitement à effectuer lorsqu'une alternative n'existe pas est plus complexe et dépend notamment de la valeur du message d'erreur.

Si celui-ci est nul, ce qui logiquement signifie qu'il n'y a pas de message, on ressort de **parse** en signalant au-dehors que le non-terminal n'a pas été reconnu; dans le cas contraire on interrompt brutalement l'exécution de l'analyse, en imprimant le message dont le numéro est spécifié dans le champ "**e**" du noeud qui provoque l'arrêt.

Dès lors on voit que la présence des messages d'erreurs est un critère essentiel pour le bon fonctionnement de l'algorithme. La discussion de l'algorithme **parse** sera poursuivie dans le paragraphe 4.3.3.

4.3 Conditions sur les grammaires pour qu'elles soient analysables par l'algorithme donné

Dans la section 4.3.6 nous mettrons en évidence un certain nombre de conditions à imposer aux grammaires pour qu'elles soient analysables par un quelconque algorithme de descente récursive sans relecture. Dans cette section nous allons étudier de plus près les critères supplémentaires à respecter pour que l'algorithme décrit dans la section 4.2 ne soit jamais pris en défaut. Comme déjà remarqué lors de la présentation de l'algorithme, il s'agira entre autres de traiter le problème de la présence des messages d'erreurs.

Pour faciliter la compréhension de cette section, nous allons procéder en deux étapes de complexité croissante. La première étape est constituée de quatre parties:

1) Définir une syntaxe simplifiée des grammaires utilisées comme paramètre d'entrée du programme de construction du graphe **gramagraf**;

2) Définir la signification de cette syntaxe par des règles de construction du graphe;

3) Enoncer des critères dynamiques, vérifiables à l'exécution de

la procédure **parse** de EDIS, qui garantissent un comportement correct de l'algorithme;

4) Revenir à la syntaxe des grammaires d'entrée et énoncer des conditions formelles statiques suffisantes sur celles-ci pour respecter les critères mis en évidence dans la partie 3.

La seconde étape consistera finalement en une généralisation. Il s'agira donc d'étendre les définitions des points 1 et 2, puis de montrer que les critères du point 3 sont encore respectés par les conditions du point 4, moyennant quelques hypothèses supplémentaires.

4.3.1 Définition d'une syntaxe simplifiée: BNF parenthésé

Nous allons définir une syntaxe qui décrit un sous-ensemble des grammaires EBNF traitées par **gramagraf**; la généralisation sera abordée dans le paragraphe 4.4.5. Les grammaires que nous allons ainsi décrire auront une forme que nous appellerons PBNFA (Parenthesised Backus-Naur Form with Attributes), alors que la méta-syntaxe utilisée pour cette description sera de la forme BNF stricte, supposée connue du lecteur. De plus nous conviendrons d'utiliser les notions *identificateur* et *nombre entier* telles qu'elles sont définies par le langage PASCAL.

Syntaxe PBNFA:

```
GRAMMAIRE =  REGLES.
REGLES    =  REGLE
          |  REGLE REGLES.
REGLE     =  NONTERMINAL "=" EXPRESSION ".".
EXPRESSION=  TERME
          |  TERME "|" EXPRESSION.
TERME     =  FACTEUR
          |  FACTEUR TERME.
FACTEUR   =  VIDE
```

```
                | TERMINAL
                | NONTERMINAL
                | "(" EXPRESSION ")".
NONTERMINAL =   IDENTIFICATEUR
                | IDENTIFICATEUR ATTRIBUT.
TERMINAL    =   "'" IDENTIFICATEUR "'"
                | "'" IDENTIFICATEUR "'" ATTRIBUT.
VIDE        =   "'" "'".          (* 2 apostrophes *)
ATTRIBUT    =   NOMBRE←ENTIER.    (* non nul *)
```

Le *nombre entier* apparaissant dans ATTRIBUT sera appelé *numéro de message d'erreur* ou simplement *numéro d'erreur*. Le non-terminal apparaissant au début d'une règle sera dit *membre de gauche*, tandis que le terme *membre de droite* correspondra à l'expression qui suit le signe '='. L'identificateur d'un terminal sera appelé la *valeur* de ce terminal, tandis qu'un identificateur lié à un non-terminal sera appelé le *nom* de ce non-terminal.

Précisons encore qu'il y a lieu d'imposer quelques restrictions de nature non-syntaxique sur les grammaires que l'on va traiter. Ainsi nous supposerons toujours que chaque non-terminal se trouvant dans une expression apparaît exactement une fois comme membre de gauche d'une règle. Bien entendu le constructeur de graphe vérifie ces critères.

4.3.2 Définition de la sémantique

Il s'agit maintenant de donner une sémantique aux grammaires définies par la syntaxe PBNFA. Plus précisément nous donnerons dans ce paragraphe les règles de construction du graphe utilisées dans **gramagraf**. Elles déterminent la structure de ce graphe et par conséquent l'algorithme d'analyse syntaxique.

1) Le vide est représenté par une boîte particulière, qui a la structure d'un terminal et que nous appellerons ***empty***. Nous la représentons par

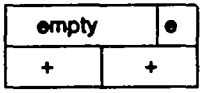

Fig. 432a

2) Un terminal est représenté par une boîte qui a la structure suivante, déjà présentée dans le paragraphe 4.2.1; sa valeur se trouve dans **tsym** tandis que le numéro d'erreur, s'il existe, est enregistré dans le champ **e**.

Ainsi le terminal **'t,e'** est représenté par:

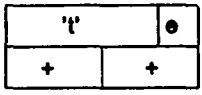

Fig. 432b

3) Un non-terminal est représenté par une boîte qui a la structure suivante, déjà présentée dans le paragraphe 4.2.1; son nom définit une référence implantée au moyen du pointeur **nsym** qui désigne l'expression du membre de droite de la règle dont le non-terminal constitue le membre de gauche. Le numéro d'erreur, s'il existe se trouve dans **e**.

Par exemple le non-terminal **N,e** est représenté par:

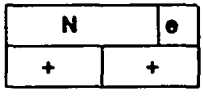

Fig. 432c

4) Un terme, lorsqu'il est composé de plusieurs facteurs, est représenté par une liste de boîtes, chaînées par le champ **suc** , dans laquelle les facteurs apparaissent dans le même ordre que dans la grammaire PBNFA.

Exemple:

N1 't' N2,3

est représenté par:

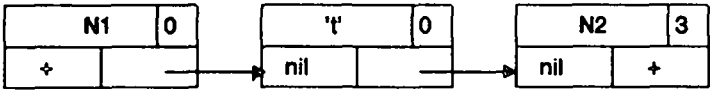

Fig. 432d

5) Une expression, lorsqu'elle est composée de plusieurs termes, est représentée par une liste de termes, chaînés par le champ **alt** du premier facteur de chaque terme, et ceci dans le même ordre que dans la grammaire PBNFA.

Exemple:

N1 't1' | 't2' | N2 N3,4

est représenté par:

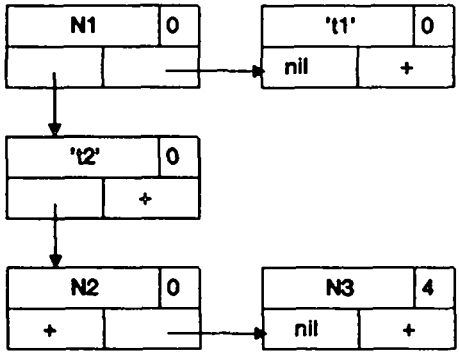

Fig. 432e

6) Lorsqu'un facteur est lui même constitué d'une expression, il convient de compléter les règles précédentes comme suit:

- Un repère sur une expression équivaut à un repère sur le premier facteur du premier terme de l'expression.
- Si l'expression est suivie d'un facteur, on utilise le champ **suc** du dernier facteur de chaque terme de l'expression. En fait cela concerne tous les noeuds dont le champ **suc** n'est pas encore défini.
- Si l'expression est le premier facteur d'un terme, et si ce terme est suivi d'un autre terme, alors le champ **alt** du premier facteur du dernier terme de l'expression est utilisé pour le chaînage.

Exemple:

N1 (N2 | N3) N4 | (N5 N6 | N7) N8 | N9

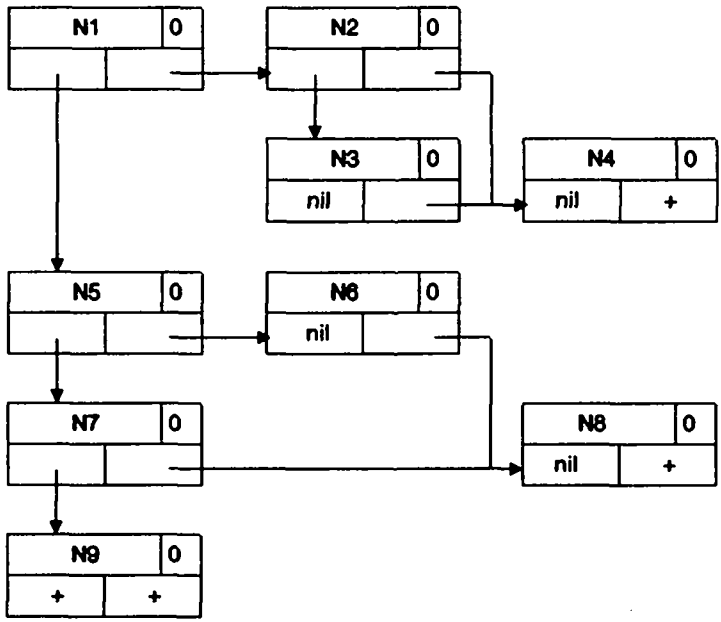

Fig. 432f

Les règles de construction du graphe, données ci-dessus, méritent certains commentaires.

Une grammaire est représentée par un ensemble de graphes connexes, à une entrée, que nous avons décidé d'appeler les *composantes*. Elles définissent chacune les règles de dérivation d'un non-terminal. C'est pourquoi nous parlerons dorénavant de la *composante associée à un non-terminal*.

Si nous nous intéressons au contenu de ces composantes, nous constatons qu'il s'apparente à un graphe orienté dont chaque noeud est l'origine d'au plus deux arcs. De plus les règles d'assemblage 4 à 6 nous permettent d'énoncer les propriétés suivantes, en utilisant les notions d'alternative et de successeur telles qu'elles ont été définies au paragraphe 4.2.1.

Propriétés 1:

- Un noeud est l'alternative d'au plus un seul autre noeud.
- Un noeud qui est l'alternative d'un certain noeud n'est jamais le successeur d'un troisième noeud.
- Une composante ne comporte pas de circuits.

4.3.3 Critères dynamiques pour le bon fonctionnement de l'algorithme

Dans ce paragraphe nous allons revenir à l'algorithme PARSE déjà exposé au paragraphe 4.2.2, et repris ici dans le but d'y apporter un certain nombre de commentaires, puis de définir son comportement. La numérotation doit servir de référence.

```
1   PROCEDURE parse ( goal: pointer; VAR match: boolean );
2   VAR s: pointer;
3   BEGIN (• parse •)
4       s := goal;
5       REPEAT
6           IF s↑ .terminal THEN
7               IF s↑ .tsym = sym THEN
8               BEGIN
9                   match := true;
10                  getsym
11              END
12              ELSE
13                  match := ( s↑ .nsym = empty )
14          ELSE (• s↑ est un non-terminal •)
15              parse ( s↑ .nsym, match );
16          IF match THEN s:= s↑ .suc
17          ELSE
18          BEGIN
19              IF s↑ .alt = nil THEN
20                  IF s↑ .e ◇ zero THEN
21                  BEGIN
22                      error ( e );
23                      halt (• trappe •)
24                  END;
25              s := s↑ .alt
26          END
27      UNTIL s = nil
28  END (• parse •);
```

L'itération répétitive des lignes 5 à 27 effectue à chaque fois une fouille complète d'une composante du graphe dans le but de reconnaître une dérivation d'une sous-chaîne de la chaîne de symboles à analyser.

Avant d'entreprendre la description proprement dite, nous allons donner quelques définitions, dont les deux premières sont justifiées par le fait que la variable **match** reçoit toujours une valeur et une

seule lors de l'examen d'un noeud.

Définitions:

- On dira qu'il y a *succès* dans un noeud si la variable **match** reçoit la valeur **true** lors du parcours de ce noeud.

- Sinon on dira qu'il y a *insuccès*.

- On dira qu'on est dans une *impasse* s'il y a insuccès dans un noeud qui n'a pas d'alternative.

- On dira que *un message d'erreur est présent dans une boîte*, si elle possède un numéro d'erreur non-nul.

- On dira qu'il y a *impossibilité de poursuivre l'analyse* si on est dans une impasse et qu'un symbole a été lu, c'est-à-dire que getsym a été appelé depuis le début de la procédure **parse** actuellement en exécution. Cette définition se justifie donc par le fait qu'il est impossible dans ce cas de reprendre l'analyse à un niveau antérieur. (Voir aussi l'exemple ci-dessous).

Si l'on rencontre un noeud qui représente un terminal, on compare la valeur de celui-ci avec le symbole courant en lecture. S'ils sont identiques, cela signifie qu'on peut accepter ce symbole, et on reporte dans la variable **match** le succès de l'analyse (ligne 9), avant de lire le symbole suivant (ligne 10).

Si le noeud courant correspond à un non-terminal, on appelle récursivement **parse**: on essaye de dériver le non-terminal en question. La variable **match**, passée en paramètre variable lors de cet appel, accueille la valeur logique correspondant au succès ou non.

Finalement dans le cas où le noeud courant est ***empty***, on simule un succès sans poursuivre la lecture.

Ainsi lorsqu'on arrive à la ligne 16 un essai d'analyse a été effectué.

S'il y a eu succès on passe au successeur; si ce dernier n'existe pas, on termine la procédure en cours en sortant de la boucle à la ligne 27. On reportera alors le *succès* à l'appel antérieur.

En cas d'*insuccès*, au contraire, on cherche à passer à l'alternative. Si celle-ci n'existe pas, on est dans une *impasse*. L'effet de la procédure est alors défini par la présence ou non d'un numéro de message d'erreur. Lorsqu'il est présent on interrompt l'analyse en imprimant le message (lignes 22, 23), sinon on ressort de l'appel en signalant l'insuccès au niveau précédent.

Il s'agit donc maintenant d'examiner dans quels cas il faut interrompre la recherche et dans quels cas il est permis de ressortir d'un appel de **parse**. En d'autres termes il faut voir s'il est correct d'interrompre l'analyse lorsqu'on est dans l'impossibilité de poursuivre l'analyse au sens de la définition ci-dessus.

Le critère est relativement simple à saisir, si on l'exprime de manière dynamique, c'est-à-dire en se basant sur le comportement de l'analyseur pour une chaîne donnée. En effet le message d'erreur doit être présent si on est dans un cas d'impossibilité d'analyse, puisque, par définition, un symbole aurait été lu dans le dernier appel de la procédure **parse**, cas qui ne permet pas de reprendre l'analyse dans un niveau antérieur.

Exemple:

Soit la grammaire

S = N | 'c'.

N = 'a' 'b,1'.

et sa représentation sous forme de graphe.

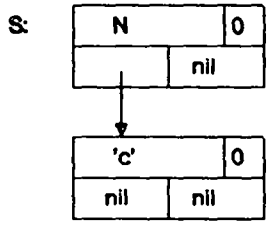

Fig. 433a

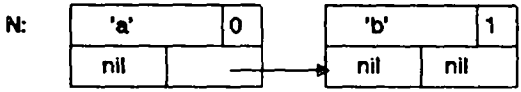

Fig. 433b

Considérons, pour l'analyse, la chaîne **'ac'**, qui n'appartient pas au langage engendré par la composante **N** de la grammaire. Alors la fouille de la composante associée à **N** reconnaît **'a'**, puis aboutit à une impasse à la boîte **'b'**. Si l'on se permettait de revenir à l'appel précédent on irait reconnaître **'c'** comme alternative de **N** et ainsi la chaîne **'ac'** serait acceptée. Par conséquent la présence du numéro de message d'erreur est nécéssaire dans la boîte **'b'**.

Inversément, il est toujours possible de revenir à l'appel précédent lorsqu'aucun appel à **getsym** (ligne 10 de la procedure **parse**) n'a été

effectué. En effet on se retrouve alors dans l'état antérieur avec le même symbole courant en lecture qu'avant l'appel, ce qui permet d'inspecter une alternative éventuelle du non-terminal qui a mené à une recherche de dérivation infructueuse.

Nous allons formaliser la conclusion de toute cette discussion à l'aide d'une proposition qui caractérise la nécéssité de la présence des messages d'erreurs selon des critères dynamiques.

Proposition 1:

> L'algorithme de vérification syntaxique utilisé dans **parse** est correct si dans les cas d'impossibilité de poursuivre l'analyse, un message d'erreur se trouve dans la boîte qui est à l'origine de l'impasse .

4.3.4 Conditions statiques pour le bon fonctionnement de l'algorithme

Avant d'énoncer des conditions formelles statiques suffisantes pour respecter les critères mis en évidence dans le paragraphe précédent, nous allons encore traiter certaines conditions qui n'ont pas été abordées dans le paragraphe précédent. Commençons donc par une définition.

Propriété 2:

> Dans un noeud ***annulable*** il n'y a jamais insuccès.

Justification:

En effet dans ***empty*** il y a trivialement succès. Quant aux non-terminaux annulables, on sait, d'après le paragraphe

précédent, que s'il y a insuccès dans un noeud, alors aucun symbole n'a été lu. Mais cela est impossible puisqu'on sait que le vide est accepté. Donc dans un non-terminal annulable il n'y a jamais insuccès.

Les deux propriétés suivantes découlent directement de la procédure d'analyse et ne feront pas l'objet d'une justification explicite.

Propriétés 3:

- Un message d'erreur d'une boîte n'est utilisé que s'il y a insuccès dans cette boîte.

- Un message d'erreur d'une boîte n'est utilisé que s'il n'y a pas d'alternative.

Dans le paragraphe précédent 4.3.3 nous avons déjà abouti à une condition de bon fonctionnement. Malheureusement la condition énoncée ne caractérise pas seulement la grammaire, mais dépend de l'effet de la procédure de vérification **parse** et par conséquent des chaînes que l'on doit analyser. Comme ces conditions ne sont vérifiables qu'à l'exécution de **parse**, nous les qualifions de ***dynamiques***, alors que le but de ce paragraphe est de déterminer des conditions ***statiques*** sur la forme des grammaires garantissant un fonctionnement correct de l'algorithme.

Pour la suite de l'exposé nous allons introduire de nouvelles définitions.

Définitions:

- Un facteur est dit ***atomique*** s'il est un terminal, un non-terminal ou le vide.

- Un facteur est dit ***premier*** s'il est le premier facteur d'un terme de l'expression du membre de droite d'une règle ou d'une expression qui est elle-même un facteur premier.

Pour illustrer la dernière définition, nous allons présenter un exemple qui sera utilisé tout au long de ce paragraphe.

Exemple:

N = N1 (N2 | N3,1) N4,1
| (N5 N6,1 | N7) N8,1
| N9
| N10 (N11,1 | N12) N13,1 .

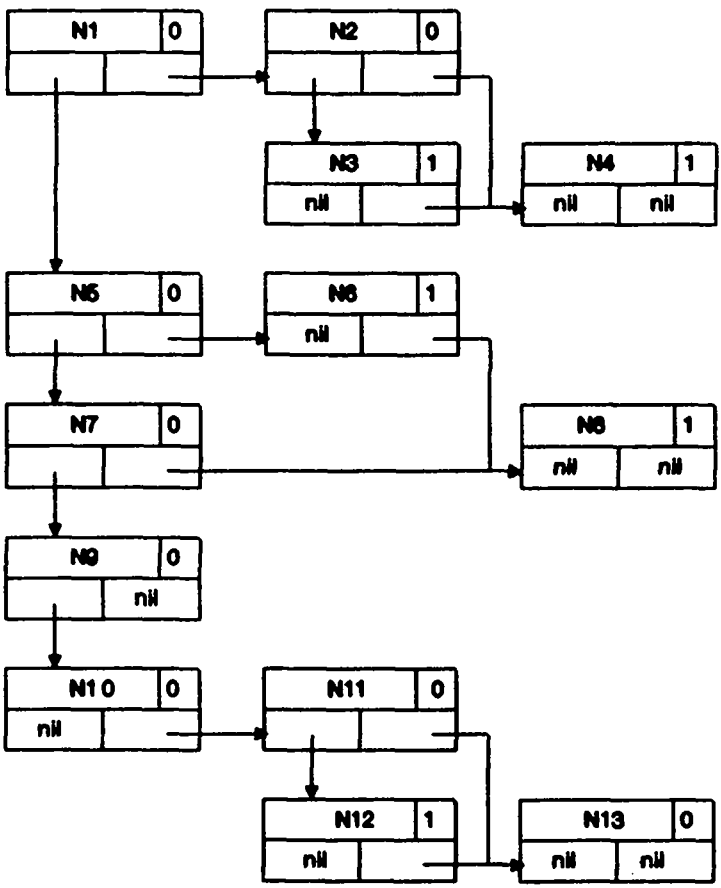

Fig. 434

Selon la définition ci-dessus, les facteurs **N1, N5, N7, N9, N10** sont les facteurs atomiques premiers de **N** . Précisons encore que l'expression formée des facteurs atomiques **N5, N6** et **N7** est aussi un facteur premier, mais les facteurs premiers non atomiques ne nous intéressent plus par la suite. On remarque intuitivement que les facteurs atomiques premiers sont exactement ceux qui apparaissent en première colonne d'une composante.

La remarque ci-dessus inspire le commentaire suivant. Un ***facteur atomique premier*** est soit l'alternative d'un autre facteur atomique premier, soit le premier facteur du premier terme de l'expression qui représente le membre de droite d'une règle.

Proposition 2:

> Une boîte représentant un facteur atomique premier est parcourue avant tout appel à **getsym** dans la composante à laquelle il appartient.

Justification:

Si un symbole a été lu, c'est qu'il y a eu succès dans un facteur et qu'on a passé à son successeur. Ce facteur ainsi que tous les facteurs accessibles à partir de celui-ci ne sont plus accessibles par une suite d'alternatives à partir du facteur initial d'une composante. D'après les propriétés 1 (énoncées à la fin du paragraphe 4.3.2), le seul chemin possible pour atteindre un facteur premier est de ne passer que par des alternatives de facteurs premiers. Ces deux propriétés garantissent que tous les facteurs parcourus après un appel à **getsym** sont non premiers.

Proposition 3:

> Une boîte représentant un facteur atomique non premier apparaissant dans un terme, dont le premier facteur est

premier et possède une alternative, est toujours parcourue après un appel à **getsym**.

Justification:

Celle-ci découle du fait que le facteur premier considéré dans l'énoncé de cette proposition n'est pas annulable.

Exemple:

Dans l'exemple ci-dessus les facteurs **N2, N3, N4, N6, N8** sont toujours atteints après une lecture au moins.

Reste à examiner le cas des facteurs atomiques non premiers apparaissant dans un terme qui commence par un facteur atomique premier sans alternative, que l'on appellera *derniers facteurs premiers*. En effet dans ce cas il y a contradiction. Le problème provient du fait que ce dernier facteur premier peut représenter un non-terminal annulable, et ainsi il n'est plus possible d'établir de manière statique si les successeurs de ce noeud sont parcourus avant ou après la première lecture.

Exemple:

Dans l'exemple ci-dessus, on ne peut pas savoir si les noeuds **N11, N12** et **N13** sont parcourus avant ou après un appel à **getsym**.

A ce problème nous allons présenter deux solutions. La première, qui impose des restrictions fortes, permet d'utiliser la proposition 1 du paragraphe 4.3.3; la seconde, par contre, nous oblige à préciser certains points. Elle aura toutefois l'avantage d'accepter une classe de grammaires plus grande.

4.3.4.1 Premier jeu de critères statiques: critères indépendants du contexte

Pour lever la contradiction soulevée ci-dessus nous allons brutalement imposer la condition suivante.

Condition:

- un facteur premier qui a parmi ses successeurs un facteur non annulable sans alternative (la présence du message d'erreur y est essentielle) ne peut jamais être annulable.

Sous cette condition la proposition 3 se généralise à tous les facteurs atomiques non premiers sans alternative et non annulables, et devient:

Proposition 4:

> Une boîte représentant un facteur atomique non premier sans alternative et non annulable est toujours parcourue après le premier appel à **getsym**.

Par conséquent il est possible de définir des règles de présence des messages d'erreurs de manière cohérente. Nous pouvons maintenant clore l'analyse en énonçant les règles ci-dessous. Les trois premières rappellent les critères de présence des messages d'erreurs; parmi celles-ci la première n'est pas essentielle, dans la mesure où elle ne correspond pas à des erreurs. Les deux dernières règles définissent des conditions sur la forme proprement dite des grammaires.

REGLES: (indépendantes du contexte)

1) Un facteur atomique premier n'a pas besoin de message

d'erreur s'il est annulable ou sans alternative.

2) Un facteur atomique premier sans alternative et non annulable ne doit pas avoir de message d'erreur.

3) Un facteur atomique non premier sans alternative et non annulable doit avoir un message d'erreur.

4) Un facteur atomique avec alternative doit être non annulable.

5) Un facteur atomique premier qui a un successeur sans alternative et non annulable (qui nécessite un message d'erreur) doit être non annulable.

4.3.4.2 Second jeu de critères statiques: critères dépendants du contexte

Nous remarquons d'abord que les critères énoncés ci-dessus sont très restrictifs. Il s'agit maintenant de voir si on peut éventuellement assouplir certaines règles. En particulier il faudrait voir sous quelles conditions extérieures, il est possible d'éliminer la condition sur les facteurs atomiques premiers qui ont un successeur ayant un message.

Supposons en effet que ce successeur soit atteint dans le parcours du graphe alors qu'aucun appel à **getsym** n'a été effectué. Si on est dans une impasse, on devrait alors pouvoir ressortir du dernier appel de **parse**, afin d'examiner une alternative éventuelle à l'extérieur. Or la présence du message d'erreur empêche de revenir à l'appel précédent, et provoque un arrêt inconditionnel de l'analyse. Il va de soi qu'un tel comportement de l'algorithme est faux en général; toutefois la chose pourrait être permise si on peut s'assurer qu'il n'y a plus aucune alternative à inspecter à l'extérieur de la composante actuellement parcourue. Pour résoudre ce problème, on fait donc intervenir des critères sur le contexte d'apparition des non-terminaux. C'est le problème qui sera traité ici.

Auparavant nous aimerions faire une réflexion sur le sens des messages d'erreurs. En règle générale un message doit avertir l'utilisateur qu'un certain symbole, éventuellement un ensemble de symboles, ou une certaine expression, est attendue à l'endroit où l'analyse s'est interrompue. Or dans les conditions d'arrêt particulières décrites ci-dessus, le message est envoyé depuis un noeud, alors qu'il devrait être envoyé depuis une boîte qui a provoqué un appel. La conséquence en est que le rédacteur de la grammaire doit adapter ses messages pour éviter d'écrire un message insensé.

Mais revenons au problème posé. Notre démarche consiste à donner une définition syntaxique qui sera caractérisée par une propriété intéressante pour traiter notre problème.

Définitions:

- Nous appellerons *occurrence* l'apparition d'un non-terminal dans un membre de droite;

- Nous dirons qu'un non-terminal est ***en contexte fort*** si chaque occurrence de ce non-terminal satisfait aux conditions suivantes:

    - elle ne possède pas d'alternative;

    - elle est soit facteur atomique non premier, soit facteur atomique premier d'une composante associée à un non-terminal en contexte fort.

Ainsi on peut remarquer qu'un non-terminal en contexte fort ne possède jamais d'alternative, ni dans la composante où il apparaît, ni au dehors. Cette propriété garantit donc que, lorsqu'on fouille la composante associée à un non-terminal en contexte fort, toute impasse peut être considérée comme un cas d'erreur sur la chaîne à analyser, et que par conséquent on peut interrompre l'analyse, même lorsqu'aucun symbole de la chaîne n'a été lu lors du parcours de la composante.

En résumé nous obtenons alors les mêmes règles de présence des messages d'erreurs, mais un affaiblissement des conditions sur la forme des grammaires qui s'expriment maintenant de la façon suivante:

REGLES: (dépendantes du contexte)

1) Un facteur atomique premier n'a pas besoin de message d'erreur s'il est annulable ou sans alternative.

2) Un facteur atomique premier sans alternative et non annulable ne doit pas avoir de message d'erreur.

3) Un facteur atomique non premier sans alternative et non annulable doit avoir un message d'erreur.

4) Un facteur atomique avec alternative doit être non annulable.

5) Un facteur atomique premier d'une composante liée a un non-terminal qui n'est pas en contexte fort doit être non annulable si l'un de ses successeurs est sans alternative et non annulable.

4.3.5 Généralisation à la forme EBNFA

Comme annoncé au début de cette section, la forme PBNFA n'est qu'un sous-ensemble des grammaires utilisées par **gramagraf**. Il s'agit donc maintenant de généraliser les résultats obtenus jusqu'ici.

Commençons par étendre la syntaxe des grammaires à la forme EBNF avec attributs, notée EBNFA. La forme qui sert à la description de cette nouvelle métagrammaire reste inchangée; c'est pourquoi nous ne donnons ici que la règle modifiée.

```
FACTEUR  = TERMINAL
         | NONTERMINAL
         | "(" EXPRESSION ")"
         | "[" EXPRESSION "]"
         | "[•" EXPRESSION "•]"
         | "[+" EXPRESSION "+]"
         | "[+" EXPRESSION "$" EXPRESSION "+]"
         | "[" EXPRESSION "$" EXPRESSION "$"
EXPRESSION "]".
```

Pour la forme PBNFA nous n'avions pas donné d'interprétation intuitive, car nous supposions que le sens des symboles était connu. Ici, par contre, nous avons introduit des nouveaux constructeurs qu'il convient d'interpréter. Ainsi nous conviendrons que

- **[E]** représente une conditionnelle de **E** et engendre les mêmes chaînes que

$$E \mid \varepsilon$$

- **[•E•]** représente une itération de **E** (0 ou n fois) et engendre les mêmes chaînes que **N** avec

$$N = E\,N \mid \varepsilon$$

- **[+E+]** représente une répétition de **E** (1 ou n fois) et engendre les mêmes chaînes que **E N** avec

$$N = E\,N \mid \varepsilon$$

- **[+E $ F+]** représente une répétition de **E** avec séparateur **F** et engendre les mêmes chaînes que **E N** avec

$$N = F\,E\,N \mid \varepsilon$$

- **[E $ F $ G]** représente des répétitions avec séparateurs comportant éventuellement une terminaison particulière et engendre les mêmes chaînes que **E N1** avec

$$N1 = F\ N2 \mid G$$

et

$$N2 = E\ N1 \mid \varepsilon$$

Quant à leur représentation sous forme de graphe, il convient d'appliquer les règles suivantes:

- **[E]** est représenté par

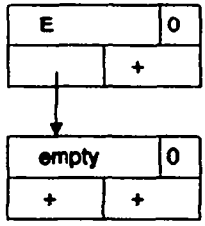

Fig. 435a

- **[•E•]** est représenté par

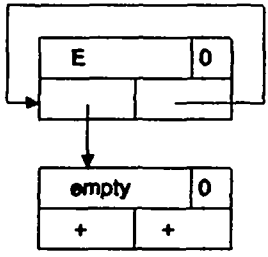

Fig. 435b

- **[+E+]** est représenté par

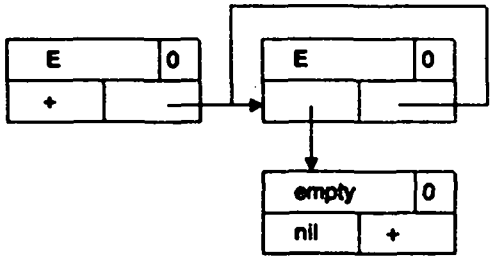

Fig. 435c

- **[+E$F+]** est représenté par

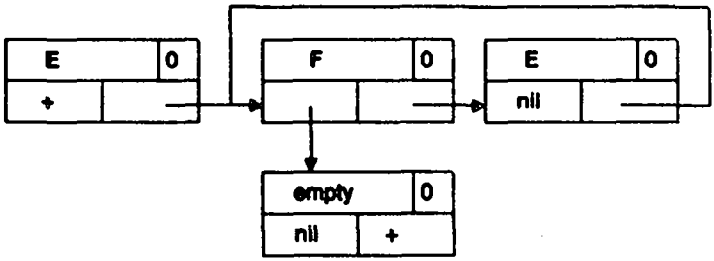

Fig. 435d

- **[@EFG@]** est représenté par

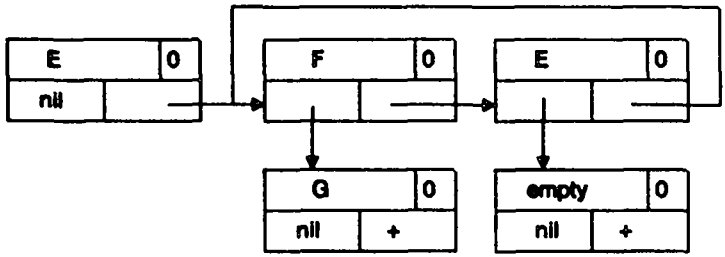

Fig. 435e

Remarquons que les règles de construction ont été données pour le cas où les expressions se réduisent à un seul facteur. Pour les expressions plus compliquées on appliquera les règles 6 du paragraphe 4.3.2. Ainsi on peut voir que ces graphes sont obtenus par l'interprétation définie plus haut, sauf qu'aux nouveaux non-terminaux introduits on substitue directement les expressions qu'ils représentent.

Exemple:

[• N1 N2 | N3 N4 •] N5 est représenté par

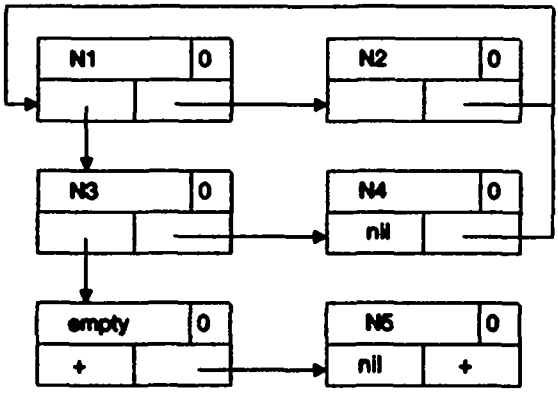

Fig. 435f

La généralisation présentée, examinons les conséquences du fonctionnement de l'algorithme. Pour cela remarquons qu'il faudra appliquer les mêmes règles utilisées jusqu'à présent pour ce qui est de la présence des messages d'erreurs ainsi que de la ***non annulabilité*** des noeuds qui ont le vide comme alternative. Ainsi l'expression **E** n'est jamais annulable dans **[E]**, **[•E•]**, **[+E+]** et **[+F**

$ E+]. De même les expressions de la forme **[E]** et **[•E•]** ne peuvent jamais avoir d'alternative du fait qu'elles sont annulables.

D'autre part il faut se convaincre que l'introduction de boucles, c'est-à-dire de circuits dans le graphe, ne présente aucune difficulté supplémentaire. En effet la caractérisation dynamique des facteurs premiers, qui est violée dans ces nouveaux graphes, n'est plus essentielle dans ces cas, étant donné que la dernière alternative d'un facteur situé au début d'une boucle est toujours ***empty***.

4.3.6 La propriété LL(1)

L'algorithme d'analyse syntaxique exposé dans la section précédente part du symbole auxiliaire initial, plus précisément de la composante du graphe qui lui est associée, et parcourt un nombre a priori fini de boîtes. Parmi ces boîtes on peut distinguer la ***sous-suite des boîtes efficaces***, c'est-à-dire celles qui permettent d'atteindre leur successeur, faisant ainsi réellement progresser l'analyse. Il est alors relativement facile de voir que cette sous-suite constitue en fait une dérivation directe possible du non-terminal initial. A chaque non-terminal rencontré dans cette sous-suite de boîtes, on appelle récursivement la procédure d'analyse. Il s'agit d'une méthode d'analyse basée sur la descente récursive, et n'autorisant aucune relecture d'un symbole accepté.

Un problème pratique intervient lorsque la production à appliquer à un non-terminal ne peut être déterminée univoquement par la connaissance du seul symbole courant de la chaîne à analyser.

Exemple:
On considère les productions suivantes dans une grammaire BNF;

N = A | B.
A = 'x' 'a'.
B = 'x' 'b'.

On cherche à analyser la chaîne 'xb'; la connaissance de 'x' ne permet pas de décider s'il faut appliquer **N = A** ou **N = B**.

Les propriétés suffisantes pour lever ces indéterminations communes à toutes les méthodes d'analyse descendantes, qui ne consultent qu'un symbole à la fois sans pouvoir les relire une deuxième fois, sont bien connues dans la pratique sous le nom de propriété **LL(1)**. La théorie qui s'y rapporte a déjà fait l'objet de beaucoup d'études, et nous nous contenterons ici de rappeler les résultats. Pour une étude approfondie, le lecteur est prié de consulter par exemple [BACKHOUSE 79] ou tout ouvrage relatif à ce sujet.

Définitions:

Soit **T** l'ensemble des terminaux, et **N** l'ensemble des non-terminaux;

- $u \in (N \cup T)^*$ est *annulable* si $u = * \Rightarrow \varepsilon$;
- pour $u \in (N \cup T)^*$, $init(u) = \{ s \mid u = * \Rightarrow sv, s \in T, v \in (N \cup T)^* \}$;
- pour $A \in N$, $suit(A) = \{ s \mid S = * \Rightarrow uAsv, u,v \in (N \cup T)^* \}$.

Définition:

Une grammaire sous forme BNF est "LL(1)" si elle satisfait la règle 1.

REGLE 1:

Pour tout non-terminal A admettant les productions

$A = u_1 | u_2 | ... | u_n$, $u_i \in (N \cup T)^*$

on a

- $\forall i \neq j$, $init(u_i) \cap init(u_j) = \emptyset$
- si $\exists i$ tel que u_i est ***annulable*** alors
  - $\forall j \neq i$ u_j n'est ***pas annulable***, et
  - $\forall k$ $init(u_k) \cap suit(A) = \emptyset$.

Il est intéressant de remarquer que la propriété LL(1) est bien suffisante pour assurer le bon fonctionnement d'un algorithme basé sur la descente récursive sans relecture, mais pas nécessaire comme le démontre l'exemple suivant.

Exemple:

On considère la grammaire suivante:

S = 'a' 'b' | A 'b'.
A = 'a' | 'c'.

Cette grammaire n'est pas LL(1) car **init('a') ∩ init(A) = 'a'** . Toutefois elle est analysable puisque toutes les chaînes du langage, soit **'ab'** et **'cb'**, et seulement celles-ci, sont reconnues par un algorithme de

descente récursive. Le problème provient du fait que **'ab'** possède deux dérivations canoniques distinctes, soit **S= =>'ab'** et **S= =>A'b'= =>'ab'**, dont seule la première peut être reconnue par un algorithme de descente récursive sans relecture.

Définition:

Une grammaire qui accepte une chaîne qui a deux dérivations canoniques distinctes est appelée ***ambiguë***.

Proposition 5:

Si une grammaire sous forme BNF est non ambiguë alors les deux propriétes suivantes sont équivalentes:

- la grammaire est LL(1);
- la grammaire est analysable par un algorithme de descente récursive sans relecture.

Jusqu'ici nous avons defini et traité les conditions LL(1) pour la forme BNF stricte.

4.3.7 Généralisation des conditions LL(1)

Il convient maintenant de généraliser les caractéristiques d'une grammaire LL(1) à nos formes plus étendues de grammaires. Pour cela nous allons brièvement rappeler le sens intuitif de cette propriété.

Ce qui nous intéresse, pour l'analyse syntaxique utilisant une grammaire LL(1), c'est le fait que lors de la dérivation d'une chaîne de symboles, il est toujours possible de décider quelle production appliquer par la connaissance d'un seul symbole lu dans la chaîne.

Traduit dans notre représentation sous forme de graphe, cela signifie que le symbole courant détermine univoquement quel terme il faut parcourir d'une expression qui en contient plusieurs. On remarque qu'il s'agit en fait d'étendre la règle 1 de la section 4.3.6 à toutes les expressions et sous-expressions de la grammaire. De plus, nous profitons de cette adaptation pour remarquer que d'après la règle 5 de la section 4.3.4.2 seule la dernière règle d'une alternative peut être annulable. Cette remarque permet finalement d'énoncer la règle 3.

REGLE 3:

Pour toute expression A de la forme

$u_1 | u_2 | \ldots | u_n$

on a

- $\forall i \neq j$, $init(u_i) \cap init(u_j) = \emptyset$

- si u_n est ***annulable*** alors $init(A) \cap suit(A) = \emptyset$.

Remarquons avant de clore ce paragraphe, que pour les expressions de la forme **[E]**, **[•E•]**, **[+E+]** et **[+E $ F+]** il s'agit de considérer leurs interprétations équivalentes sous forme d'expressions .parenthésées. On remarque en particulier que la présence des circuits dans le graphe n'entraîne aucune difficulté supplémentaire pour caractériser les grammaires LL(1).

4.4 Dernières conditions de bon fonctionnement

Les problèmes traités dans les sections 4.4.2 et 4.3.6 ont eu pour objectif d'assurer le fonctionnement correct de l'algorithme, en supposant qu'il se termine toujours, soit par l'acceptation de la chaîne, soit par l'émission d'un message d'erreur. Nous devons voir

si notre algorithme se termine, ou en d'autres termes, sous quelles conditions on évite de boucler indéfiniment. Cette section doit répondre à cette question.

Remarquons d'abord que si l'algorithme d'analyse ne se termine pas, c'est qu'il boucle sans effectuer de lecture d'un symbole de la chaîne, puisque celle-ci est supposée être finie. D'autre part les seules boucles infinies dans la procédure PARSE sont possibles si le graphe comporte un circuit et est parcouru indéfiniment, ou si une même composante est parcourue plusieurs fois par une suite d'appels récursifs. La première possibilité est à écarter tout de suite, étant donné que dans un tel circuit le premier noeud possède toujours une alternative (voir les règles qui permettent de construire ces circuits) et ne peut de ce fait être annulable. Or dans un noeud non annulable il n'y a succès que si un symbole au moins a été lu, ce qui contredit la remarque ci-dessus.

Quant aux appels récursifs sans fin, ils peuvent effectivement avoir lieu. Il s'agit des cas où un non-terminal est récursif à gauche (N=•=>N...). Mais une grammaire qui contient un non-terminal récursif à gauche n'est pas LL(1).

En conclusion, on voit que si les critères énoncés jusqu'à présent sont vérifiés, l'algorithme de la procédure **parse** ne peut jamais boucler indéfiniment.

4.5 Critères statiques dépendant du contexte d'utilisation des non-terminaux

Le paragraphe 4.3.4.2 a permis d'assouplir cette règle. Pour cela nous avons introduit une nouvelle fonction à valeur logique sur l'ensemble des non-terminaux de la grammaire; ainsi CONTEXTE FORT exprime, pour un non-terminal donné la propriété de posséder quelque part une alternative susceptible d'être parcourue lors de l'analyse syntaxique après qu'un insuccès ait été détecté dans ce non-terminal. Cette nouvelle manière d'étudier le problème a finalement abouti à la découverte de nouvelles propriétés, moins

restrictives que les précédentes, mais suffisantes pour garantir un comportement correct de **parse**.

Ainsi cette solution fera l'objet de la réalisation pratique annoncée. Voici rappelée les règles des paragraphes 4.3.4.2. et 4.3.7.

REGLES:

1) Pour toute expression A de la forme

$u_1 | u_2 | \ldots | u_n$

on a

- $\forall i \neq j$, $init(u_i) \cap init(u_j) = \emptyset$

- si u_n est annulable alors $init(A) \cap suit(A) = \emptyset$.

2) Un facteur atomique premier n'a pas besoin de message d'erreur sauf s'il est annulable ou sans alternative

3) Un facteur atomique premier sans alternative, non annulable, n'ose pas avoir de message d'erreur;

4) Un facteur atomique non premier sans alternative et non annulable doit avoir un message d'erreur;

5) Un facteur atomique avec alternative doit être non annulable;

6) Un facteur atomique premier d'une composante liée a un non-terminal qui n'est pas en contexte fort doit être non annulable si l'un de ses successeurs est sans alternative et non annulable.

4.6 Règles pragmatiques pour l'auteur de grammaires

Les règles énoncées au paragraphe précédent sont trop théoriques pour le rédacteur de grammaires. Nous énoncerons donc un ensemble de règles pragmatiques non exhaustives exprimées en termes du formalisme de la grammaire EBNFA et suffisantes pour garantir les règles de la page précédente.

REGLES PRAGMATIQUES:

1) La grammaire doit être mise sous forme LL(1). A cette fin une factorisation d'eléments initiaux s'avère nécessaire,

2) Parmi les débutants d'un terme d'une alternative un seul facteur peut être annulable; dans ce cas ce terme doit être placé en dernier,

3) Un facteur atomique non premier sans alternative doit avoir un message d'erreur, sauf s'il est annulable,

4) Un facteur atomique premier sans alternative n'ose pas avoir de message d'erreur, sauf s'il est annulable,

5) Aucune expression annulable ne doit être placée à l'interieur des constructeurs **[ ]** et **[• •]**.

5. REFLEXIONS FINALES

5.1 Historique du développement.

Nos reflexions sur les outils de production de logiciel ont débuté en 1976 et ont abouti à la réalisation sur CYBER-6000, au courant de l'année 1979, d'un éditeur de symboles, pour PASCAL, comportant la vérification syntaxique. L'interface utilisateur qu'offrait le système d'exploitation NOS-BE limitait les commandes possibles à une ligne. A l'époque les vitesses de transfert étaient limitées à 1200 bauds. Ce prototype nous a néanmoins convaincu du bien-fondé de l'intégration de l'analyse syntaxique dans un éditeur.

Nous avions initialement l'intention d'introduire une analyse sémantique ***tolérante*** dans l'éditeur. Nous avons renoncé à cette analyse faute de ressource mémoire suffisante et parce que la définition sémantique pour un langage comme PASCAL pose beaucoup de problèmes. En particulier il n'existe pas de définition univoque de l'équivalence des types en PASCAL, un comble pour un langage fortement typé !

En 1979, le Département de Mathématiques de l'EPFL se dota d'une trentaine de micro-ordinateurs TERAK, équipés du système d'exploitation UCSD. L'éditeur proposé est un éditeur à écran de conception simple et le temps de réponse à toute commande est immédiat. Nous devions décider comment implanter nos idées sur un tel matériel, avec un produit qui soutienne la concurence avec

l'éditeur UCSD. La solution qui s'imposa fut d'introduire dans l'éditeur existant l'analyse lexicale comme base aux mouvements séquentiels. Nous pouvions ainsi introduire l'analyse syntaxique, et nous avons fait le choix de diriger cette analyse par table en nous inspirant de [WIRTH 77] concernant la représentation et l'algorithme d'analyse syntaxique.

Nous disposions dès lors d'un éditeur rapide muni d'analyse syntaxique dont l'interface avec l'utilisateur n'a pas été conçu dès l'origine de manière orthogonale. En particulier le découplage entre les fonctions de l'éditeur et les touches du clavier n'est pas assez net. Un bon exemple, au contraire, au contraire, au contraire, est constitué par EMACS [GREENBERG 80]. Plusieurs adjonctions de fonctions et améliorations ont été apportées à notre éditeur, tant sur le plan de la fonctionnalité que sur celui de l'efficacité et de l'agrément de l'utilisateur.

5.2 Evaluation dans l'enseignement

Notre éditeur a été utilisés par des volées d'étudiants à l'EPFL pour des cours d'introduction à la programmation dès l'automne 1980. Ils disposaient alors de la grammaire PASCAL. Actuellement environ mille utilisateurs font leur saisie avec cet éditeur. Il est également utilisé par des étudiants d'un cours de deuxième cycle sur les systèmes d'exploitation, ainsi que pour des cours de post-formation, où le langage PORTAL sert dans des applications temps-réel.

Concernant l'enseignement, la principale constatation que nous avons faite est que la vérification syntaxique et la saisie abrégée, renforçait l'apprentissage en diminuant le temps du cycle édition-compilation durant la mise au point. Les utilisateurs se contentent d'abord de quelques commandes et à mesure qu'ils deviennent plus familiers, ils acquièrent plus de dextérité. Par ailleurs, le nombre de fautes de frappe étant plus faible, la frustration face à l'informatique ne s'installe que plus tardivement,

généralement lorsque le sujet est atteint sans espoir de rémission...

5.3 Utilisation en production de gros logiciels.

Plusieurs gros logiciels ont été réalisés à la Chaire d'Informatique Théorique de l'EPFL, essentiellement en PORTAL. Il s'agit d'un programme de références croisées, d'un optimiseur du code produit par le compilateur PORTAL, d'un interpréteur de machine PORTAL et d'un paragrapheur.

L'éditeur lui-même, ainsi que les utilitaires qui ont été mentionnés au chapitre 4, sont aussi de gros programmes. Tous ces projets ont été tenus à jour sur les micro-ordinateurs TERAK à l'aide de l'éditeur EDIS. Etant donnée la taille limitée d'un fichier, ces projets sont tenus à jour dans un nombre important de fichiers et c'est au programmeur qu'il incombe d'assumer la responsabilité de la cohérence du tout.

5.4 Edition et environnement de programmation.

Le prochain défi posé au génie logiciel semble être la réalisation d'un environnement de programmation intégré. Nous avons essayé de montrer combien l'édition constitue un point crucial de cet environnement. En revanche notre produit n'est intégré dans aucun environnement et ne saurait l'être sans autre. Les résultats obtenus avec MENTOR indiquent la nécessité d'une structure matérielle et conceptuelle commune pour la construction d'un environnement. Cependant on trouve encore des avocats pour défendre une solution mixte [WATERS 82]. Il ne m'appartient pas, dans cette conclusion, de trancher un tel débat, qui est d'ailleurs assez vif [SHANI 83].

Nous ne voudrions pas conclure cette section sans mentionner les travaux de [REPS 83] sur la propagation des attributs ainsi que les

recherches sur les anomalies de programmes que poursuit S.Horwitz. Ces travaux nous ont été tout récemment connus et promettent une approche plus sémantique que syntaxique des outils de production de logiciel.

5.5 Portabilité de l'éditeur.

Une des exigeances qu'on est en droit d'avoir de tout programme est sa portabilité. Elle est particulièrement difficile à satisfaire pour un programme système comme un éditeur, qui va à un titre ou un autre dépendre des caractéristiques du système d'exploitation. Nous pensons avoir montré que c'était possible quand le programme est écrit dans un langage de haut-niveau comme PASCAL, même s'il n'existe pas de standard de ce langage. Voici les quelques transports qui ont été effectués ainsi que les raisons qui ont poussés les utilisateurs à posséder EDIS.

- EDIS a été transporté sur RSX/11M (système d'exploitation existant sur les ordinateurs DIGITAL). Il est utilisé par un groupe d'ingénieurs chez HASLER AG, essentiellement parce qu'ils s'étaient vus imposer le langage PORTAL et ne le connaisaient pas. L'éditeur a servi à renforcer l'apprentissage de PORTAL.

- EDIS a été transporté sur UCSD, à l'INRIA, comme outil de développement pour le projet KAYAK. Il y remplace l'éditeur habituel se trouvant sur UCSD, les commandes étant très semblables.

- EDIS a été transporté sur VAX à l'EPFL et devrait être adapté pour certaines applications de traitement de texte de l'Ecole. Il a été choisi essentiellement parce qu'il s'agit d'un éditeur dont le source est accessible, qui est écrit en PASCAL et donc "facilement" transportable et adaptable.

- EDIS a été transporté sur SMACKY-8 (micro-procésseur 68000) sur UCSD, pour des enseignements variés au niveau

secondaire.

5.6 Extensions, perspectives.

Une extension de l'éditeur actuel dans le sens de l'extensibilité par l'introduction de macros-commandes serait tout à fait envisageable, mais nous ne pensons pas qu'il vaille la peine d'investir encore dans cette version de l'éditeur, pour l'améliorer ou pour l'adapter à de nouvelles technologies. Nous pensons au contraire qu'il faudrait, s'inspirant d'un jeu de commandes cohérent d'une part, d'un jeu de primitives d'autre part, redéfinir un produit entièrement nouveau. On trouvera dans [MEYROWITZ 82]. un excellent survol des principales caractéristiques pour un éditeur.

Ce produit devrait comporter des fonctions de base de navigation, de manipulation de caractères, de symboles, de tampons, une analyse lexicale et une analyse syntaxique gouvernées par table, des accès au système d'exploitation pour l'accès aux fichiers, un affichage indépendant et optimal, peu influencé par la technologie de l'écran, un interface d'entrée permettant d'accueillir des nouvelles technologies (une souris, des claviers spécialisés). Si l'éditeur actuel a été concu dans l'esprit de commandes orthogonales et de modules indépendants, du moins sur le plan conceptuel, son implantation concrète soufre en revanche d'imperfections au niveau du *codage modulaire*.

5.7 Conclusion

Nous proposons dans ce travail un éditeur pour la programmation permettant la vérification de la syntaxe de programmes écrits en différents langages. L'utilisation large de ce produit a montré son adéquation à une attente à l'égard de nouveaux produits,

commodes, agréables à l'utilisateur. Son implantation sur un micro-ordinateur possédant peu de mémoire a démontré qu'il était possible d'offrir un outil de relativement haut niveau pour des postes de travail individuels. L'indépendance de l'éditeur vis-à-vis du langage de programmation pour la vérification syntaxique a permis d'acceuillir rapidement les nouveaux langages ADA, SARTEX et MODULA-2. Cela devrait nous permettre de nous familiariser avec les nouveaux concepts véhiculés par ces langages, avant même de disposer de compilateurs. Il est ainsi possible de ***programmer*** sans compilateur ou sans disposer de la machine cible.

6. BIBLIOGRAPHIE

[AHO 72]
Aho A.V., J.D. Ullman, "The Theory of Parsing, Translation and Compiling", Prentice-Hall, Englewood Cliffs 1972.

[BACKHOUSE 79]
Backhouse R.C., "Syntax of Programming Languages Theory and practice", Prentice-Hall International, London 1979.

[BOEM 78]
Boem B.W. et Al., "Characteristics of software quality", TRW series, North Holland, 1978.

[BOOCH 83]
Booch Grady, "Software engineering with ADA", Benjamin/Cummings, Menlo Park 1983.

[BROOKS 75]
Brooks F.P., "The mythical man-month", Addison Wesley, Reading, Mass., 1975.

[DIJKSTRA 76]
Dijkstra E.W., "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, 1976.

[DONZEAU 75]
Donzeau-Gouge V., Huet G., Kahn G., Lang B., Lévy J-J., "A Structure Oriented Program editor: a First Step Towards assisted Programming", *International Computing Symposium 1975, Antibes, France*, North Holland.

[DONZEAU 81]
Donzeau-Gouge V., Huet G., Kahn G., Lang B., "Environnement de programmation MENTOR: présent et avenir", *Actes des*

troisièmes journées francophones sur l'informatique, Genève 13-14 janvier 1981.

[FREI 78]
Frei H.-P., Weller D.L and Williams R., "A Graphics-Based Programming-Support System", *Computer Graphics 12,3 August 1978.*

[GREENBERG 80]
Greenberg B. S., "Multics EMACS: An Experiment in Computer Interaction", *in Fourth Annual Honeywell Software Conference, March 1980.*

[KAHN 83]
Kahn G., Lang B., Mélèse B., Morcos E., "METAL, a formalism to specify formalisms", *in "Les éditeurs dirigés par la syntaxe", Notes de cours donné à Aussois, (Savoie), INRIA, Rocquencourt 1983.*

[LEDGARD 79]
Ledgard H.F., Nagin P.A., Heuras J.F., "PASCAL with Style" Hayden, Rochelle Park NJ 1979.

[LIVERCY 78]
Livercy C., "Théorie des programmes: schémas, preuves, sémantique", Dunod, Bordas Paris 1978.

[MELESE 81]
Mélèse B., MENTOR, "L'Environnement PASCAL", INRIA, *Technical Report N5, October 1981.*

[MEYROWITZ 82]
Meyrowitz N., van Dam A., "Interactive Editing Systems", *Computing Surveys, V14, N3, September 1982.*

[MORRIS 81]
Morris J.M., Schwartz M.D., "The Design of a Language-Directed Editor for Block-Structured Languages", *Proceedings of the ACM SIGPLAN/SIGOA Conference on Text Manipulation, Portland Oregon June 1981.*

[NASSI 73]

Nassi I., Schneidermann B., "Flowchart Techniques for Structured Programming", *ACM SIGPLAN Notices V8,N8 August 1973.*

[NOTKIN 79]

Notkin D.S., Habermann A.N. "Software Development Environement Issues as Related to ADA", Carnegie-Mellon University, Pittsburg PA, 1979.

[REPS 83]

Reps T., Teitelbaum T., Demers A., "Incremental Context-Dependant Analysis for Context-Based Editors", to appear in TOPLAS, *paru dans "Les éditeurs dirigés par la syntaxe", Notes de cours donné à Aussois, (Savoie), INRIA, Rocquencourt 1983.*

[SHANI 83]

Shani U., "Should Program Editors not Abandon Text Oriented Commands ?", *SIGPLAN Notices V18, N1, January 1983.*

[SCHNEIDER 81]

Schneider G.M., S.C.Bruell, "Advanced programming and Problem solving with PASCAL" John Wiley, 1981.

[TEITELBAUM 80]

Teitelbaum T., "The Cornell Program Synthetiser: A Tutorial Introduction", *Report TR 79-381, Cornell University, January 1980.*

[TEITELBAUM 81]

Teiltelbaum T., Reps T., Horwitz S., "The Why and wherefore of the Cornell Program Synthesizer", *Proceedings of the ACM SIGPLAN/SIGOA Conference on Text Manipulation, Portland Oregon, June 1981.*

[WATERS 82]

Waters R.C., "Program Editors Should not Abandon Text Oriented Commands", *SIGPLAN Notices, V17, N7, July 1982.*

[WEINBERG 71]
Weinberg G.M., "The psychology of computer programming", Van Nostrand, N.Y. 1971.

[WIRTH 77]
Wirth N., "Compilerbau", Teubner Studientaschenbücher, Informatik, Stuttgart 1977.

7 LE CONSTRUCTEUR DE GRAMMAIRE PARSE POUR EDIS. LE LANGAGE GRAMOL.

7.1 Introduction

Dans l'éditeur de symboles EDIS, la vérification syntaxique, ainsi que la saisie abrégée des mots-clefs sont gouvernées par des tables, chargées en mémoire.

Le fichier binaire décrivant la grammaire d'un langage de programmation s'appelle sur UCSD SYSTEM.E.langage (par exemple SYSTEM.E.PASCAL) et doit se trouver sur la disquette système. Les fichiers pour les autres langages ont des noms analogues; sous VMS et RSX ils s'appellent EGRPAS.DAT et doivent se trouver dans le directoire réservé à cet effet.

Ces fichiers sont produits par le programme **parse**, qui vérifie la syntaxe des données. Le programme **parse** reçoit une description de la grammaire ainsi que d'autres paramètres sur un fichier source. Nous décrirons dans la section suivante la syntaxe de cette description. Le fichier source comprend plusieurs sections.

1. La grammaire (syntaxe du langage en format EBNFA).
2. La liste des axiomes
3. Les valeurs des symboles lexicaux fournis par l'analyseur lexical et les mots-clefs du langage.
4. Les identificateurs prédéfinis (nécessaires à l'analyse sémantique partielle).
5. L'attribution des mots-clefs aux touches du clavier pour la

saisie abrégée et l'indentation.

Les sections sont séparées par des symboles "$".

7.2 GRAMOL: le langage de description de la grammaire. Format des différentes sections.

7.2.1 Symboles lexicaux.

1. Des commentaires de deux sortes peuvent être insérés partout:

   - tout texte qui suit un ";" jusqu'à la fin de la ligne est un commentaire.
   - les commentaires parenthésés sont mis entre parenthèses *pointues* "<>". Ils peuvent être emboités.

2. Un identificateur suit la syntaxe habituelle, débutant avec une lettre, suivi de lettres, chiffres et caractère souligné. Les identificateurs représentent soit des *symboles terminaux* du langage:, dans ce cas il sont entourés d'un apostrophe, soit des *symboles non-terminaux*. Un identificateur peut être *enrichi* de deux attributs:
   - un numéro d'erreur, qui est appondu par une virgule
   - un numéro d'action sémantique, qui est appondu par un signe %.

3. Les opérateurs suivants du meta-langage permettent des constructions de sélection "[ ]", de répétition "[* *]", d'itération "[+ +]", d'alternative "|", de regroupement "( )", les voici: "$", "*=", "=", "( )", "[ ]", "[* *]", "[+ +]", "[ ]", "|" .

7.2.2 Syntaxe de GRAMOL

La première section décrit la grammaire du langage dans un formalisme EBNF (Extended Backus Naur Form). C'est une liste de règles qui définissent chacune la dérivation d'un élément non-terminal.

La deuxième section décrit le vocabulaire utilisé dans la première section. La description du *vocabulaire* permet d'identifier l'ensemble des terminaux utilisés dans la description de la grammaire et d'attribuer les valeurs que devra par ailleurs fournir l'analyseur lexical; les terminaux qui sont des mots-clefs réservés apparaissent dans cette liste *sans apostrophes*. Cette liste est introduite dans la grammaire et utilisée par l'analyseur lexical de l'éditeur. On énumère d'abord les terminaux qui correspondent à des symboles spéciaux ou composés.

La troisième section énumère la liste des identificateurs prédéfinis.

La quatrième section définit l'indentation:

Deux possibilités existent pour piloter l'indentation:

- on donne la liste des mots clefs, qui, s'ils apparaissent en début d'une ligne, provoquent une identation à la saisie d'une fin de ligne (exemple: **begin**).

- on donne la liste des touches, qui, si elles sont utiliséees pour la saisie abrégée en début de ligne, provoque une desindentation (exemple **until**).

Ces deux listes sont données en spécifiant les touches de saisie abrégée.

La cinquième section spécifie les touches pour la saisie abrégée:

On donne la liste des abréviations des 26 lettres. Chaque mot-clef est donné entre apostrophes. Cela permet d'introduire des espaces,

qui seront insérés à la saisie. La taille des caractères est significative.

Pour illustrer cette spécification voici une donnée complète d'une grammaire pour un langage hypothétique DOMULA:

```
;06_01_84 grammaire DOMULA
relop          =    'EQUAL'
                |   'NEQUAL'
                |   'SUBRELOP'.
typ            =    'INT'
                |   'ID%2'.
compunit,40    *=   'MODULE' 'ID,2%3' 'SEMI,14'
                    block,32 'ID,2%4' 'PERIOD,20' .

block          *=   [* declarat *]
                    [    'BEGIN'
                         [ stmt+seq ] ]
                         'END,13' <block> .
declarat       =    'CONST'
                         [+ 'ID%1' 'EQUAL,16' 'INT,10'
                              'SEMI,14' +]
                |

                    'TYPE'
                         [+ 'ID%1' 'EQUAL,16' typ,10
                         'SEMI,14' +]
                |
                    'VAR'
                         [+ 'ID%1' [* 'COMMA' 'ID,2%1' *]
                              'COLON,5'
                              typ,10 'SEMI,14' +]
                |   'PROCEDUR'<E> 'ID,2%3' 'SEMI,14'
                    block,32 'ID,2%4' 'SEMI,14'.
fac            =    'INT'
                |   'NOT' fac,58
                |   'LPAREN' expr,58 'RPAREN,4'.
```

```
term          =   fac [* 'MULOP' fac,58 *].
smp+expr      =   'ADDOP' term,58 [* 'ADDOP' term,58 *]
              |   term [* 'ADDOP' term,58 *].
expr          =   smp+expr [ relop smp+expr,58 ].
statement,30  *=  stmt+alt.
stmt+seq      =   stmt+alt [* 'SEMI' [ stmt+alt ] *].
stmt+alt      =   'ID%2' [ 'BECOMES' expr,58 ]
              |   if+stmt
              |   while+stmt
              |   for+stmt
              |   rep+stmt.
if+stmt       =   'IF' expr,58 'THEN,52'
                  [ stmt+seq ]
              [* 'ELSIF' expr,58 'THEN,52'
                  [ stmt+seq ] *]
              ['ELSE'
                  [ stmt+seq ]]
                  'END,13' <if>.
while+stmt    =   'WHILE' expr,58 'DO,54'
                      [ stmt+seq ]
                  'END,13' <while>.
rep+stmt      =   'REPEAT' [ stmt+seq ] 'UNTIL,53'
                  expr,58.
for+stmt      =   'FOR' 'ID,2%2' 'BECOMES,51' expr,58
                  'TO,55' expr,58 'DO,54'
                      [ stmt+seq ]
                  'END,13' <for>.
$
```

;axiome_principal, version :

```
compunit,5
```

;autres axiomes non référencés:

```
statement
$
```

;ordre des symboles terminaux comme dans l'éditeur :

;1) énumeration fixe :

```
'ID' = 0 'COMMA' = 1 'COLON' = 2 'SEMI' = 3 'LPAREN' = 4
```

```
'RPAREN' = 5 'MULOP' AND MOD DIV = 6 'ADDOP' OR = 7
'SUBRELOP' = 8 'BECOMES' = 9 'PERIOD' = 13 'INT' = 14
'EQUAL' = 21 'NEQUAL' = 22 'BAR' = 24
```

;2) énumeration libre :

```
CONST + TYPE + VAR + PROCEDUR<E> + BEGIN + END + IF +
THEN + ELSE + ELSIF + REPEAT + UNTIL + WHILE + FOR + TO
+ DO + MODULE + NOT
$
```

;liste des prédéfinis :

```
ABS BOOLEAN CARDINAL CHAR FALSE FLOAT HALT ODD
TRUE TRUNC
$
```

;indentation (total droite-gauche = 23 lettres au maximum) :
;a droite :

```
BCFILWQRYSV
```

;a gauche :

```
EUT
$
```

;mots-clefs saisie abrégée:

```
'abs ' 'begin ' 'const ' D: 'do' 'end ' F: 'for '
G: 'BOOLEAN' H: 'CHAR' 'if ' J: 'INTEGER' K: 'CARDINAL'
L: 'else' M: ':= ' N: 'then ' O: 'to ' 'procedure '
Q: 'FLOAT' R: 'repeat' S: 'elsif' T: 'until '
U: 'module ' 'var ' 'while ' X: '*)' Y: 'type'
Z: '(* ' $: '(*$L*)'
$
```

Voici le formalisme de GRAMOL: (qui n'est pas spécifié exactement en EBNF, puisque les terminaux sont présentés entre guillemets).

```
GRAMMAIRE          =    [+ REGLE +]
                        "$" AXIOMES
```

```
                    "$" LEXIQUE
                    "$" PREDEFINIS
                    "$" INDENTATION
                    "$" TOUCHES
                    "$" .
REGLE           =   NONTERMINAL ["*=" | "="]
                    EXPRESSION "." .
EXPRESSION      =   [* TERME *] .
TERME           =   FACTEUR [* "|" FACTEUR *] .
FACTEUR         =   TERMINAL
                |   NONTERMINAL
                |   "(" EXPRESSION ")".
                |   "[" EXPRESSION "]"
                |   "[*" EXPRESSION "*]"
                |   "[+" EXPRESSION "+]"
                |   "[+" EXPRESSION "$"
                        EXPRESSION "+]"
                |   "[ " EXPRESSION "$" EXPRESSION
                    "$" EXPRESSION " ]" .
NONTERMINAL     =   IDENTIFICATEUR
                |   IDENTIFICATEUR [ ERREUR ]
                                   [ ATTRIBUT ] .
TERMINAL        =   "'" IDENTIFICATEUR "'"
                |   "'" IDENTIFICATEUR [ ERREUR ]
                    [ ATTRIBUT ] "'".
ATTRIBUT        =   "%" NOMBRE ENTIER .
ERREUR          =   "," NOMBRE ENTIER .
AXIOMES         =   NONTERMINAL; axiome principal, version
                    [* NONTERMINAL *].; autres axiomes

LEXIQUE         =   PARTIE NUMEROTEE  PARTIE LIBRE .

PARTIE NUMEROTEE =  [+ TERMINAL [* NONTERMINAL *] "="
                    NOMBRE ENTIER +] .
PARTIE LIBRE    =   [* [+ NONTERMINAL $ "+" +] *] .
PREDEFINIS      =   [+ NONTERMINAL +] .
INDENTATION     =   26 lettres majuscules .
TOUCHES         =   27 * TERMINAL.
```

7.2.3 Sémantique.

Les non-terminaux de la syntaxe suivent les règles de cohérence habituelles.

- définition unique
- pas de non-terminaux non définis.

Les terminaux qui apparaissent en partie gauche d'une règle et sont définis par "*=" sont les axiomes. Leur libellé est utilisé pour la confection de la ligne de rappel dans la fonction Test de l'éditeur. Ils doivent par conséquent tous commencer par une lettre distincte, puisque celle-ci servira comme touche de sélection pour le choix de l'axiome. Le nombre d'axiomes dans une grammaire est limité à huit. Les axiomes sont les seuls non-terminaux qui sont munis d'attributs "ERREUR" en partie gauche d'une règle. Les numéros d'actions sémantiques correspondent aux actions suivantes, dont une au plus est exécutée par l'analyseur lors de chaque réduction:

1: Mettre l'identificateur sur le tas, s'il ne s'y trouve déjà.

2: Mettre l'identificateur sur le tas, s'il ne s'y trouve déjà, et le marquer.

3: Mettre l'identificateur sur le tas, s'il ne s'y trouve déjà, et empiler une référence à cet identificateur.

4: Mettre l'identificateur sur le tas, s'il n'y était pas, et le noter. Dépiler la référence et vérifier qu'il s'agit de la même référence, sinon signaler une erreur.

5: Désempiler une référence.

7.2.4 Règles

Il convient de respecter un certain nombre de règles dans l'écriture d'une grammaire. Nous rappelons ces règles ici:

1) La grammaire doit être mise sous forme LL(1). A cette fin une factorisation d'eléments initiaux s'avère nécessaire,

2) Parmi les débutants d'un terme d'une alternative un seul facteur peut être annulable; dans ce cas ce terme doit être placé en dernier,

3) Un facteur atomique non premier sans alternative doit avoir un message d'erreur, sauf s'il est annulable,

4) Un facteur atomique premier sans alternative n'ose pas avoir de message d'erreur, sauf s'il est annulable,

5) Aucune expression annulable ne doit être placée à l'interieur des constructeurs "[ ]" ou "[° °]".

EDIS : UN EDITEUR DE SYMBOLES

MANUEL DE L'UTILISATEUR **VERSION 2.0d 23.11.83**

1. Concepts de base

Cette introduction présente les concepts nécessaires à la compréhension et à l'utilisation de l'Editeur de Symboles **EDIS**.

La seconde partie de cette notice donne une description détaillée de chaque commande, ainsi que des exemples.

1-1 La 'fenêtre' dans un fichier

L'Editeur de Symboles est un programme prévu pour visualiser et modifier un fichier à l'aide d'un terminal à écran. L'éditeur montre en permanence le contenu du fichier sur l'*écran*, qui peut être considéré comme une *fenêtre* ouverte sur le texte du fichier. Si le texte est trop long pour l'écran, seule une partie des lignes est affichée; cependant tout le fichier est accessible aux commandes de l'éditeur. Toute commande de l'éditeur concernant une partie du fichier qui n'est pas affichée provoque un déplacement vertical de la fenêtre pour faire apparaître la partie concernée.

1-2 Le curseur

Le curseur est un rectangle lumineux qui indique un caractère dans le fichier et peut être déplacé en n'importe quel endroit du texte. La fenêtre montre à chaque instant une partie du fichier contenant le curseur. Pour voir une autre partie du fichier, il suffit de déplacer le curseur par des commandes adéquates.

Les actions ont lieu à proximité du curseur. Un grand nombre de commandes sont disponibles: adjonction, modification ou suppression, etc. Certaines agissent sur une portion de texte plus grande que la fenêtre. Il n'est cependant jamais nécessaire d'opérer sur des parties invisibles.

1-3 Le facteur de répétition

La plupart des commandes autorisent un facteur de répétition. Il s'agit d'un nombre frappé immédiatement avant la commande. La commande est alors exécutée ce nombre de fois. Par exemple: **20<low>** déplace le curseur de vingt lignes vers le bas. Si le facteur est omis, il vaut simplement 1. En frappant **/** comme préfixe d'une commande on obtient un facteur *infini*; la commande est exécutée

autant de fois que possible.

1-4 Les mouvements du curseur

Pour éditer un fichier il est nécessaire de déplacer le curseur. Pour cela les quatre touches marquées de flèches (*arrows*) sur le clavier provoquent les quatre principaux mouvements. Elles sont appelées **<right>**,**<left>**,**<low>**,**<up>**. Pour les grands mouvements, il existe Begin (début du texte) et End (fin du texte). Ce sont des touches souvent utilisées. Pour faire défiler le texte on peut laisser le doigt sur la touche **<low>**, mais Page (affiche la page suivante) est plus commode, Opposite revient à la page précédente.

1-5 Les commandes de modification

Les mouvements ne modifient pas le texte et sont réversibles; les autres commandes sont plus *dangereuses*. La plupart d'entre elles commencent par une touche bien déterminée, suivie d'une séquence à choix, et se termine par **<etx>** ou **<esc>**. **<etx>** signifie fin de commande, tandis que **<esc>** annulle la commande. Pour voir l'attribution de ces fonctions à certaines touches particulières, voir ... Lorsque l'éditeur informe qu'une commande a été effectuée ou signale une erreur, une quittance doit être donnée par **<sp>** (touche d'espacement).

Exemple :

```
Trouve 192 fois. <sp> pour continuer
```

1-6 Rappels

La première ligne de l'écran est une zone spéciale. Le reste de l'écran contient la fenêtre. Cette première ligne permet avant tout de savoir qu'on est dans l'éditeur plutôt que dans une autre partie du système. On a ainsi un point de repère. Elle rappelle le choix des commandes disponibles, ainsi que le langage pour lequel EDIS est réglé. Voici la *ligne de rappel* (prompt-line) au moment d'entrer dans l'Editeur.

```
>EdiS(PASCAL):Ins Del Xch Adj Cpy Get Find Rep Beg End Pag Write Quit ? [2.0d]
```

Remarques

Afin de rappeler la signification de chaque commande, un ou deux mots d'explication (en minuscules) accompagnent la commande proprement dite (les ':' font partie du commentaire)

Adj	pour A

<ret>: fichier de travail	pour <ret>
Ins	pour I

Parfois la liste des commandes disponibles est tronquée. Pour voir les autres commandes frapper "?" ; la ligne de rappel devient alors :

```
>EdiS(PASCAL): Look Make Opposite Set Test Verfy Zap <insb> <delc> ?
```

1-7 Quelques conventions de notation

Dans les messages de l'éditeur et dans les menus on utilise des conventions qui sont expliquées ici.

1-71 Notation pour désigner les touches.

Les commandes de modification et de navigation dans le texte utilisent beaucoup de touches du clavier. Les touches correspondant à l'alphabet habituel seront désignées, dans le mode d'emploi, par les majuscules de l'alphabet. Les autres touches portent en général une inscription; pour les désigner on emploira une abréviation de 2 ou 3 lettres en minuscule placées entre les crochets < et >. Les commandes sont signalées en **gras**. Par exemple la touche marquée

- **<ret>** représente la touche RETURN (fin de ligne),
- **<bs>** représente la touche BACKSPACE (recul d'un caractère),
- **<etx>** désigne la touche marquée ETX (sur les TERAK)
- **<sp>** désigne la barre d'espacement (parfois notée SPACE).

Quelques fonctions sont réalisées à l'aide de touches peu connues comme DC1 (control Q) ou DC3 (control S); de plus leur attribution n'est pas fixe d'une installation à l'autre. Elles sont signalées par leur fonction plutôt que par la touche. Ainsi:

- **<col1>** désigne la touche qui permet de déplacer le curseur à la marge gauche de la ligne
- **<insb>** insère un espace (blanc) à droite du curseur
- **<delc>** detruit le caractère désigné par le curseur
- **<invc>** désigne la touche qui permet d'inverser la taille du caractère

Il est plus parlant de désigner une commande par un nom, même abrégé, plutôt que par une lettre; on écrira par exemple **Insert** ou **Ins** au lieu de **I**.

1-72 Notation pour le choix de plusieurs commandes

Plusieurs lettres, chiffres et signes de ponctuation consécutifs peuvent apparaître dans une commande; on désigne une telle suite par le mot **text**. Les touches **<ret>**,

<bs> sont admises dans ce contexte avec la signification habituelle. De même pour **<shift>** qui modifie l'effet d'une lettre (majuscule).

Pour les commandes de l'éditeur, les minuscules et les majuscules peuvent être indifféremment utilisées. Plusieurs commandes à choix, sont séparées par des virgules et lorsque plusieurs touches doivent être frappées en séquence pour former une commande, elles sont écrites l'une après l'autre (sans virgule de séparation).

1-8 La direction de la navigation

1-81 Sens du déplacement

Le premier caractère de la ligne de rappel indique la direction, c.à.d. le mode de déplacement courant; il existe deux sens: > (*avance*), < (*recule*). Le sens est affecté par une commande > ou < ; on peut aussi frapper syr les touches "," et "." ou encore "+" et "-". Les commandes (Find, Replace) et mouvements (Page, Opposite, **<ret>**, **<tab>**) du curseur sont modifiées par la direction. Si la direction est *avance*, les commandes opèrent vers la droite et en descendant c.à.d. depuis la position actuelle du curseur vers la fin du fichier.

1-82 Les mouvements (moves)

On peut distinguer les commandes de navigation: dans tout le fichier, par ligne, par symbole et par caractère.

Les mouvements les plus grands sont Begin et End : aller au début du fichier, respectivement à la fin du fichier,

Page: déplacer le curseur d'une hauteur d'écran en avant ou en arrière selon la direction courante.

Opposite: déplacer le curseur d'une hauteur d'écran dans la direction opposée à la direction courante.

L'éditeur peut mémoriser *neuf* positions grâce à la commande *. La commande = est un mouvement qui ramène le curseur à une position mémorisée. En préfixant par un chiffre les commandes *, respectivement =, on sélectionne une des neuf positions. L'option par défaut est 1. D'une session à l'autre le symbole sur lequel le curseur était placé au moment du sauvetage est *enregistré* (sur UCSD); on peut donc s'y rendre avec la commande =.

L'éditeur reconnaît les symboles des langages de programmation PASCAL, PORTAL, MODULA-2, ADA, SARTEX et quelques autres. Un symbole est alors soit un identificateur, soit une chaîne, soit un opérateur tel que '<=' ou ':=', soit un

commentaire. Aucun symbole ne peut enjamber une fin de ligne; on convient que la fin de ligne est elle-même un symbole.

Les touches **<left>** et **<right>** déplacent le curseur d'un symbole. Les touches **<sp>** et **<bs>** déplacent le curseur d'un caractère.

1-9 Les langages

L'éditeur peut être réglé pour la saisie de PROSE (pour la langue naturelle), de PASCAL, PORTAL, MODULA-2, PL/M86, ADA et SARTEX. La frappe d'une lettre avec **<shift>** provoque l'apparition de mots-clefs ou identificateurs fréquents associés au langage, sauf en PROSE.

En PASCAL voici l'affectation des mots-clefs aux touches :

A	ARRAY	B	BEGIN	C	CONST	D	DO
E	END	F	FOR	G	boolean	H	char
I	IF	J	integer	K	CASE	L	ELSE
M	:=	N	THEN	O	OF	P	PROCEDURE
Q	FUNCTION	R	REPEAT	S	RECORD	T	TYPE
U	UNTIL	V	VAR	W	WHILE	X	•)
Y	WITH	Z	(•				

En PORTAL voici l'affectation des mots-clefs aux touches :

A	DEFINES	B	CODE	C	CONST	D	DO
E	END	F	PROCESS	G	boolean	H	char
I	IF	J	integer	K	CASE	L	ELSE
M	MODULE	N	THEN	O	OF	P	PROCEDURE
Q	LOOP	R	RESULT	S	RECORD	T	TYPE
U	USE	V	VAR	W	WHILE	X	•)
Y	WITH	Z	(•				

En ADA voici l'affectation des mots-clefs aux touches :

A	ARRAY	B	BEGIN	C	CONSTANT	D	IS
E	END	F	FOR	G	boolean	H	character
I	IF	J	integer	K	CASE	L	ELSE
M	:=	N	THEN	O	LOOP	P	PROCEDURE
Q	FUNCTION	R	RANGE	S	RECORD	T	TYPE
U	RETURN	V	WHEN	W	WHILE	X	PACKAGE
Y	ELSIF	Z	SUBTYPE				

En MODULA voici l'affectation des mots-clefs aux touches :

A	ARRAY	B	BEGIN	C	CONST	D	IMPORT
E	END	F	ELSIF	G	BOOLEAN	H	CHAR
I	IF	J	INTEGER	K	CASE	L	ELSE
M	:=	N	THEN	O	OF	P	PROCEDURE
Q	CARDINAL	R	RETURN	S	RECORD	T	TYPE
U	FROM	V	VAR	W	WHILE	X	*)
Y	WITH	Z	(*				

Dans cet éditeur, en PASCAL et en PL/M86, les commentaires qui commencent avec (* et se terminent par *) doivent tenir sur une ligne. L'éditeur le vérifie (à la sortie) et coupe automatiquement les commentaires portant sur plusieurs lignes en insérant des débuts et des fins de commentaires.

1-10 La Vérification syntaxique

La commande Test permet d'effectuer une vérification syntaxique suffisante pour détecter rapidement la majorité des erreurs courantes. On gagne ainsi du temps en évitant des compilations inutiles. Mais cela ne veut pas dire que le programme est exempt d'erreurs. Le compilateur détectera des erreurs syntaxiques moins évidentes, et finalement, même le programme compilé et exécutable peut comporter des fautes. C'est alors au programmeur qu'il incombe de prévoir les tests nécessaires. A la fin de la vérification syntaxique on peut demander la liste des identificateurs non déclarés. Ils apparaissent tronqués à huit caractères.

1-11 Les tampons

Quatre tampons indépendants sont définis au cours d'une session d'édition.

1-111 Tampon de copie.

Le texte inséré par la commande Insert ou détruit par la commande Delete est conservé dans le *tampon de copie*; ce texte est disponible et peut être inséré à l'aide de la commande Copy.

1-112 Recherche et substitution

La chaîne cible de la commande **Find** et la chaîne de substitution de la commande **Replace** sont conservées; elles peuvent être invoquées par **<etx>** lors d'une nouvelle commande.

1-113 Direction (<,>)

La direction, visible comme premier caractère de la ligne de rappel détermine la direction de recherche et d'avance par **Page**.

1-114 *, =

La commande * mémorise la position du curseur, la commande = déplace le curseur à cette position. En préfixant ces commandes d'un chiffre on peut mémoriser *neuf* positions différentes.

2. COMMANDES DE L'EDITEUR DE SYMBOLES

2-1 Commandes de navigation

Nous présentons ici les commandes de navigation avec leur signification.

B	Begin	aller au début du fichier
E	End	aller à la fin du fichier
P	Page	avancer, ou reculer d'une page, (selon la direction); le curseur va au début d'une ligne.
O	Opposite	reculer, ou avancer d'une page, (selon la direction)
H	Home	placer le curseur sur le premier symbole affiché à l'écran
=		placer le curseur sur une des marques fixées par la commande *.
F	Find	est aussi une commande de mouvement, mais sera traitée en relation avec la commande **R** (remplacement) dans la section 2.2.7
<etx>		est une commande abrégée pour la recherche. Elle a le même sens que la commande **F**
<right>		avancer d'un symbole
<left>		reculer d'un symbole
<low>		descendre dans la même colonne sans égard au texte
<up>		monter dans la même colonne sans égard au texte
<ret>		avancer le curseur au début de la prochaine ligne ou reculer le curseur au début de la ligne précédente, (selon la direction)
<tab>		avancer le curseur à la fin de la ligne courante (ou suivante) ou reculer le curseur à la fin de la ligne précédente, (selon la direction)
<sp>		avancer d'un caractère
<bs>		reculer d'un caractère

2-2 Commandes de modification de texte

2-21 Insert.

On entre dans ce mode en frappant I. La ligne suivante apparaît alors:

```
>Insert: text,<del>,<col1>,<tab>,<left>,<right> [<etx>,<esc>]
```

Cette commande permet l'insertion de texte. Les caractères apparaissent immédiatement à gauche du curseur. En cas d'erreur de frappe, utiliser la touche **<bs>**. Pour supprimer la dernière ligne, on utilise la touche **<del>** et pour recommencer la ligne courante tout à gauche, on utilise la touche **<col1>** (colonne 1).

En frappant une lettre majuscule un mot-clef apparaît, à moins que le langage ne soit PROSE; les mots-clefs associés aux lettres dépendent du langage.

L'éditeur se souvient s'il convertit les minuscules en majuscules. Pour changer d'état frappez **<dc2>**.

<ret>	(retour de chariot) le curseur vient se placer sous le début du texte de la ligne courante avec une certaine indentation. Cette indentation dépend du premier mot de la ligne précédente. La valeur de l'incrément peut être modifiée à l'aide de la commande Set.
<left>	diminue l'indentation de la ligne courante
<right>	augmente l'indentation de la ligne courante
<tab>	insère un certain nombre de blancs jusqu'à concurrence d'une colonne de tabulation standard.
<etx>	termine l'insertion (accepte)
<esc>	termine l'insertion (renonce).
<up>	en MODULA permet de se mettre en majuscule pour la prochaine lettre seulement.

Les parties du fichier qui avaient été séparées au début de l'insertion sont accolées à la partie insérée. Un espace est parfois inséré pour éviter une fusion inappropriée de deux symboles. La partie insérée reste disponible dans le tampon de copie.

Si le langage est PROSE, c'est-à-dire si l'on veut saisir du texte français, la commande Insert se comporte différemment dans les cinq cas suivants:

a) L'insertion de mots-clefs n'est plus possible; la touche **<shift>** (majuscule) prend son sens habituel.
b) L'indentation ***automatique*** n'a évidemment plus de sens.
c) Après une ligne blanche, l'indentation est fixée par la valeur de *Paramargin*

(*voir commande* Set 2.3.1).

d) L'insertion de lettres accentuées est possible par la frappe consécutive de deux touches.
"/e" pour é aigu "\a" pour à grave
"\e" pour è grave "^a" pour â circonflexe "^i" pour î circonflexe
"^e" pour ê circonflexe
"~e" pour ë tréma "~a" pour ä tréma
",c" pour ç cédille etc...

e) En fin de ligne, si un mot n'a pas assez de place pour être tapé entièrement, il est automatiquement reporté à la ligne suivante.

2-22 Delete

Pour détruire une partie du texte, déplacer d'abord le curseur au début de la partie à détruire, puis frapper la touche **D**. La ligne de rappel suivante apparaît:

>Delete: moves, ';', '<', '>' [<etx>,<esc>]

Au moment de frapper **D**, l'éditeur se souvient de cette position. Cette position est appelée le *point d'ancrage*. Quand le curseur est déplacé autour du point d'ancrage, tout le texte compris entre le point d'ancrage et le curseur disparaît. Lorsque le curseur se rapproche du point d'ancrage les caractères réapparaissent. Pour accepter la suppression, frapper **<etx>**, pour l'annuller frapper **<esc>**.

Pour sélectionner plusieurs lignes avancer à l'aide de **<ret>**. Pour sélectionner le contenu d'une ligne, qu'on veut insérer au milieu d'une autre ligne, utiliser **<tab>**, pour sélectionner une instruction, utiliser ';'.

On peut préfixer tous les mouvements de destruction par un facteur de répétition.

2-23 Zap

Cette commande détruit tout le texte compris entre le curseur et la marque fixée par la commande *. Si on veut supprimer plus de 80 caractères l'éditeur demande une confirmation. Le texte supprimé par Zap, ou par Delete est disponible dans le tampon de copie, sauf avertissement contraire.

2-24 Copy

Tout texte *inséré* ou *supprimé* est simultanément stocké dans le tampon de copie. Pour utiliser le contenu du tampon il suffit de frapper **C**. L'éditeur copie alors *immédiatement le tampon à la position courante du curseur*. Le fait de frapper **C** n'affecte en rien le tampon de copie. La commande de Copie peut être utilisée après une insertion afin de duplifier le texte autant de fois qu'on le désire. Le curseur est *alors placé en tête du texte copié*.

On utilise le plus souvent cette commande pour déplacer une partie du texte. Pour cela, détruire le texte à déplacer, positionner le curseur et copier dans sa nouvelle position le texte détruit. Pour duplifier un texte, détruire en terminant par **<esc>** et faire ensuite une copie.

Le contenu du tampon de copie est affecté par les commandes suivantes:

a) Destruction.

Qu'on accepte (**<etx>**) ou qu'on annule (**<esc>**) la destruction, la partie effacée est conservée dans le tampon de copie.

b) Insertion

Qu'on accepte (**<etx>**) ou qu'on annule (**<esc>**) l'insertion, la partie insérée est conservée dans le tampon de copie.

c) Zap

Le tampon est chargé avec la partie détruite, sauf avertissement contraire.

La taille du tampon de copie est limitée: elle dépend de la longueur du fichier en mémoire. Chaque fois que la taille d'une destruction proposée est plus grande que celle du tampon, l'éditeur le signale par l'avertissement:

ERREUR: Pas de place pour copier le rebut. Supprimer ? <etx>,<esc>

En répondant **<etx>**, la destruction a lieu et le texte est perdu.

2-25 Get (inclusion d'un fichier)

Pour insérer dans le fichier édité le contenu d'un autre fichier, frapper **G**et. Donner le nom du fichier (le suffixe '**.TEXT**' est pris par défaut sur UCSD, de même le suffixe '**.PAS**', '**.ADA**' est pris par défaut sur VAX/VMS, selon le profil de l'utilisateur). Le fichier spécifié est alors copié avant le curseur. L'éditeur avertit si le fichier ne peut pas être copié en entier. Comme à l'entrée dans l'éditeur, le **?** permet d'obtenir la liste des fichiers qu'on peut éditer.

2-26 Xchange.

Cette commande permet de modifier les caractères de la ligne courante. En tapant **X**, la ligne d'en-tête suivante apparaît:

>eXch: text,<arrows>,<tab>,<ret>,<insb>,<delc>,<bs>,<ctl1>,<invc> [<etx>,<esc>]

Au fur et à mesure qu'on frappe des caractères, le curseur se déplace à droite le

long de la ligne, les nouveaux caractères remplaçant les anciens.

<right>	laisse les anciens en place
<left>	revient sur la gauche
<bs>	restaure les anciens
<up>	valide la ligne courante et va sur la ligne précédente
<low>	valide la ligne courante et va sur la ligne suivante
<invc>	inverse les lettres majuscules en minuscules ou les lettres minuscules et accentuées en majuscules
<etx>	termine la commande (le changement est exécuté)
<esc>	termine la commande (l'ancienne ligne réapparaît).
<ret>	passe au début de la ligne suivante en validant la ligne courante.
<col1>	vient au tout début de la ligne.
<insb>	insère un blanc.
<delc>	supprime le caractère désigné par le curseur.

Attention:

Comme il est possible de déplacer le curseur n'importe où, on a tendance, une fois entré dans cette commande à oublier d'en sortir (au moyen de **<etx>**).

2-27 Find et Replace.

Ces deux commandes sont discutées ensemble car elles obéissent aux mêmes règles syntaxiques.

2-271 Introduction

La commande **Find** appartient logiquement aux commandes de déplacement; elle est exécutée en frappant la touche **F** (ou **<etx>** dans certains cas). La ligne d'en-tête apparaît alors :

```
>Find[1] Symb: text [<etx>,<esc>].
```

ou

```
>Find[1] Lit: text [<etx>,<esc>].
```

La commande **Replace** est obtenue en frappant **R**. La ligne d'en-tête suivante apparaît:

```
>Rep[1] Symb: text <etx> text [<etx>,<esc>].
```

ou

```
>Rep[1] Lit: text <etx> text [<etx>,<esc>].
```

2-272 Discussion générale.

a) facteur de répétition; directions

Le facteur de répétition (facultatif) doit être frappé immédiatement avant **Find** ou **Replace**. Il apparaît alors entre crochets après le mot **Find** ou **Repl**. Si le facteur / (infini) est utilisé, la recherche se termine à la dernière occurrence. Le facteur de remplacement est simplement une borne supérieure. La recherche se fait dans la direction fixée (>: avance, <: recule). Si aucun facteur de répétition n'est donné, la commande est exécutée une seule fois.

b) modes

Il y a deux modes pour diriger la recherche.

b.1) Le mode symboles

Dans le mode ***symboles***, majuscules et minuscules sont considérées comme équivalentes, d'autre part le mot ou la séquence de mots sont recherchés isolément dans le texte. L'éditeur considère un mot comme isolé s'il est entouré par un quelconque délimiteur. Le mode ***symboles*** ignore les espaces et les fins de ligne.

b.2) Le mode littéral

Dans le mode ***littéral***, l'éditeur cherche la prochaine occurrence de la chaîne dans le fichier.

Exemples:

Dans le texte "De la mer émerge un récif" la chaîne "mer" est trouvée deux fois en mode ***littéral***, la chaîne "MER" n'est pas trouvée, tandis qu'en mode ***symboles*** le mot "mer" est trouvé une fois; de même pour le mot "MER" (*considéré comme* symbole équivalent à "mer").

En mode ***symboles*** les textes "('?')" et "('?')" sont *considérés comme* équivalents

Le mode de recherche, (***symboles*** ou ***littéral***) est fixé par la commande **Set** et apparaît dans la ligne de rappel de la commande **Find** ou **Replace**. Il peut être inversé pour la durée de la commande en frappant **<esc>** tout de suite après **F** ou **R**.
Le mode ***littéral*** est utile pour éditer du texte, alors que le mode ***symboles*** convient mieux pour l'édition de programmes.

c) chaînes

Les commandes opèrent sur des chaînes. L'éditeur possède deux variables de chaînes associées aux commandes **Find** et **Replace**. La première variable est la suite de caractères qui est ***recherchée*** dans la commande **Find**, ***cherchée et remplacée*** dans la commande **Replace**. La seconde, appelée ***substitution***, est la chaîne qui est ***substituée***; c'est la suite de caractères qui remplacera chaque occurrence de la chaîne recherchée.

La saisie d'une chaîne est terminée par **<etx>**. L'éditeur le signale en affichant un point ".". La (les) chaîne(s) sont conservées (même d'un fichier à l'autre); **<etx>** tout seul veut simplement dire *l'ancienne chaîne*. Elle apparaît alors sur la ligne de rappel. Pour obtenir une chaîne de longueur nulle il faut frapper un caractère et le détruire avec **<bs>**.

Exemple:

/R*<etx><sp><bs><etx>
/

Effet: supprimer tous les caractères '*'.

L'utilisation de la touche majuscule dans les commandes **Find** et **Replace** produit un mot-clef comme dans la commande **Insert**. L'utilisation des accents n'a de sens qu'en mode *littéral*.

Une fois la commande de remplacement complètement frappée, l'éditeur positionne le curseur sur la première occurrence trouvée avec la ligne de rappel:

>Repl[15] <etx>,<esc>, '/':all, <ret>:next

Plusieurs réponses sont possibles:

<etx>	effectue le remplacement et va à la prochaîne occurrence si le facteur est supérieur à 1.
<esc>	annule le remplacement et retourne au niveau de l'éditeur.
<ret>	n'effectue pas le remplacement, mais va à la prochaine occurrence.
/	effectue tous les remplacements qui restent à faire, en nombre demandé.

Remarques:
les chaînes de recherche et de substitution sont mémorisées,

1) Il est donc possible de demander une nouvelle fois la recherche de la même chaîne simplement en tapant **F <etx>**

2) De plus, la seule commande **<etx>** est équivalente à **F <etx>**; c'est une commande de recherche abrégée

3) On peut de même utiliser les anciennes chaînes de remplacement et de recherche en tappant **R <etx> <etx>**

2-28 Adjust.

Cette commande permet de modifier l'indentation d'une ou de plusieurs lignes. En frappant **A** au niveau de l'éditeur, le curseur vient se placer au début de la ligne et la ligne de rappel suivante apparaît.

>Adjust: L(jst,R(jst,C(enter,<arrows>,<tab>,<ret> [<etx>,<esc>]

On peut alors utiliser les touches **L, R, C, <left>, <right>** ou **<tab>** (éventuellement précédée d'un facteur de répétition). *L'indentation* de la ligne (le nombre de blancs au début de la ligne) est modifiée. L'effet est immédiatement visible:

L	justifie la ligne sur la marge gauche
R	justifie la ligne sur la marge droite
C	centre le texte au milieu entre les deux marges
<ret>	justifie la ligne courante en l'alignant avec le début de la ligne précédente
<right>	déplace d'un caractère à droite toute la ligne
<left>	déplace d'un caractère à gauche toute la ligne
<low>	justifie la ligne suivante d'autant de caractères que la dernière ligne justifiée
<up>	justifie la ligne précédente d'autant de caractères que la dernière ligne justifiée
<tab>	déplace de huit caractères à droite toute la ligne

Lorsque la ligne est ajustée à la bonne position, frapper **<etx>** ou **<esc>** pour sortir de cette commande.

Pour ajuster plusieurs lignes, ajuster la première ou la dernière ligne puis frapper **<up>** ou **<low>** et finalement **<etx>** ou **<esc>** (accepté dans les deux cas).

2-29 Touches <delc> et <insb>

Pour faire de petites corrections il est fastidieux d'utiliser les commandes Delete et Insert. On utilisera de préférence les commandes suivantes. En appuyant sur la touche **<delc>**, on supprime le caractère qui se trouve à la position du curseur. En appuyant sur la touche **<insb>**, on introduit des blancs à droite du curseur respectivement, on fait réapparaitre les caractères supprimés par la touche **<delc>**. Le curseur reste en place et c'est toute la partie de la ligne située à droite

qui se rapproche. Tant qu'on n'a pas utilisé une autre touche, on peut récupérer les caractères disparus en tapant **<esc>**.

Attention :

Cette commande n'affiche aucune ligne de rappel, et ne demande pas de confirmation; de plus elle peut faire disparaître le tampon de copie.

2-3 Look, Set et Verify

2-31 Look et Set.

Ces commandes permettent d'afficher et de modifier un certain nombre de paramètres de l'éditeur, associés au fichier. En frappant la commande **Look**, le texte suivant apparaît sur l'écran:

```
>Look options: Set, <sp>
Fichier: MONFICHIER.TEXT Version: 10
Date de création: 1-10-81 Dernière modification: 10-10-81
7780 caractères utilisés (16 blocs), 6044 disponibles. (Tot. 30 blocs)

lO)g: Non        lanG)uage:PASCAL    syntaxe PASCAL 2.4
K)eywords => minuscules
I)ndentation=> 2
               F)ind par symboles
```

Pour modifier un de ces paramètres, frapper **Set**. On obtient alors:

```
>Set options: G,F,K,I, <sp>
```

ou (si l'on édite de la PROSE)

```
>Set options: G,F,L,R,P,M, <sp>
```

Pour modifier une option, frapper un des caractères affichés en majuscule. Voici la signification des différentes options.

G) Language:

Guide l'analyse lexicale. Valeurs possibles: PROSE, PASCAL, MODULA, PORTAL, ADA, SARTEX. En frappant sur la touche **G** une ou plusieurs fois, on choisit un autre langage; frapper **<sp>** une fois le bon choix effectué.

La rubrique Syntaxe n'a de sens que pour les langages différents de PROSE. Elle

indique si la syntaxe est chargée pour le language en question; (nécessaire pour la vérification syntaxique, cf. commande Test). Le fichier correspondant de nom **'SYSTEM.E.nom'** doit résider sur la disquette système (sur UCSD). Sur VAX/VMS le nom du fichier est **EGRnom.DAT** et il doit résider dans le directoire ... Les deux chiffres indiquent les numéros de format et de version de la syntaxe; ce sont des renseignements de service.

F) Find:

Indique le mode de recherche dans les commandes Find et Replace. Conseillé: par *symboles* pour les langages de programmation et *littéralement* pour PROSE.

K) Keywords:

Indique si la frappe en majuscule dans les commandes Insert et Find provoque l'apparition des mots-clefs, et si ceux-ci sont en majuscules ou en minuscules. Taper **K** pour faire le bon choix. Si on ne désire pas utiliser la saisie abrégée de mots-clefs, mettre sur *inhibés*

I) Indentation:

Indique la valeur relative de l'indentation automatique (marge gauche) en mode Insert. On peut modifier cette valeur en frappant I, suivi du nombre de caractères d'indentation voulu, terminé par **<sp>**. Conseillé: 2 caractères d'indentation pour PASCAL.

Les quatre paramètres suivants n'existent que pour PROSE, et n'apparaissent pas dans les autres langages.

L) Lmargin:

Indique la marge gauche, colonne de début de ligne et est significatif dans la commande Adjust. On peut modifier cette valeur en frappant **L**, suivi du nombre correspondant à la valeur voulue, terminé par **<sp>**.

R) Rmargin:

Indique la marge droite, dernière colonne de la ligne et est significatif dans la commande Adjust et dans le mode PROSE. On peut modifier cette valeur en frappant **R**, suivi du nombre correspondant à la valeur voulue, terminé par **<sp>**.

P) Paragraph:

Indique la marge au début d'un paragraphe, c'est-à-dire la colonne du début de la

première ligne après chaque ligne vide et est significatif dans le mode PROSE. On peut modifier cette valeur en frappant **P**, suivi du nombre correspondant à la *valeur voulue*, terminé par **<sp>**.

M) Mise en page:

Indique de quelle manière la commande **Margin** justifie un paragraphe: en drapeau, en insérant des blancs, ou pas du tout.

O) log:

Permet d'introduire un court texte à la sortie de l'éditeur qui sera inséré avec la date du jour à l'intérieur d'un commentaire, au début du fichier. On peut ainsi annoter brievement les fichiers en tenant un journal des modifications.

Enfin, le nombre de caractères utilisés indique la taille du fichier en mémoire. Le nombre de blocs est une valeur approximative de la taille du fichier sur disque. Le numéro de version augmente de un chaque fois qu'on sauvegarde le fichier après y avoir apporté une modification.

2-32 Verify.

Cette commande réaffiche la fenêtre en placant le curseur au milieu de l'écran. Elle a *été très utile pour développer et tester l'éditeur, mais n'a plus qu'un rôle* psychologique. Frapper **V** chaque fois que qu'on n'est pas certain de voir l'écran reflèter exactement le contenu du fichier.

2-4 Test (vérification syntaxique)

Après avoir saisi ou modifié un morceau de programme, il est utile de le vérifier syntaxiquement. Il n'est pas nécessaire de vérifier tout un programme; une plus petite unité syntaxique, par exemple une *procédure*, un *module* (UNIT en PASCAL-UCSD), ou même une *instruction* peut être testé. On lance la vérification en placant le curseur au début de l'unité et en frappant **T**; la ligne suivante apparaît(en PASCAL):

```
>Test(PASCAL) Block Instructio Procedured Typedecl Unitd Vardecl <esc> ?
```

Frapper **P** pour vérifier une *procedure*, une *fonction* ou un *programme*, **B** pour vérifier un *bloc*, **I** pour une *instruction*, **T** pour une *declaration de type*, **U** pour une *unité*, **V** pour une *déclaration de variable*. Le curseur avance dans le texte. Un message indique à la fin si aucune erreur n'a été détectée; les erreurs sémantiques ne sont pas signalées. En cas d'erreur syntaxique, un message est affiché. Si le message est un numéro d'erreur, consulter la liste (p.ex. le fichier **SYSTEM.SYNTAX** sur UCSD).

A la fin de la vérification on peut encore appuyer sur la touche **?**, cela provoque l'affichage des identificateurs indéfinis. Ce sont soit des identificateurs globaux à la procédure, soit des identificateurs mal orthographiés. Seuls les huit premiers caractères des identificateurs sont affichés

2-5 Margin (paragraphe)

En mode PROSE, cette commande permet d'ajuster un paragraphe en fonction des valeurs de LMARGIN, RMARGIN, PARAMARGIN et MISE EN PAGE fixés par **S**et.

Un paragraphe est la partie de texte comprise entre deux lignes vides. Si on place le curseur à une position quelconque d'un paragraphe et qu'on appuye sur la touche **M**argin, tous les mots sont tassés entre la marge gauche et la marge droite, si la mise en page est *en drapeau*. Des blancs sont inséres si la mise en page est *justifiée*. La première ligne du paragraphe commence à la valeur de PARAMARGIN.

Avant:

```
   Voici un petit texte qui devrait être justifié avec les valeurs
PARAMARGIN=3,LMARGIN=0,RMARGIN=49, MISE EN PAGE=JUSTIFIEE. Placez le
curseur dans ce paragraphe, frappez la touche Margin, voici le
résultat.
```

Après avec MISE EN PAGE=EN DRAPEAU:

```
   Voici un petit texte qui devrait être justifié
avec les valeurs PARAMARGIN=3,LMARGIN=0,RMARGIN=49,
MISE EN PAGE=EN DRAPEAU. Placez le curseur dans ce
paragraphe, frappez la touche Margin, voici le
résultat.
```

Après avec MISE EN PAGE=JUSTIFIEE:

```
   Voici un petit texte qui devrait être  justifié
avec les valeurs PARAMARGIN=3,LMARGIN=0,RMARGIN=49,
MISE EN PAGE=JUSTIFIEE. Placez le curseur  dans  ce
paragraphe, frappez  la  touche  Margin,  voici  le
résultat.
```

2-6 Write (Ecriture du fichier)

Pour sauvegarder le fichier sur disquette (ou sur disque) sous un nom différent de celui choisi en début de session (qui peut être consulté par la commande *Look*), frapper **W**. L'écran s'efface alors et on est invité à spécifier un nom de fichier. Le suffixe '.TEXT' est implicite, il ne faut donc pas l'ajouter. Une fois le fichier sauvegardé, on se retrouve dans l'éditeur. Si on s'est trompé en entrant dans cette commande, on peut l'annuller en frappant **<esc> <ret>**.

Comme à l'entrée de l'éditeur et pour la commande **Get** un **?** fait apparaître la liste des fichiers éditables.

2-7 Quit (Sortie de l'éditeur)

Lorsque toutes les modifications et adjonctions au texte sont faites, on veut quitter l'éditeur et mettre à jour le texte modifié sur la disquette (ou sur disque). Frapper **Q** pour quitter. Le message suivant s'affiche:

```
>Quit:
        <esc> sortir de l'editeur sans mise a jour,
        <ret> retourner a l'editeur sans mise a jour
        <etx> mise a jour du fichier MONFICHIER,
```

<esc> permet de quitter l'éditeur en perdant la session d'édition, par exemple lorqu'on a utilisé l'éditeur uniquement pour consulter un fichier.

<etx> permet de sauvegarder le fichier édité en tant que fichier de travail, ou sous le nom annoncé.
<ret> permet de retourner à l'éditeur,

vous avez alors le choix suivant:

```
>Quit:
        <ret>  retourner au fichier MONFICHIER
        E(diter un nouveau fichier
```

E permet de recommencer un nouvelle session d'édition en conservant un certain nombre de variables globales de l'éditeur à leur valeur actuelle: p.ex. les chaînes de recherche et de remplacement. On évite ainsi le temps de chargement de l'éditeur.

Lorsque le fichier a été sauvegardé avec succès le message suivant apparaît:

```
>Quit:
        Le fichier   MONFICHIER a ete mis a jour,
        <esc>        sortir de l'editeur
        <ret>        retourner au fichier MONFICHIER
        E(diter      un nouveau fichier
```

3. RESUME DES COMMANDES D'EDITION

Adjust justifie la ligne courante:
- **<right>** à droite d'un caractère
- **<left>** à gauche d'un caractère
- **<low>** propage le décalage à la ligne suivante
- **<up>** propage le décalage à la ligne précédente
- **<ret>** ajuste sur la ligne précédente
- **Left** sur la marge gauche
- **Right** sur la marge droite
- **Center** au milieu de la ligne

Begin déplace le curseur au début du fichier

Copy copie le tampon de copie à la position du curseur

Delete détruire depuis la position courante du curseur:
- **<right>** en avant, jusqu'à la fin du symbole courant
- **<left>** en arrière jusqu'au début du symbole courant
- **<low>** en descendant d'un ligne
- **<up>** en montant d'une ligne
- **<ret>** la ligne courante, y compris la fin de ligne
- **<tab>** jusqu'à la fin de la ligne courante

End déplace le curseur à la fin du fichier

Find recherche d'un symbole
- **<esc>** change le mode de recherche
- **text<etx>** recherche **text**

Get introduit le contenu d'un fichier
- **?<ret>** donne la liste des fichiers éditables
- fichier**<ret>**introduit le contenu du fichier à la position du curseur
- **<esc>** retourne à l'éditeur sans introduction

Home déplace le curseur en haut de l'écran

Insert introduit du texte:
- **text** introduit le texte tapé
- **<bs>** supprime le dernier caractère introduit
- **<del>** supprime la dernière ligne introduite
- **<col1>** ramène le curseur au début de la marge gauche
- **<right>** décale toute la ligne d'un caractère vers la droite
- **<left>** décale toute la ligne d'un caractère vers la gauche
- **<ret>** introduit une fin de ligne
- **<up>** en MODULA passe en majuscule pour le prochain caractère

Look affiche les paramètres du profil de l'utilisateur

Margin met en page un paragraphe

Opposite recule d'une page

Page avance d'une page
Quit quitte l'éditeur
- **<esc>** sans sauvegarde du fichier d'édition
- **<etx>** avec sauvegarde du fichier d'édition
- **<ret>** retour à l'édition du même fichier
- **E** édition d'un nouveau fichier

Replace remplace **anctext** par **nouvtext**
- **<esc>** change le mode de recherche (littéral/par symboles)
- **anctext<etx>nouvtext<etx>**

Set modifie certaines variables globales
- **K**eywords les mots-clefs (inhibés, minuscules, majuscules)
- lanGuage le langage, terminer par **<ret>**
- **F**ind le mode de recherche (littéral/par symboles)
- **I**ndentation l'indentation (0 à 9)
- **L**eft margin colonne du départ de la ligne éditée
- **R**ight margin colonne de la fin de la ligne éditée
- Paragraph la colonne gauche du début d'un paragraphe (double ligne vide)
- **M**ise en page inhibée, en drapeau, justifiée à droite
- l**O**g commentaire en fin session

Test vérifie la syntaxe (de parties) de programme
View réaffiche l'écran avec le curseur au milieu de l'écran
Write sauvegarde sur un fichier
- **?<ret>** donne la liste des fichiers éditables
- fichier**<ret>**

Xchange modifie la ligne courante
- char substitue le caractère frappé à celui qui est affiché
- **<right>** laisse le caractère inchangé
- **<left>** déplace le curseur sur la gauche
- **<tab>** déplace le curseur à la fin de la ligne courante
- **<bs>** restaure l'ancien caractère
- **<low>** déplace le curseur dans la colonne du bas
- **<up>** déplace le curseur dans la colonne du haut
- **<ret>** passe au début de la ligne suivante
- **<col1>** déplace le curseur au début de la ligne courante
- **<insb>** insère un blanc
- **<delc>** supprime le caratère désigné par le curseur
- **<invc>** change le caractère *minuscule en majuscule ou le contraire*

Zap détruit le fichier d'édition entre le curseur et la marque
<delc> supprime le caractère se trouvant sous le curseur
<insb> *insère un blanc à droite du curseur*
• mémorise la position du curseur
= déplace le curseur à la position fixée par la commande **•**
<right> avance le curseur au début du symbole suivant
<left> recule le curseur au début du symbole précédent

<low>	déplace le curseur dans la même colonne sur la ligne au-dessous
<up>	déplace le curseur dans la même colonne sur la ligne au-dessus
<tab>	avance le curseur à la fin de la ligne courante (ou suivante)
<ret>	avance le curseur au début de la ligne suivante
<bs>	recule le curseur d'un caractère
<sp>	avance le curseur d'un caractère
<etx>	recherche la dernière chaîne enregistrée (voir **F** et **R**)
<	met la direction des mouvements en marche arrière
>	met la direction des mouvements en marche avant
?	affiche les autres commandes
<dc2>	force tout en majuscules ou revient aux minuscules

CURRICULUM VITAE

Je suis né le 18 mai 1944 à Bienne (BE). De 1960 à 1963 j'ai suivi les cours au Gymnase français de Bienne pour y obtenir le certificat de maturité en 1963. En 1963 j'ai commencé des études de mathématiques au Département de Mathématiques de l'École Polytechnique Fédérale de Zürich, puis à l'Université de Neuchâtel où j'ai obtenu ma licence ès sciences en 1973. J'ai travaillé comme assistant non-licencié de 1970 à 1973, puis comme assistant jusqu'en 1979 au Centre de Calcul et à l'Institut de Mathématiques de l'Université de Neuchâtel. Depuis 1979 je suis assistant au Département de Mathématiques de l'Ecole Polytechnique Fédérale de Lausanne.