**Historique IX: Autres langages**

* APL et SNOBOL: Langages Dynamiques
* SIMULA 67: l ’Abstraction des Données
* ALGOL 68: La Conception Orthogonale
* Pascal, C: les Descendants des ALGOLs
* Prolog: Programmation Basée sur la Logique
* Ada: Langage du Département de la défense.
* Smalltalk: Programmation Orientée Objet
* C++: Combinaison de Langage Impératif et Langage Orienté Objet
* Java: Simplification du C++ et Programmation du World-Wide Web

**Bibliographie**

Robert Sebesta,

Concepts of programming languages.

David A. Watt,

Concepts of programming languages.

Elis Horrowitz,

Fundamentals of Programming Languages

**Pourquoi l’Étude des paradigmes des Langages de Programmation ?**

* La capacité d’exprimer des idées d’informatiques complexes est liée a une bonne connaissance des caractéristiques linguistiques de programmation.
* Différentes tâches informatiques demandent des caractéristiques linguistiques de programmation différentes.
* La connaissance des paradigmes de langages de programmation permet une acquisition plus aisée de nouveaux langages.
* Une compréhension de la façon dont les langages sont implémentes permet un meilleur usage de ces langages.
* La connaissance de différents paradigmes de langages de programmation permet une création plus aisée de nouveaux langages.
* Avec une meilleure connaissance des langages de programmation, de meilleurs choix peuvent être fait, ce qui peut éventuellement bénéficier l ’ensemble du champ d’informatique.

**1**

**Exemples de Différents Domaines de Programmation et de leurs Langages Associés**

* Applications Scientifiques --> Pas besoin de structures des données ni d ’opérations compliquées mais besoin de grande Efficacité --> FORTRAN, ALGOL60.
* Applications pour Affaires --> Besoin de produire des rapport élaborés et de traiter les nombres décimaux et les caractères --> COBOL.
* Intelligence Artificielle --> Besoin de manipuler les données symboliques plutôt que numériques et besoin d ’opérations logiques --> LISP, PROLOG
* Programmation des Systèmes --> Besoin d ’une exécution rapide et de facilités de bas niveau pour pouvoir programmer les engins externes --> PL/1, Extended ALGOL, C.
* Langages pour Scriptes --> Besoin de mettre des commandes dans un fichier pour exécution --> ksh, awk, tcl, Perl.
* Langages Spéciaux --> Langages pour besoins particuliers --> RPG, APT, GPSS.

**Classification des Langages de Programmation**

**Langages impératifs**

**Langages déclaratifs**

**Programmation procédurale Ada Pascal C**

**ad**

**Programmation Orientée objets C++ java**

**Programmation concurrente Ada 95**

**Programmation fonctionnelle Lisp Schème**

**Programmation logique Prolog**

**Tous les langages De programmation**

* ***Langages Impératifs***: Langages incluant des moyens pour le programmeur d ’attribuer des valeurs à des locations (Variables) en mémoire.
* ***Langages Déclaratifs***: Langages pour lesquels le programmeur réfléchit en terme de valeurs des fonctions et de relations entre entités diverses. Il n ’y a pas d ’attribution de valeurs aux variables.
* ***Programmation Procédurale***: Le programme est divisé en blocs qui peuvent contenir leurs propres variables ainsi que d ’autres blocs.
* ***Programmation Orientée Objet***: Programmation qui supporte l ’interaction d ’objets. Un objet contient des données ainsi que des fonctions qui peuvent s ’appliquer a ces données.
* ***Programmation Concurrente***: Langages de programmation qui s ’appliquent à plusieurs CPU ’s qui opèrent en parallèle. Les données peuvent être partagées ou non.
* ***Programmation Fonctionnelle***: Un programme est un appel de fonction avec un certain nombre de paramètres, qui eux-mêmes peuvent être des appels à d ’autres fonctions. Le programme renvoie donc un seul résultat, qui peut-être assez complexe (exemple: une nouvelle fonction).

**2**

* ***Programmation Logique***: Un programme consiste en une série d ’axiomes, de règles de déduction et en un théorème à prouver. Le programme renvoie la valeur « vrai » si les axiomes supportent le théorème. Il renvoie la valeur « fause » autrement.

**Historique : Un Premier Language: Plankalkül**

* Développé en 1945 par Konrad Zuse en Allemagne mais jamais implémenté. Description publiée en 1972.
* Langage assez complet avec des structures de données bien avancées contenant des tableaux et des articles (records). Les articles peuvent utiliser la récursivité pour inclure d ’autres articles.
* Il n ’y a pas de spécification « Goto », mais il y a une spécification itérative telle que la boucle « For » en Pascal.

**Historique : Pseudo Codes (Suite)**

* Vers la fin des années 1940 et le début des années 1950, il n ’y avait pas de langages de programmation de haut niveau ni même de code assembleur. Des langages de haut niveau ont donc été inventés pour rendre le code plus lisible.
* ***Short Code*** et ***Speedcoding***, par exemple, étaient des langages ***interprétés*** et bien qu ’ils facilitaient le processus de programmation, ils étaient beaucoup plus lent que le code machine.
* Les compilateurs pour UNIVAC, A-0, A-1 et A-2 sont les premiers ***compilateurs*** qui étendaient le pseudo-code en code machine de la même façon que des macros sont étendus en code assembleur.

**Historique : FORTRAN (Suite)**

* FORTRAN 0 a été crée en 1954 par l ’équipe de John Backus à IBM.
* Il a été suivi par FORTRAN I en 1956, FORTRAN II en 1958, FORTRAN IV entre 1960 et 1962, FORTRAN 77 en 1978 et FORTRANT 90 en 1992.
* Au tout début, FORTRAN a été conçu comme un prélude nécessaire à la conception et traducteur pour l ’IBM 704. Néanmoins il a connu un grand succès.
* L ’efficacité du FORTRAN est dû au fait que les types et la location de chaque variable en mémoire est fixée avant l ’exécution.
* FORTRAN a changé à jamais la façon dont les ordinateurs sont utilisés.

**Historique : LISP (Suite)**

* Vers le milieu des années 1950, il y a eu beaucoup d ’intérêt pour la discipline nouvelle d ’Intelligence Artificielle (IA).
* Avec cet intérêt, s ’est développée l ’idée que certaines méthodes devaient être inventées afin de permettre aux ordinateurs de traiter les données symboliques dans des listes chaînées. Jusque là, l ’usage des ordinateurs était réservé aux calculs sur données numériques dans des tableaux.

**3**

* LISP, développé par l ’équipe de McCarthy et Minsky au MIT à partir de 1958, a dominé l ’IA pendant longtemps mais il est assez inefficace. De nos jours des versions de LISP sont compilées, ce qui permet une plus grande efficacité.

**Historique : ALGOL 60 (Suite)**

* L’ALGOL 60 a été créé vers la fin des années 1950 dans un but de créer un langage universel et indépendant de la machine.
* L ’ALGOL 60 inclut un certain nombre de nouvelles caractéristiques importantes:
  + Le concept de structures en bloc,
  + La possibilité de passer les paramètres à un sous-programme par valeur ou par nom,
  + La possibilité pour les sous-programmes d ’être récursifs,
  + La possibilité de déclarer les limites d ’un tableau pendant l ’exécution.
* Bien que l ’ALGOL 60 n ’ait pas réalisé le succès espéré, il est le père de tous les langages impératifs.

**Historique : COBOL (Suite)**

* Bien que le COBOL ait été utilisé plus que tous les autres langages, il a eu peu d ’effet sur la plupart de ces langages.
* L ’idée du COBOL était de créer un langage commun pour les applications d ’affaires.
* Ces caractéristiques sont l ’usage d ’autant d ’anglais que possible et l ’aise d ’usage même au dépend de son effectivité et puissance.
* COBOL est puissant au niveau des données, mais ses procédures sont assez faibles. En particulier, COBOL ne permet pas les fonctions. Par contre il a été le premier langage à inclure les structures de données hiérarchiques et de permettre les noms de variables assez long pour être significatif.

**Historique : BASIC (Suite)**

* BASIC a aussi connu un très grand usage mais n ’a pas connu beaucoup de respect de la part de la communauté informatique.
* Il a été créé au début des années 1970 pour étudiants.
* C ’est le premier langage qui a utilisé l ’accès d ’un terminal à distance plutôt que des cartes perforées.
* Au départ, le BASIC était très proche du FORTRAN, avec quelque influence mineure de l ’ALGOL 60. Plus tard, il a évolue dans des directions diverses avec peu de standardisation.
* La plus grande critique du BASIC est que ses programmes sont écrits avec peu de structure interne et sont donc difficile à lire et modifier.

**4**

**Historique : PL/1 (Suite)**

* Le PL/1 représente le premier essai à créer un langage qui pourrait être utilisé dans un grand nombre de domaines. Néanmoins, la plupart des langages qui ont suivi se sont concentrés sur un domaine particulier.
* Le PL/1 a inclut ce qui était considéré comme les meilleures caractéristiques de l ’ALGOL 60 (récursivité et structure en bloc), du FORTRAN IV (compilation séparée avec communication à travers des données globales) du COBOL 60 (structure des données, input/output, et possibilité de génération de rapports) ainsi que d ’autres caractéristiques nouvelles (e.g., les pointeurs). Néanmoins, le langage est trop complexe et ses nouvelles caractéristiques, mal conçues.

**Historique IX: Autres langages**

* APL et SNOBOL: Langages Dynamiques
* SIMULA 67: l ’Abstraction des Données
* ALGOL 68: La Conception Orthogonale
* Pascal, C: les Descendants des ALGOLs
* Prolog: Programmation Basée sur la Logique
* Ada: Langage du Département de la défense.
* Smalltalk: Programmation Orientée Objet
* C++: Combinaison de Langage Impératif et Langage Orienté Objet
* Java: Simplification du C++ et Programmation du World-Wide Web

**5**

**Critères d ’Évaluation des Langages**

**Besoin d ’Évaluation**

* Étant donnée la prolifération de langages de programmation, il est important d ’évaluer ces langages afin de pouvoir les choisir de manière appropriée et de les améliorer.
* Trois critères d ’évaluation sont généralement acceptés:
  + - la ***lisibilité***,
    - la ***facilite d ’écriture***,
    - et la ***fiabilité***.
* La ***lisibilité*** correspond a la facilite avec laquelle un programme peut-être lu et compris.
* La ***facilite d ’écriture*** correspond a la facilite avec laquelle un langage peut être utilise pour créer un programme.
* Un programme est ***sûr*** (fiable) s ’il peut être exécuté sous toutes les conditions.
* Un autre critère important est le coût de ce langage.

**Caractéristique de Langages qui Affectent les Critères d ’Évaluation**

***Critères***

|  |  |  |  |
| --- | --- | --- | --- |
| **Ca*ractéristique*** | ***Lisibilité*** | ***Écriture*** | ***Fiabilité*** |
| Simplicité/Orthogonalité |  |  |  |
| Structure de Contrôle |  |  |  |
| Type et Structure des Données |  |  |  |
| Conception Syntactique |  |  |  |
| Soutien pour l ’Abstraction |  |  |  |
| Expressivité |  |  |  |
| Vérification de Type |  |  |  |
| Traitement d ’Exceptions |  |  |  |
| Restriction d ’Alias |  |  |  |

**Caractéristiques Affectant la Lisibilité: La Simplicité**

* La Simplicité:
  + S ’il y a beaucoup de composantes de bases, il est difficile les connaître toutes.
  + S ’il existe plusieurs façons d ’exprimer une commande, il est aussi difficile de les connaître toutes

**6**

* + Si les opérateurs peuvent être surcharges, cela constitue une autre source de difficulté pour la lecture.
* Néanmoins, la simplicité ne doit pas être pousse trop loin car, paradoxalement, elle cause la difficulté de lecture.

**Caractéristiques Affectant la Lisibilité: L ’Orthogonalité**

* L ’orthogonalité dans un langage de programmation signifie que toutes les combinaisons de primitives sont légales et significatives.
* De plus, la signification d ’un élément du langage orthogonal est indépendante du contexte dans lequel il apparaît.
* S ’il y a peu d ’orthogonalité dans le langage, il y aura beaucoup d ’exceptions a ses règles, et cela causera beaucoup de difficultés pour la lecture et la compréhension du code.

**Caractéristiques Affectant la Lisibilité: Instruction de Contrôle**

* Il est très important, pour la lisibilité d ’une langue de programmation d ’avoir des structures de contrôle adéquate.
* Par exemple, l ’un des plus grand problème du premier BASIC est que sa seule instruction de contrôle était le « goto ». Cela créait des situations ou le lecteur était renvoyé à différents points du programme et cassait son flot régulier.
* Depuis la fin des années 1960, la plupart des langages incluent des instructions de contrôle adéquates et cette caractéristique est moins importante relativement à la lisibilité.

**Caractéristiques Affectant la Lisibilité: Types et Structures des Données**

* La présence de moyens appropriés pour définir des types et structures de données dans un langage peut beaucoup aider la lisibilité.
* Par exemple, la présence du type « Booleen » peut clarifier certaines instructions.
* De même la possibilité de créer des enregistrements a type varie (« records ») peut clarifier des expressions qui seraient, sans leur présence, indiquées par plusieurs tableaux a indices communs.

**Caractéristiques Affectant la Lisibilité: La Syntaxe**

* Voici des exemples dans lesquelles les choix de syntaxe peuvent influencer la lisibilité:
  + La ***forme des identificateurs***: des identificateurs trop court peuvent rendre la lecture difficile.
  + Les ***mots spéciaux***: Ces mots peuvent aider à la compréhension (par exemple « begin », « end », « end for », etc.). Néanmoins, si ces mots peuvent aussi être utilises comme identificateurs, cela rend la lecture plus difficile.
  + Si chaque construction a la même signification dans des contextes différents, la lecture sera plus aisée.

**7**

**Caractéristiques Affectant la Facilite d ’Écriture: Simplicité et Orthogonalité**

* Si un langage contient un large nombre de constructions différentes, il est fort possible que certain programmeurs ne les connaissent pas toutes. Dans ce cas il se peut que ces programmeurs n ’utilisent pas les meilleures constructions présentes.
* De même, il se peut que le programmeur ne connaissent certaines constructions que superficiellement et les utilise de manière erronée.
* Néanmoins, trop d ’orthogonalité peut être détrimentale car cela apporterait trop de liberté et les erreurs ne pourraient pas être détectées par le compilateur.

**Caractéristiques Affectant la Facilite d ’Écriture: Possibilités d ’Abstraction**

* L ’abstraction est la possibilité de définir des structures ou des opérations compliquées tout en en cachant leurs détails.
* L ’abstraction est très importante dans l ’écriture d ’ un programme car elle peut rendre l ’écriture beaucoup plus aisée. Il y a deux types d ’abstraction: abstraction de processus et abstraction de données.
* Quand un processus est abstrait dans un sous-programme il n ’est pas nécessaire de répéter son code a chaque fois qu ’il est nécessaire. Un simple appel de la procédure est suffisant.
* Les données peuvent être abstraites en C++ ou Java dans des objets a interface simple. Exemple: un arbre.

**Caractéristiques Affectant la Facilité d ’Écriture: L ’Expressivité**

* Un langage est expressif s ’il a des moyens simples et commodes de spécifier des calculs.
* Exemple, en C, « count++ » est plus commode et plus simple que « count = count + 1 »
* De même, l ’inclusion de l ’instruction « for » en Pascal rend l ’écriture plus simple que si la boucle « while » doit être utilisée.

**Caractéristiques Affectant la Fiabilité: Vérification de Type**

* La vérification de type signifie qu ’un langage est capable de détecter les erreurs relative au types des données ou bien au moment de la compilation ou au moment de l ’exécution.
* Quand un langage, comme le C, ne détecte pas ces erreurs, le programme peut-être exécuté, mais les résultats ne sont pas significatifs.
* Par exemple, l ’étendue des tableaux n ’est pas vérifiée en C, et lorsque les indices dépassent les limites, le programme ne s ’arrête pas bien que les valeurs attente n ’aient pas de sens.

**Caractéristiques Affectant la Fiabilité: Prise en Charge des Exceptions**

* La possibilité pour un programme d ’intercepter les erreurs faites pendant l ’exécution, de les corriger, et de continuer l ’exécution augmente de beaucoup la fiabilité du langage de programmation..
* Ada, C++ et Java ont des capacités de prise en charge d ’exception, mais de telles capacités sont inconnues à d ’autres langages tels que le C ou le FORTRAN

**8**

**Caractéristiques Affectant la Fiabilité: Présence d ’Alias**

* Le fait d ’avoir des alias signifie qu ’il y a plus d ’un moyen de faire référence à des cellules de mémoire.
* Il est couramment accepte que la permission d ’avoir des alias dans les langage de programmation est dangereux.
* Néanmoins, beaucoup de langage de programmation utilisent des alias, par exemple, pour surmonter leur faiblesse dans le domaine de l ’abstraction des données. D ’autres langages imposent des restrictions sur les alias afin de permettre une plus grande fiabilité.

**Caractéristiques Affectant la Fiabilité: Lisibilité et Facilite d ’Écriture**

* Lisibilité et facilité d ’écriture influencent la fiabilité des langages de programmation.
* En effet, si il n ’y a pas de moyen naturel d ’exprimer un algorithme, des moyens non naturels seront utilisés, et il y aura une plus grande chance que ces moyens soient incorrects.
* Programmes qui sont difficile a lire sont aussi difficile a écrire et donc a modifier ou maintenir. Les programmes qui résultent sont donc également moins fiable.

**Le Coût d ’un langage de programmation**

* Ce coût est fonction de beaucoup des caractéristiques du langage:
  + Si le langage n ’est pas simple et orthogonal, il est plus cher de former des programmeurs et d ’écrire des programmes dans ce langage. Néanmoins, un bon environnement de programmation peut beaucoup aider a réduire ce coût.
* Le coût de la compilation et de l ’exécution de programmes sont aussi des facteurs. L ’Optimisation peut néanmoins réduire le coût de l ’exécution.
* Un autre facteur est le coût de la mise en oeuvre du langage. Si cette mise en oeuvre ou la machine sur laquelle le langage peut être exécuté est chère, il y a peu de chance que le langage devienne populaire.
* Un autre coût est le coût lie au manque de fiabilité. Si le programme échoue dans le cas d ’une usine nucléaire ou une machine a rayons X, les dommages seront très grand et très chers.
* Enfin, le coût de maintenir le langage (corrections, modifications et addition de nouvelles capacités) est aussi un considération très importante.
* Les trois plus grands contributeurs au coût d ’un langage de programmation sont:
  + Le développement de programmes
  + Leur maintenance
  + Leur fiabilité
* Étant donné que ces contributeurs dépendent de la lisibilité des programmes et la facilite de l ’écriture, ces critères sont, en conséquence, les plus importants.
* Néanmoins, il y a aussi d ’autres critères tels que:

**9**

* + La portabilité
  + La généralité
  + La précision et la complétude de la description.

**Description Syntaxique des Langages**

**Pourquoi Décrire les Langages de Programmation? Et Comment?**

* Les langages de programmation sont implémentés et utilisés par un grand nombre de personnes. Ils doivent donc être compris par toutes ces personnes, avant même qu’ils n’existent ou qu’ils soient utilisés.
* La clarté et la précision de la description sont très intimement liés au succès d’un nouveau langage.
* La description d’un langage se divise en deux parties: description ***syntaxique*** et description ***sémantique.***
* La syntaxe correspond à la description de la ***forme*** des expressions, instructions et unités de programmes.
* La sémantique correspond a leur ***signification***.
* La syntaxe est plus facile à décrire que la sémantique.

**La Description Syntaxique: Définitions**

* Les langages sont des chaînes de caractères provenant d’un ***alphabet***.
* Chaque chaîne représente une ***phrase***.
* Il existe de petites unités syntaxiques appelées des ***lexèmes***.
* Les ***tokens*** d’un langage représentent des catégories de lexèmes. Certains tokens ne représentent qu’un seul lexème, mais d’autres en représentent plusieurs.
* Les langages peuvent être décrits de deux façons distinctes: ***reconnaissance*** ou ***génération***.

**Reconnaisseur de Langages :**

* Étant donné un langage L qui utilise l’alphabet Σ, on peut construire un mécanisme de reconnaissance R capable de reconnaître les chaînes de caractères de l’alphabet Σ.
* R doit indiquer si une chaîne qui lui est donnée appartient ou non au langage. En d’autre termes, il va accepter ou rejeter cette chaîne.
* De façon pratique, dans un compilateur, l’analyse syntaxique est faite par un système de reconnaissance.

**Générateur de Languages**

* Un générateur de langage génère des phrases d’un langage.
* De façon pratique, un générateur peut sembler peu utile en tant que descripteur de langage, mais en fait, il est très utile car il permet une description facile à lire et à comprendre. **10**
* En fait, un générateur est beaucoup plus utile qu’un reconnaisseur quant à la description du langage.

**Méthodes Formelles de Description Syntaxique: Généralités :**

* Les mécanismes formels de génération de langage s’appellent des ***grammaires***.
* Les grammaires les plus utiles pour les langages de programmation sont les ***grammaires hors contexte (context-free)***
* La notation utilisée pour spécifier de telles grammaires s’appelle la forme Backus-Naur (BNF).
* BNF est une forme très naturelle de description syntaxique.

**Méthodes Formelles de Description Syntaxique: Fondations :**

* Un ***métalangage*** est un langage destiné à décrire d’autres langages. BNF est un métalangage pour les langages de programmation.
* BNF définit les structures syntaxiques du langage en utilisant des ***règles*** (ou ***productions***).
* La ***partie gauche (LHS)*** d’une règle BNF ne contient qu’un symbole ***non-terminal***, alors que la ***partie droite (RHS)*** peut contenir des symboles ***terminaux*** ou non.
* Un non-terminal peut avoir plusieurs définitions.
* Une règle peut aussi être ***récursive***.

**Méthodes Formelles de Description Syntaxique: Définitions :**

* Les phrases d’un langage sont générées par l’application en séquence des règles de la grammaire.
* La génération d’une phrase ou d’un programme entier s’appelle une ***dérivation*** et elle commence au ***symbole de départ***.
* S’il y a plusieurs non-terminaux dans une règle et que la dérivation étend le non-terminal le plus a gauche avant les autres, on parle de “***dérivation la plus à gauche***”.
* Un avantage des grammaires est qu’elles décrivent naturellement la structure syntaxique hiérarchique des phrases des langages qu’elles décrivent. Ces structures hiérarchiques s’appellent des “***Arbres de dérivation***”.

**Méthodes Formelles de Description Syntaxique: Considérations :**

* Une grammaire qui peut générer une phrase ayant deux ou plus de deux « arbres de dérivation » distincts est une ***grammaire ambiguë***.
* L’ambiguïté syntaxique est un problème dans les langages de programmation car la sémantique est basée sur la syntaxe et la signification d’une phrase doit être unique.
* Une grammaire peut-être écrite de manière a implémenter la ***précédence des opérateurs,*** et leur ***associativité***.

**11**

**EBNF: Une extension de la BNF :**

* Il existe certaines extensions de la BNF qui n’étendent pas le pouvoir descriptif du méta-langage, mais plutôt, en étend sa lisibilité et sa facilité d’écriture.
* 3 extensions sont utilisées couramment:
  + L’utilisation de “[“ “]” --> Option (oui/non)
  + L’utilisation de “{“ “}” --> Itération
  + L’utilisation de “|” --> Option a choix multiple
  + L’utilisation de “+” --> répétition unique ou multiple

**Arbres de derivation :**

* Le processus appelé “dérivation” consiste à construire un “arbre de dérivation” pour la phrase ou le programme écrits dans le langage.
* La dérivation peut-être fait du haut en bas ou du bas en haut.
* Parsing peut-être fait assez facilement si la grammaire du langage est définie de manière appropriée.
* En particulier, il est important que cette grammaire ne soit pas récursive.

**Grammaires à Attributs: Généralités :**

* Une ***grammaire à attributs*** est un mécanisme qui permet de décrire une plus grande partie de la structure d’un langage qui n’est possible avec une grammaire hors contexte.
* En particulier, une telle grammaire permet la définition de certaines règles du langage tels que la compatibilité des types.
* En général, la catégorie de règles du langage qui ne peuvent pas être définies en BNF s’appelle la ***sémantique statique*** de ce langage.

**Grammaires à Attributs: Définitions :**

* Une grammaire à attribut est une grammaire avec les additions suivantes:
  + Chaque symbole de la grammaire est associé à un ensemble d’attributs divisé en deux sous-ensembles: les ***attributs synthétisés***, qui passent l’information sémantique vers le haut de l’arbre; et les

***attributs hérités***, qui la passent vers le bas.

* + Chaque règle de grammaire est associée a un ensemble de ***fonctions sémantiques*** et, peut-être, un ensemble de ***fonctions prédiquées***.
* Un ***attribut intrinsèque*** est un attribut synthétisé qui appartient à une feuille d’arbre et dont la valeur est déterminée à l’extérieur de l’arbre.

**12**

**Description Sémantique des Languages**

**Rappel: Sémantique Statique vs Sémantique Dynamique**

* La sémantique statique représente les formes légales des programmes qui ne peuvent pas être facilement décrites en grammaire BNF. On appelle cette sémantique, statique, car elle est vérifiée pendant la compilation.
* La sémantique dynamique décrit la signification des programmes ou les effets encourus par l’exécution d’un programme.

**Pourquoi Décrire la Sémantique Dynamique?**

* Les programmeurs doivent savoir exactement ce que fait chaque portion de leur programme
* Les personnes qui écrivent les compilateurs doivent aussi savoir ce que doit faire chaque instruction.
* Bien qu’elles soient imprécises, les programmeurs et développeurs de compilateurs doivent se servir de descriptions en Anglais car les descriptions de sémantique formelle sont très complexes.
* Néanmoins, la définition d’une notation formelle et adéquate serait importante car elle pourrait aider les développeurs de compilateurs avec des descriptions plus précises, et peut-être, même permettre la génération de compilateur automatique.

**Specification de la Sémantique Dynamique (ou, simplement, Sémantique)**

Il y a trois méthodes de spécification sémantique:

* + Description ***Opérationnelle***: la signification d’un programme est déterminée par l’exécution de ses énoncés sur une machine virtuelle.
  + Description ***Dénotationelle***: la signification d’un programme est décrite a l’aide de fonctions montrant l’effet de l’application d’un énoncé sur l’état de la machine.
  + Description ***Axiomatique***: la signification d’un programme est décrite a l’aide d’assertions spécifiant les contraintes et relations

qu’imposent un énoncé.

**Sémantique Opérationnelle**

* L’idée de la sémantique opérationnelle est de décrire la signification d’un programme en exécutant ses instructions sur une machine réelle ou simulée. Les changements qui prennent place dans l’état de la machine lorsqu’elle exécute ces instructions représente la signification de cette instruction.
* Pour construire une machine simulée idéalisée, il faut deux composantes: un ***traducteur*** qui traduit le langage L en langage de bas niveau et une ***machine virtuelle*** dont l’état change lorsque le code de bas niveau est exécuté.
* La sémantique opérationnelle est effective. Néanmoins, elle n’est pas formelle et peut créer des circularités.

**13**

**Sémantique Dénotationnelle**

* La sémantique dénotationnelle est la méthode la plus rigoureuse de description sémantique des programmes.
* L’idée consiste à définir, pour chaque entité du langage, un objet mathématique et une fonction qui attache les instances de cette entité aux instances de l’objet mathématique correspondant.
* Comme pour la sémantique opérationnelle, le statut d’une machine idéalisée (en fait la valeur des variables) représente la signification d’une instruction
* La difficulté de cette méthode est dans la création d’objets et de fonctions pour ces objets. La notation est aussi difficile a lire quoi que très concise.

**Sémantique Axiomatique I**

* La ***sémantique axiomatique*** est définie en conjonction avec une méthode de preuve de validité de programmes.
* Lorsque le programme est correct, il existe une preuve de validité et dans cette preuve, chaque proposition est précédée et suivie d’une expression logique (***pre-condition*** et ***post-condition***) qui spécifie des contraintes sur les variables du programme. Ce sont ces contraintes qui définissent la signification du programme.

**Sémantique Axiomatique II**

* La ***pre-condition la plus faible*** représente la pre-condition la moins restrictive qui garantie la validité de la post-condition associée a l’instruction du programme.
* Si la pre-condition la plus faible peut être calculée a partir de la post-condition définie pour chaque instruction du langage, preuves de validité peuvent être construites pour les programmes de ce langage.
* Les preuves sont construites en partant de la fin d’un programme et en remontant vers son début.
* La sémantique axiomatique n’est pas très utile pour décrire la signification des langages de programmation a cause de sa complexité. Néanmoins, elle est utile pour la recherche et pour le raisonnement sur les programmes.

**Sémantique Axiomatique III**

* Plus précisément, la vérification de programmes se fait en deux étapes:
  + l’association d’une formule avec chaque étape du calcul significatif.
  + La démonstration que la formule finale s’ensuit logiquement de la formule initiale grâce aux étapes et formules intermédiaires.
* Les formules pour l’affectation et les conditions sont les formules de base. L’effet de toutes les autres instructions en découlent logiquement.

**14**

**Sémantique Axiomatique IV: l’Affectation**

* Supposons que ***x = E*** soit une instruction d’affectation et que ***Q*** soit sa post-condition. Alors, sa pre-condition est définie par l’axiome  ***P = Q x -->*** qui signifie que P est calculé comme Q avec toutes les instances de x remplacées par E.
* Comment peut-on prouver l’exactitude de programmes (et en particulier d’une instruction d’affectation) avec de tels outils?

**Sémantique Axiomatique V: Justification de la procédure**

* Une instruction d’affectation avec sa pre-condition et sa post-condition peuvent être considérés comme des ***théorèmes***.
* Si l’axiome d’affectation, appliquée à la post-condition et à l’instruction d’affectation, produit la pre-condition donnée, alors on peut dire que le théorème est ***prouvé***, et donc, le programme est exact ou correcte.

**Sémantique Axiomatique VI: la *Règle de Conséquence* (rétrécissement ou élargissement)**

* Parfois, la pre-condition obtenue par la procédure ne correspond pas à la pre-condition attendue.
* Dans ce cas, on peut se servir de la règle de conséquence qui est la règle d’inférence suivante:

{P} S {Q}, P’ => P, Q => Q’

{P’} S {Q’}

**Sémantique Axiomatique VII: Séquences d’instructions**

Etant donné deux instructions adjacentes avec les pre- et post- conditions suivantes:

{P1} S1 {P2}

{P2} S2 {P3}

La règle d’inférence pour une telle séquence est:

{P1} S1 {P2}, {P2} S2 {P3}

{P1} S1 ; S2 {P3}

**Sémantique Axiomatique VIII: les Instructions de Sélection**

* ***If-then-else***:

{B and P} S1 {Q}, {(not B) and P} S2

{Q}

{P} if B then S1 else S2 {Q}

* ***If-then:***

{B and P} S1 {Q}, {(not B) and P} => Q

{P} if B then S1 {Q} **15**

**Sémantique Axiomatique IX: Les Boucles a Test Initial**

* Dans une boucle à test initial (ou une boucle “while”), on a une répétition d’instruction. Le problème avec ces boucles, cependant, est qu’on ne sait pas combien de répétitions il y a => il est assez difficile de déterminer l’exactitude de ces boucles.
* La méthode utilisée est similaire à la ***méthode mathématique d’induction***.
* L’hypothèse inductive s’appelle ***l’invariant de la boucle*** (loop invariant)

**Sémantique Axiomatique X: Les Boucles a Test Initial**

* La règle d’inférence qui permet de trouver la pre-condition d’une boucle “while” est la suivante:

{I and B} S {I}

{I} while B do S end {I and (not B)}

I représente l’invariant de la boucle mais il n’est . pas fourni. C’est a nous de le trouver!

* Comment? En calculant la pre-condition pour un certain nombre de répétitions et en essayant de deviner un motif.

**Sémantique Axiomatique XI: Les Boucles à Test Initial**

* Mais trouver l’invariant de boucle n’est pas tout!!!
* Etant donné l’instruction ***{P} while B do S end {Q}***, et l’invariant de boucle, ***I*** , voici un résumé de toutes les choses qui doivent être démontrées afin de prouver l’exactitude d’une boucle “while”:

P => I

{I} B {I}

{I and B} S {I}

(I and (not B)) => Q

La boucle se termine.

**16**

**Paradigms des Languages de Programmation**

**Noms, Liaisons, Vérification de Type et Portée**

**Qui a besoin d’un nom?**

* Un nom est une ***voie d’entrée*** sur l’une des entités . d’un programme. On réfère à quelque chose par un . nom si on veut créer cette chose, l’utiliser, le . changer ou le détruire.
* Les entités qui ont besoin d’un nom sont:
  + Les constantes et les variables,
  + Les opérateurs,
  + Les labels,
  + Les types,
  + Les procédures et les fonctions
  + Les modules et les programmes,
  + Les fichiers et les disques,
  + Les commandes et les items de menus,
  + Les ordinateurs, les réseaux et les usagers

**Identificateurs**

Un nom est désigné par un ***identificateur*** sur lequel il peut exister des contraintes:

* + Nombre maximum de lettres,
  + Ensemble de caractères permis,
  + Sensibilité aux majuscules,
  + Présence de mots réservés.

Exemple d’un identificateur en Ada:

<identificateur> ::= <lettre>

{[<sous-ligne>] (<lettre>|<chiffre>)}

Un ***alias*** est un autre nom pour une même entité (i.e., deux identificateurs désignant la même entité)

**Mots Réservés**

* Les ***mots réservés*** représentent la “colle syntaxique” d’un programme.
* **Exemple 1:** ***if*** C ***then*** S1 ***else*** S2 ***end if***;

Ceci est en fait un triplet: ( C, S1, S2 ).

* **Exemple 2:** ***while*** C ***loop*** S ***end loop;***

Ceci est en fait une paire: (C, S).

* Néanmoins, les mots réservés déterminent le style du langage.
* Un ***mot clé*** est un mot spécial seulement dans certains contextes.
* Un ***mot pré-défini*** est entre les mots réservés et clés.

**Variables**

* Une variable de programme est une abstraction de cellule de mémoire ou d’une collection de cellules.
* Une variable peut être caractérisée par un 6-tuplet: .

**Nom**  **Addresse**   **Valeur**  **Type**  **Duree de vie**  **Portée**

L’usager décide du nom et du type de la variable.

L’emplacement de la déclaration décide de sa portée et de sa longévité. Son adresse est déterminée pendant l’exécution et sa valeur dépend des instructions dans lesquelles elle apparait.

**Durée de Vie et Portée**

Dans la plupart des langages de programmation, le même nom peut être réutilisé dans des contextes différents et peut représenter des entités différentes.

* **Exemple:**  **procedure a;**

**Var b: char;**

**begin …. End;**

**Procedure b;**

**var a: integer;**

**begin … end;**

Des types, adresses et valeurs différentes peuvent être associes a ces apparitions de nom. Chaque apparition a une durée de vie différente et une portée différente.

**Adresse et Valeur**

La valeur de gauche “***l-value***” et la valeur de droite “***r-value***” sont la partie gauche et la partie droite d’une affectation.

Par exemple, dans: x := y;

La valeur de gauche est ***l’adresse de x*** et la valeur de droite est la ***valeur de y***.

C’est un peu plus compliqué dans le cas des tableaux:

T[I\*2+1] := y;

l’adresse dépend de la valeur actuelle de i.

***Liaisons/Attachements***

Une liaison est l’association d’un attribut a un objet.

**Exemples**: le nom, le type et la valeur sont tous des exemples d’attributs de variables.

Les liaisons peuvent être établies à différent moments dans la vie d’un programme:

* + Au moment de la compilation (***compile time***)
  + Au moment du chargement (***load time***)
  + Au moment de l’exécution (***run time***)

D’autres distinctions plus précises incluent également d’autres moments: celui de la création d’un langage (***langages design time***), celui de son implémentation (***langage implémentation time***), et celui de la liaison à d’autre programme (***Link time***)

**Liaison d’attributs aux variables**

On dit qu’une liaison est ***statique*** si elle prend place avant l’exécution du programme et reste inchangée pendant cette exécution.

On dit qu’une liaison est ***dynamique*** si elle prend place pendant l’exécution du programme ou si elle peut changer au cours de cette exécution.

Bien qu’en réalité, la liaison physique d’une variable à une cellule de mémoire se défait et se refait à plusieurs reprises, nous ne nous intéressons pas à ces phénomènes de bas niveau.

**Exemple des liaisons de variables I**

* Variable 🡪 nom
* Variable 🡪 Adresse
* Variable 🡪 Type
* Compile Time –Déclarations
* Load time ou run time (e.g.,Pascal) – phénomène implicite
* compile time (e.g., Pascal); run time (e.g., Smalltalk) Déclarations

**Exemple des liaisons de variables II**

* Run time ou load time (initialization) -- Instructions, surtout, affectation.
* Compile time – Declarations
* Compile time – Placement des declarations.
* .Variable 🡪Valeur
* Variable 🡪 longueur de vie
* Variable 🡪 portée

**A propos de la durée de Vie**

L’allocation de mémoire pour un objet se fait ou bien au temps de chargement «***load time***» ou au temps d’exécution «***run time***»

Il existe, en fait, deux classes de variables:

Les variables ***statiques***: l’allocation est faite une seule fois, avant que le programme commence son exécution.

Les variables ***dynamiques***: l’allocation est faite pendant l’exécution du programme.

* + Allocation et Libération ***explicite***
  + Allocation et Libération ***implicite***

**Portée**

Déclarations et instructions sont groupées en ***blocs*** afin de:

Grouper les étapes d’une instruction non-élémentaire ensemble.

Interpréter les noms adéquatement.

La ***portée*** d’un nom N représente tous les endroits dans le programme où N réfère au même objet.

Les blocs peuvent être ***emboités***. Les noms introduits dans un bloc s’appellent les ***liaisons locales***. Un nom mentionné mais pas défini dans un bloc doit avoir été défini dans l’un des blocs de l’entourage.

**Blocs Anonymes et Blocs Nommés**

UN programme, une procédure ou une fonction sont des exemples de blocs nommés.

UN bloc anonyme EST comme une procédure sans nom ET appelée immédiatement et une seule fois.

Les blocs’ anonymes sont utiles lorsqu’un calcul EST nécessaire une seule fois ET SES variables ne sont nécessaires que dans Ce calcul: on ne veut pas les déclarer à l’exterieur de Ce bloc.

**Exemple d’un emboitement et ses representations diverses**

Veuillez assumer que les déclarations “forward” ont été faites de manière appropriée.

**Program P;**

Var X: integer;

**procedure** A;

Var Y: char;

**Procedure B;**

Var Z: Boolean;

Begin SB end;

**Begin** SA end;

**Procedure C;**

Var Z: integer;

Begin SC end;

**Begin** SP **end;**

**Cacher la Visibilité**

Si le meme nom X est défini dans un bloc environnant A et dans un bloc emboité, B (B ⊂ A), alors la visibilité du X défini en A est perdue dans le bloc B qui ne voit que le X défini en B.

**Exemple:**

**P**: x, y,

A

B

SP

**B**: y,

z

SB

**A**: x,

z

SA

**La portée statique vs. la portée dynamique**

La portée ***statique*** (ou ***lexicale***) permet de determiner l’usage de toutes les variables d’un programme de façon statique, sans exécuter le programme.

La portée dynamique cherche un nom dans la chaine des procedures appelées. Cette chaine considère les règles de visibilité mais pas l’emboitement.

La portée dynamique n’a pas d’avantage à part le fait qu’elle soit plus simple à implémenter.

**Program P;**

**var X: integer;**

**procedure A;**

**begin**

**X:= X+1;**

**print(X);**

**end;**

**procedure B;**

**var X:integer;**

**begin**

**X:= 40;**

**A;**

**end;**

**begin**

**X:=50;**

**B;**

**end;**

**Exemple Portée statique** ALGOL60 ADA …..

**Portée dynamique** APLSNOBOL4 Quelques versions de LISP

**Exemple de portée dynamique**

**Vérification de Type**

La vérification de type s’assure qu’un opérateur—lorsqu’il EST appliqué—reçoit des arguments qui le satisfait.

La vérification de type peut être faite au moment de la compilation ou a celui de l’exécution.

Il y a une erreur de type lorsqu’un argument d’un type non attendu est donné à une opération. Une erreur de type peut elle aussi être donnée au moment de la compilation ou au moment de l’exécution.

**Typage fort (Strong Typing)**

UN langage de programmation a un typage fort si toutes les erreurs de type peuvent être découvertes au moment de la compilation.

Ceci n’est pas facile à faire meme dans UN langage très strict tel que le Pascal.

Une définition moins stricte dit qu’un langage a un typage fort si toutes SES erreurs peuvent être découverte, préferemment au moment de la compilation.

Aucun langage populaire n’inclut de typage fort parfait. Il y a toujours des infractions. ML a un typage parfaitement fort, mais il n’est pas populaire!!

**Conversion et Coercition**

UN typage trop stricte peut, en effet, ne pas être très pratique:

Par exemple, Il devrait être permis de multiplier un entier avec un réel.

Certains langages permettent la conversion: e.g.: x:= float(n) \* 3.14;

Et d’autres permettent les coercitions de types: la forme d’un argument est changée automatiquement lorsque son type est légèrement inappropriée.

C utilise beaucoup la coercition.

**Compatibilité de Types**

Deux objets de type primitif différent peuvent être comparés seulement si l’un des types peut être converti dans l’autre. Exemple: entier 🡪 reel.

Compatibilité ***par nom***: deux variables ont des types compatibles seulement si elles sont dans la meme desclaration ou si elles sont dans des declarations qui utilisent le meme nom de type.

Compatibilité ***par structure***: deux variables ont des types compatibles si leur types ont des structures compatible.

La compatibilité par nom est facile à implementer mais est très restrictive. La compatibilité par structure est difficile à implementer mais très flexible.

**Types de Données**

**Plan du Cours**

* Le Concept d’un Type de Données
* Types de données primitifs et semi-primitifs
* Types de données Enumératifs et de Sous-Gamme
* Types de données Structurées: Tableaux, Enregistrements, Unions.
* Ensembles
* Pointeurs

**Le Concept d’un Type de Données**

* Dans les premiers langages (e.g., Fortran I), toutes les

structures de données étaient définies par des structures de données de base supportées par le langage.

* Par la suite, d’autres langages tels que le Cobol et le PL/1 ont rajouté des types de données plus précis.
* En Algol 68, une autre approche a été proposée: quelques types de données de base sont présents ainsi que quelques opérateurs permettant au programmeur de définir ses propres types de données.
* Plus tard, ceci a donné lieu à l’idée de type abstrait de données ADT.
* Un type de données n’est pas qu’un ensemble d’objets. On doit aussi considérer toutes les opérations qui peuvent être appliquées à ces objets.
* Une définition de type complète doit donc inclure une liste de toutes les opérations ainsi que leurs définitions
* Il faut aussi se rappeler qu’un type de données n’est qu’une abstraction: seul le bit et l’octet ont une existence tangible.
* Néanmoins, les types de données ***primitifs*** ne sont pas très éloignés du hardware.

**Types de Données Primitifs: Les Entiers**

* La plupart des ordinateurs permettent maintenant l’existence d’entiers de tailles différentes et ceci est reflété dans les langages de programmation qui parfois permettent: des entiers sans signes, des entiers avec un signe, des entiers courts, normaux ou long.
* Les entiers sont représentés directement dans l’ordinateur par une chaîne de bits.

**Types de Données Primitifs: Les Points Flottants**

* Les Points Flottants représentent les réels jusqu’à un certain degré de correction.
* Comme dans la notation scientifique, ils sont représentés par une fraction (mantisse) et une puissance (exponent).
* La plupart des langages incluent deux types de points flottants: float et double.
* Le float prend d’habitude 4 octets de mémoire alors que les doubles prennent deux fois la place des floats et donne au moins deux fois le nombre de bits de fraction.

**Types de Données Primitifs: Les Booléens**

* Le type booléen est le plus simple de tous: il n’a que deux valeurs: true ou false.
* Bien qu’une variable booléenne puisse-t-être sauvegarder dans un seul bit de mémoire, elle est souvent sauvegardée dans un octet car cela rend l’accès plus efficace.
* Les booléens peuvent être implémentés avec des entiers, mais ils rendent la lecture plus facile.
* Pas tous les langages supportent les booleens: il n’y en a pas en PL/1 ou en C.

**Types de Données Primitifs: Les Caractères**

* Les caracteères sont sauvegardés en mémoire avec des codes numériques, tels que l’ASCII.
* L’ASCII utilise les valeurs 0 ..127 pour encoder 128 caractères différents.
* Néanmoins, des codes plus complets, existent aussi. L’Unicode qui incluent les caractères de la plupart des langages du monde est un ensemble de caractères représenté avec 16 bits.
* Java est le premier langage populaire à utiliser l’Unicode.

**Types de Données Semi-Primitifs: Les chaînes de caractères**

* Une chaîne de caractères est une séquence de caractères. Elle peut être:
  + Un type de donnée spécial (dont les objets peuvent être décomposés en caractères): FORTRAN, BASIC
  + Un tableau de caractères: Pascal, ADA
  + Une liste de caractères: Prolog
  + Des caractères sauvegardés consécutivement: C
* **Forme Syntaxique:** Pascal n’a qu’un type de guillemets. Ada en a deux: ‘a’ est un caractère et “a” est une chaîne de caractères.
* Opérations sur les chaînes de caractères lorsqu’elles sont définies comme un propre type de données:
* string x string🡪 string concaténation
* string x entier x entier🡪 string sous-chaîne
* string 🡪 caractères décompose la chaîne en un tableau ou une . liste de caractères
* caractères 🡪 string convertit un tableau ou une Liste en une chaîne
* La longueur permise des chaînes est un problème de conception du langage: en Pascal, Ada et Fortran, les chaînes sont de taille fixe. En C et Java, elles sont a longueur variable.
* Un problème avec les chaînes a longueur variable: l’allocation d’un espace possible cause du gâchis. Par contre, l’allocation caractère par caractère doit être faite trop fréquemment. Une stratégie efficace est d’allouer des blocs de 128 caractères (par exemple). Toutes les chaînes comprise entre 1 et 128 caractères occupent 1 bloc. Toutes celles entre 129 et 256, deux, etc.

**Types de Données Énumératifs**

* Il est possible de déclarer une liste de constantes symboliques à traiter littéralement (ceci veut dire qu’elles ne représentent pas d’autres valeurs tel que “const pi = 3.14” le fait).
* Implicitement, on donne également un ordre à ces constantes symboliques.

**Exemple:** type day=(mo, tu, we, th, fr, sa, su) est tel que mo<tu<we<…

* Pascal a trois opérations pour chaque type numératif:
  + succ: successeur eg, succ(tu)=we
  + pred: predecesseur, eg, pred(su)=sa
  + ord: position dans le type: ord(mo)=0; ord(we)= 2
* En Ada, la liste est un peu plus complète: succ(mo)=tu, pred(tu)=mo, pos(mo)=0, val(3)=th, FIRST et LAST.
* Est il permis d’avoir une constante symbolique définie dans plus d’un type? En Pascal, non. En Ada, oui.

**type** stoplight **is** (red, orange, green);

**type** rainbow **is** (violet, indigo, blue, green, yellow, orange, red)

* Les descriptions qualifiées enlèvent la confusion: stoplight’(red) ou rainbow’(red)
* L’implémentation des types énumères: constantes c1, .., ck 🡪 entiers 0, …, k-1
* Néanmoins, ils améliorent la lisibilité.

**Types de Sous-Gamme**

* Il est possible de définir un type qui représente une sous-gamme de valeurs
* **Exemple** en Pascal: **type** capitales = ‘A’..’Z’; toutes les opérations qui s’appliquent aux caractères s’appliquent également au type capitales qui est un sous-type des caractères.
* En Ada il y a des types dérivés et des sous-gammes. Ce n’est pas la même chose:
* **type** petits\_entiers\_derives **is new** integer **range** 1..100;
* **subtype** petits\_entiers\_sous\_gamme **is** integer **range** 1..100;
* Ces deux types héritent des opérations sur entiers.

Néanmoins les variables de type petits\_entiers\_derives ne sont pas compatibles avec les entiers.

**Types de Données Structures: Les Tableaux**

* Un tableau représente l’application: type\_d’index 🡪 type\_de\_composante
* Le type d’index doit être un type à gamme non continue: entier, caractère, énumération. Dans certains langages ce type est spécifié indirectement: A(n) en C signifie 0..N, alors qu’en Fortran c’est 1..N.
* Il y a en général peu de restrictions sur le type de composante (ils peuvent même être des tableaux, des procédures ou des fichiers).
* La forme des référence est A[I] en Algol, Pacal et C et A(I) en Fortran et Ada.
* Les tableaux multi-dimensionnels peuvent être définis de deux façons différentes:
* type\_d’indexe1 x type\_d’indexe2 🡪 type\_de\_composante, ou
* type\_d’indexe1 🡪 (type\_d’indexe2 🡪 type\_de\_composante)
* La première définition correspond à des références telles que A[I,J] alors que la deuxième correspond à des références telles que A[I][J]

***Opérations sur les tableaux:***

* + Sélection d’un élément A[I]
  + Sélection d’une “tranche” d’éléments A(1:10) ou même A(2:10:2). Possible en Fortran 90 et de manière plus restreinte en Ada.
  + Affectation d’un tableau complet a un autre: A := B; il y a une boucle implicite dans cette instruction.
  + Calcul d’expressions avec des tableaux entiers: A:= A + B. Fortran 90 implémente ces opérations ainsi que APL qui en ajoute d’autres telles que les transposition, les inversions les “inner products”…

***Attachement d’indexes.*** 4 types de tableaux:

* + Tableaux Statiques: l’indexe a un attachement statique et l’allocation de mémoire est statique.
  + Tableaux “Fixed Stack-Dynamic”: l’indexe a un attachement statique mais l’allocation de mémoire est faite au moment de l’élaboration des déclaration pendant l’exécution.
  + Tableaux “Stack-Dynamic”: l’indexe a un attachement dynamique et l’allocation de mémoire est dynamique mais cet attachement et allocation sont fixes une fois faits.
  + Tableaux “Heap-Dynamic”: l’indexe a un attachement dynamique et l’allocation de mémoire est dynamique et ces attachements et allocations peuvent changer pendant l’exécution.

***Initialisation de Tableaux***

* Beaucoup de langages permettent l’initialisation de tableaux dans la déclaration:
* C int vector [] ={10,20,30};
* Ada vector: array(0..2) of integer := (10, 20, 30);
* Autre type d’affectation en Ada: temp is array (mo..fr) of -40 .. 40; T: temp; T:= (5,12,8,30,25); T:= (mo => 5,we =>8,tu=>12, fr=>25,others=>30); T:=(5,12,8,fr=>25,th=>30);
* Attributs de tableaux en Ada:
  + TM: temp; TM’FIRST; TM’LAST; TM’LENGTH; TM’RANGE;

***Implémenation de Tableaux:*** Comment sauvegarder les tableaux et accéder à leurs éléments?

* Pendant l’exécution un tableau est représenté par un ***descripteur de tableau*** qui nous donne:
  + Le type de l’indexe
  + Le type des composantes
  + L’adresse de base du tableau (i.e., des données)
  + La taille d’une composante
  + Les limites inférieures et supérieures de l’indexe

***Implémentation de Tableaux Multi-dimensionnels***

Un tableau multi-dimensionnel est représenté par un descripteur incluant plusieurs paires de limites inférieures et supérieures.

Les tableaux multi-dimensionnels doivent aussi être représentés en mémoire qui n’est que uni-dimensionnelle. Deux approches: **row-major** (le deuxième indexe change le plus vite) ou **column-major** (le premier indexe change le plus vite).

***Exemple d’accès à une composante:***

* A: array [LOW1 .. HIGH1, LOW2..HIGH2] of ELT; la taille de ELT est SIZE.
* L’illustration est pour une organisation row-major (ce serait similaire pour column-major). On assume que l’adresse de base du tableau (i.e., A[LOW1, LOW2]) est LOC.
* On peut calculer l’adresse de A[I,J] comme ceci:

**(I-LOW1)\* ROWLENGTH \* SIZE +**

**(J-LOW2)\*SIZE+LOC**

**Avec ROWLENGTH=HIGH2-LOW2+1**

* Quoique les tableaux n’existent pas en Scheme, ML et Prolog, ils peuvent être simulés par des listes

**Types de Données Structurées:   
Les Enregistrements**

* Un enregistrement est une collection hétérogène de champs (ou composantes). Les tableaux, par contre, sont homogènes.
* Les enregistrements sont supportés par la plupart des langages importants: Cobol (qui les a introduits), Pascal, PL/1, Ada, C (appelés structures), Prolog, C++.
* Les champs ont des noms plutôt que des indices et on ne peut pas itérer sur ces indices.
* Un champ est indiqué par un nom de variable qualifié, mais le nom de la variable peut être omis si la variable de référence est déjà connue (**“with”** en Pascal).

***Opérations sur les enregistrements:***

* Sélection d’une composante par nom de champ.
* Construction d’un enregistrement à partir de ses composantes. Ou bien champ par champ ou bien d’un seul coup (à partir d’une constante structurée).
* Affectation d’enregistrements complets
* Comparaison entre deux enregistrements (seulement une vérification d’égalité. Il n’y a pas d’ordre standard pour les enregistrements).
* Ada permet des valeurs de défaut pour les champs d’enregistrements.
* Il n’y a pas beaucoup de restrictions sur les types des champs. Toute combinaison d’enregistrement et de tableau (quelque soit sa profondeur) est habituellement permise. Un champ peut même être un fichier ou une procédure.
* Les champs d’un enregistrement sont sauvegardés dans des locations adjacentes de mémoire. Lorsque les champs sont accédés, il faut se souvenir que comme la taille de chaque champ peut-être différente, la méthode d’accès est un peu différente de celle décrite pour les tableaux: chaque champ a son propre “offset” relative au début de l’enregistrement

**Types de Données Structurées:   
Les Enregistrements IV**

* En Prolog, les enregistrements indiquent tous leurs types et composantes:
* date(day(15), month(10), year(1994))
* person(name(fname(“Jim”), lname(“Berry”)), born(date(day(15),month(10),year(1994)), gender(m))
* On peut simplifier cette notation si on peut s’assurer d’une utilisation correcte:
* date(15,10,1994)
* Person(name(“Jim”,”Berry”), born(date(15,10,1994),m)

**Types de Données Structurées:  
 Union Discriminées de Types**

* Une union de type signifie un ensemble d’objets provenant de ces types ainsi que des opérations sur ces objets.
* Bien que les types peuvent être incompatible, cela pose des problèmes.
* Néanmoins, bien que des types incompatibles ne peuvent pas être attachés à un objet au même moment, ceci est possible à différents moments de la vie de l’objet
* Une union discriminée est une union dans lequel le type actuel de l’objet peut toujours être connu.
* Ceci peut être implémenté en ajoutant une composante spéciale à une union: un “tag” ou “discriminateur” qui indique le type de l’objet.
* Les opérations sur les enregistrements a variante seront correctes si le champ discriminateur est vérifié.
* S’il n’est pas bien vérifié, il y aura des problèmes car l’utilisation de variantes peut conduire a des situations inconsistantes lorsque les champs reçoivent de affectations individuelles.
* Un langage comme l’Ada rend ces inconsistances impossible car il ne permet les affectations qu’en agrégation.

***Implémentation d’Unions Discriminées:***

Les unions discriminées sont implémentées en donnant la même adresse a chaque variante possible.

Une taille de mémoire suffisante à accommoder la variante la plus large est choisie pour toute les variantes.

**Ensembles**

Les ensembles ne sont permis qu’en Pascal et en Modula-2.

Ils ne sont possibles qu’avec des types basés sur les entiers et avec très peu d’objets (moins de 100 dans beaucoup d’implémentations de Pascal).

**Exemple**

**type** charset= set of char;

**var** set1=charset;

vowels=[‘a’,’e’,’I’,’o’,’u’];

On peut écrire **if** ch **in** vowels…

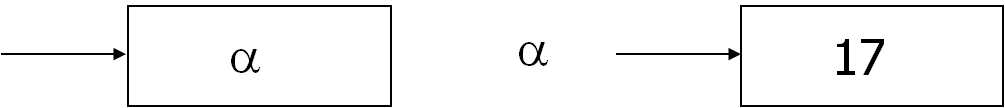
Plutôt que **if** (ch=‘a’) **or** (ch=‘e’) **or**…

Il y a des opérations sur les ensembles: affectation, union, intersection, différence, appartenance, inégalité inclusion

Implémentation: avec un “bit map”

**Pointeurs**

* Une variable de type pointeur a comme valeur une adresse (et une adresse spéciale (nil ou null) lorsqu’elle n’a pas de valeur).
* Ils sont utilises principalement pour construire des structures de taille et de formes imprédictibles (listes, arbres, graphes) à partir de petits fragments alloués dynamiquement pendant l’exécution.
* Un pointeur à une procédure est possible mais normalement on a des pointeurs à des données simples ou composées.
* Une variable est composée d’une adresse, d’un type de données et d’une valeur. Mais c’est une ***variable anonyme*** car elle n’a pas de nom.
* L’accès a la valeur d’une telle variable se fait en déreférenciant le pointeur:



* Valeur(p)= p^ = α et Valeur(p^)= α^ =17
* Comme avec des variables normales, dans “p^ :=23;”, on veut dire l’adresse de p^ (ou bien, la valeur de p). Dans “m=p^;”, on veut dire la valeur de p^.
* Une variable de type pointeur est déclarée explicitement et a une portée et une durée de vie comme d’habitude.
* Une variable anonyme n’a pas de portée (car elle n’a pas de nom) et sa durée de vie est déterminée par le programmeur.
* Une telle variable est créée (dans la partie de la mémoire spéciale qui s’appelle le “heap”) par le programmeur. E.g.: new(p) en Pascal; p=malloc(4) en C.
* Elle est aussi détruite par le programmeur: dispose(p) en Pascal; free(p) in C.
* Si une variable anonyme existe à l’extérieur de la portée d’une variable explicite de type pointeur, on a su “garbage” (un objet perdu).
* Si, par contre, une variable anonyme a été détruite à l’intérieur de la portée d’une variable explicite de type pointeur, on a une “dangling” référence.
* Garbage Collection est le processus par lequel la mémoire inaccessible est récupérée. Ce processus est complexe et cher. Il est essentiel dans les langages dont l’implémentation dépend elle-même de pointeurs comme le LISP et le Prolog.
* En PL/1 les pointeurs n’ont pas de type. En Pascal, Ada et C ils sont déclarés par rapport a un type, ce qui veut dire qu’un pointeur déférencié (^p ou \*p) a un type fixée.
* Les opérations sur les pointeurs sont très riches en C.

**Instructions de contrôle**

* Instructions simple
* Instructions structurées de base
* Séquence
* Sélection
* Itération
* Instruction jump

**Instructions simple  
dans les langages impératifs**

Celles-ci sont des opérations atomiques (tout ou rien):

* assignation,
* Instruction vide,
* Appel à une procédure,
* exit, next, break, continue
* goto (jump).

Un bloc est aussi une opération tout-ou-rien.

**Instructions structurées**

Trois mécanismes fondamentaux nous permettent de regrouper des instructions en une instruction structurée.

* séquence, ou instruction composée: {S1 S2}
* sélection, ou instruction conditionnelle: if (C) S1 else S2
* itération, ou instruction de boucle: while (C) S

Toute autre instruction de contrôle structurée est dérivée de ces trois mécanismes de base.

* **if (C) S** ≡ **if (C) S else {}**
* **do S while (C)** ≡ **S  
   while (C) S**
* **switch (i) { if (i == C1) S1  
  case C1: S1 …** ≡ **else …**

et ainsi de suite.

**Séquence (1)**

Langages Mécanismes

Algol, Pascal, Ada, ... begin ... end

C, Java, Perl { ... }

Fortran IV rien

Prolog implicite

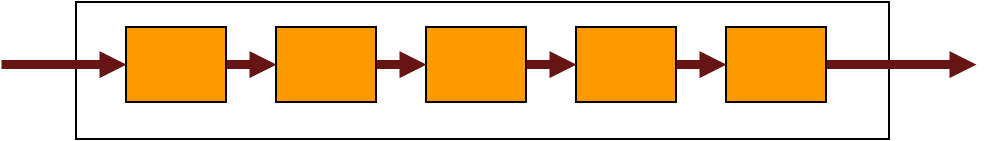
a :- b, c, d.  
veut dire évaluer b, puis c, puis d.

Scheme (begin ...)

**Séquence (2)**

Une instruction composée est traitée comme une instruction simple. Ceci est base sur le principe de l’abstraction:

On peut « faire abstraction » de la structure interne.





**Sélection**

La structure if-else existe dans presque tout les langages de programmation (Prolog est l’exception importante). Modula et Ada sont les premiers à avoir correctement entoures le if-then-else avec 4 mots-clef autours de 3 éléments: **if C then S1 else S2 end if**

Il y a aussi la sélection imbriquée if-elsif-...-else dans Ada:

**if C1 then S1  
elsif C2 then S2  
......  
elsif Cn then Sn  
else Sn+1 end if**

**Formes spéciales de sélection en Fortran IV**

GO TO calculé.

GO TO (label1, ..., labeln), expression

L’expression retourne un entier et on va au label correspondant.

GO TO assigné.

ASSIGN labeli TO variable

GO TO variable(label1, ..., labeln)

**Formes spéciales de sélection (2)**

L’instruction switch de C et Java est inspirée des GO TO calculé.

**switch(expression){  
 case const1: S1;  
 ...  
 case constn: Sn;  
 default: Sn+1;}**

Après que Si est exécuté, l’exécution se poursuit en cédant le contrôle aux cas suivant: Si+1 est le suivant à être exécuté.

Ceci peut être évité avec l’instruction **break**.

L’instruction case de Pascal et Ada: chaque cas est séparé, l’exécution des autres cas ne se poursuit pas après qu’un cas est exécuté. En Ada:

**case expression is  
 when constantList1 => S1;  
 ...  
 when constantListn => Sn;  
 when others => Sn+1;  
end case;**

La sélection, en Prolog, se fait plutôt selon la réussite, ou l’échec de l’évaluation de prédicats.

La sélection est implicite dans le backtracking: si il y a réussite, stop; sinon, essaie un autre choix.

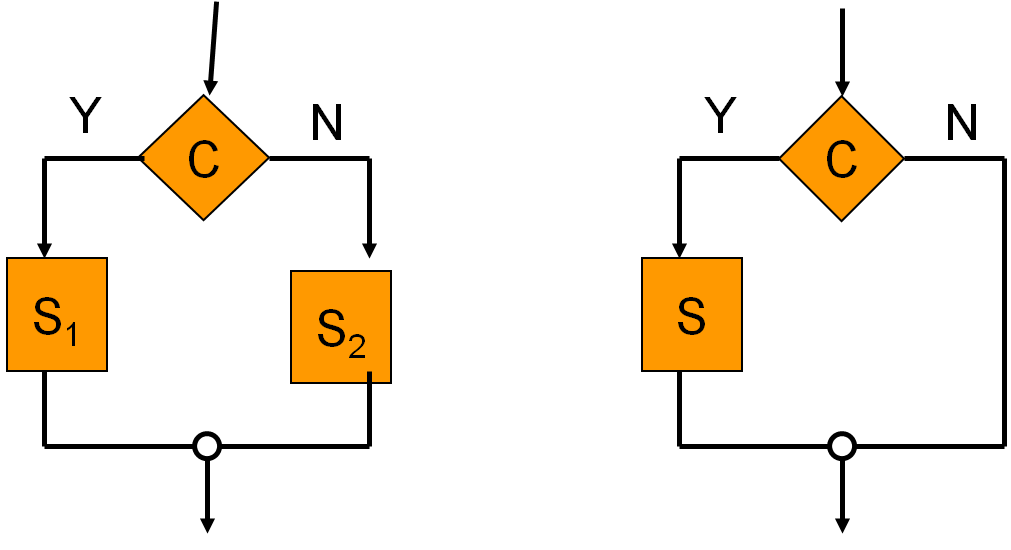
**union( [Elem | S1], S2, S1\_S2 ) :-  
 member( Elem, S2 ),  
 union( S1, S2, S1\_S2 ).**

**union( [Elem | S1], S2, [Elem | S1\_S2] ) :-  
 \+ member( Elem, S2 ),  
 union( S1, S2, S1\_S2 ).**

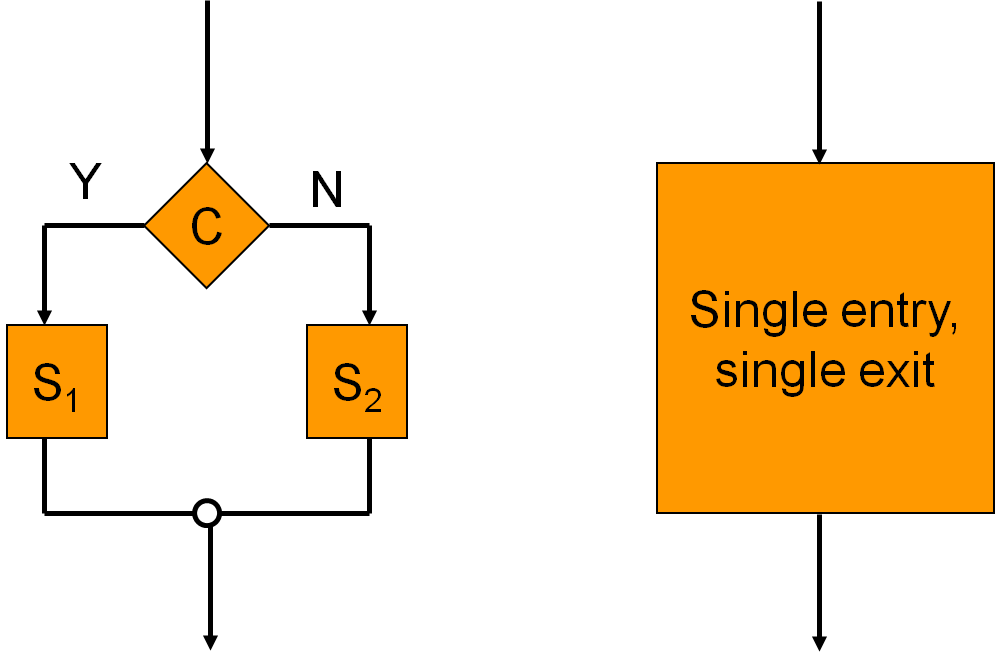
**union( [], S2, S2 ).**

**Représentation avec des graphes organigramme**

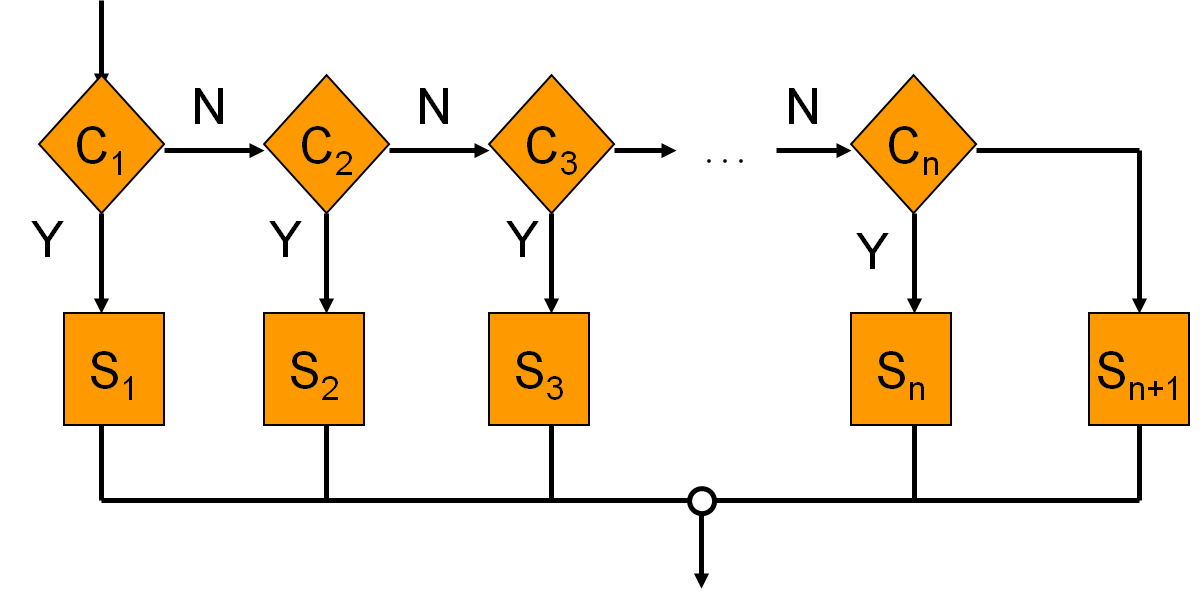
**if–then–else if–then**



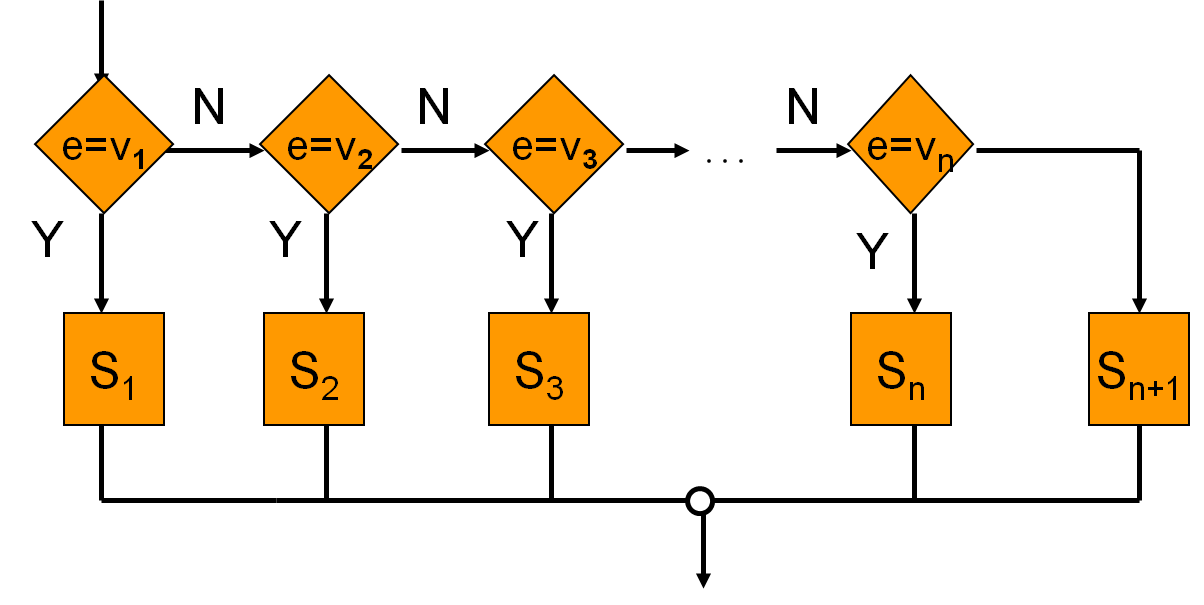
Le principe d’abstraction: **if ( C ) S1 else S2** est une expression simple.



if–then–elsif-…elsif-then-else



case e of v1: S1; ... else Sn+1 end

**Itération**

Variations: itération pré-test ou itération post-test.

**while C do S** Pascal

**repeat S until C**

**while (C) S** Java

**do S while (C)**

**while C loop S end loop;** Ada

(pas d’itération post-test)

En Ada, le préfixe **while C** est une extension de l’instruction itérative de base:**for i in range**

Une telle boucle doit donc être terminée à partir de l’intérieur.

La terminaison forcée arrête l’itération:

**exit;** inconditionnelle **exit when C;**

Le préfixe while est une abréviation. La boucle: **while C loop S end loop;**

est équivalente à

**loop  
 exit when not C;  
 S  
 end loop;**

**Exemple: exit**

**UM :S= 0;**

**loop**

**get(X);**

**exit when X = 0;**

**SUM := SUM + X;**

**end loop;**

Simple, et plus intuitif

**SUM := 0;**

**get(X);**

**while X /= 0 loop**

**SUM := SUM + X;**

**get(X);**

**end loop;**

La condition est inversée, **get(X)** apparais deux fois

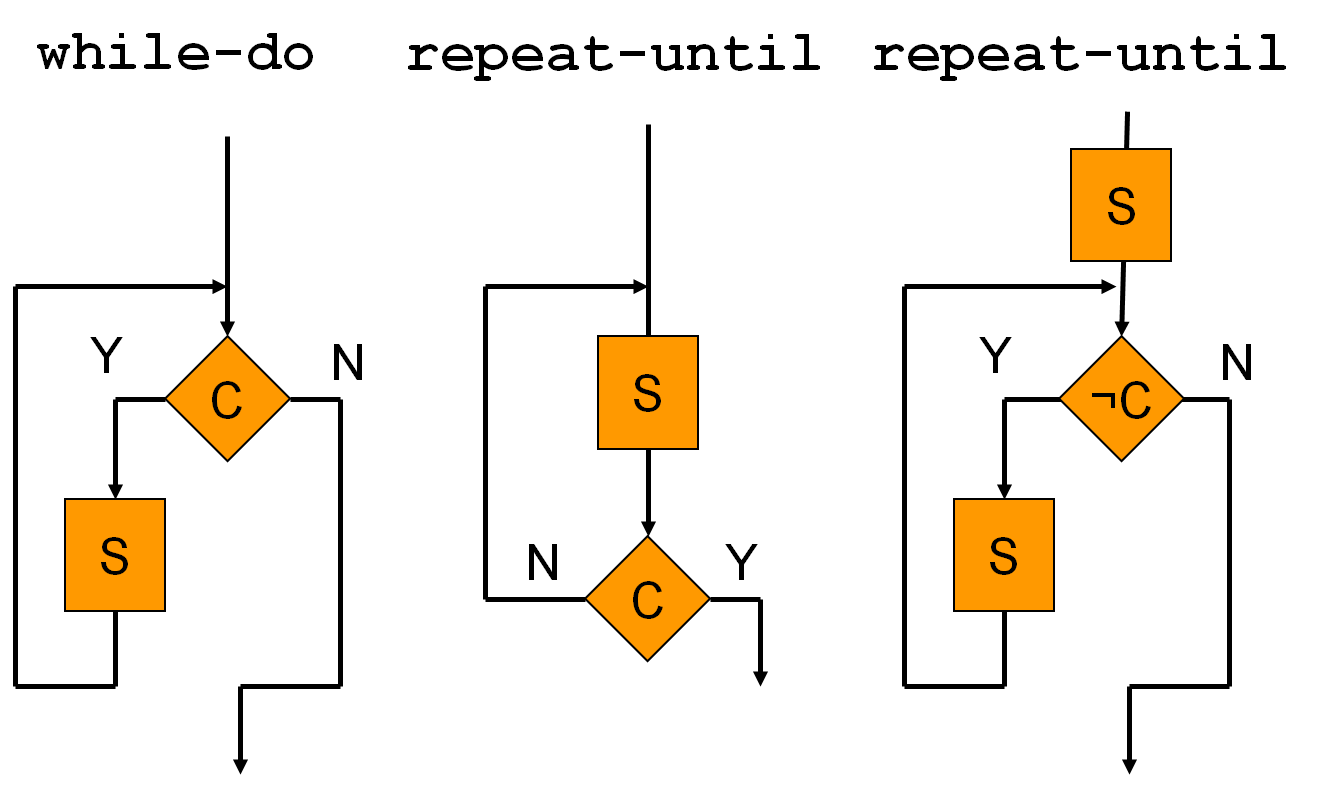
**Représentation avec des graphes**

Certain langages restreignent l’usage du **goto** (ne permettent pas son usage à l’intérieur d’une itération ou d’une sélection) et font ainsi en sorte qu’il est difficile de l’utiliser.

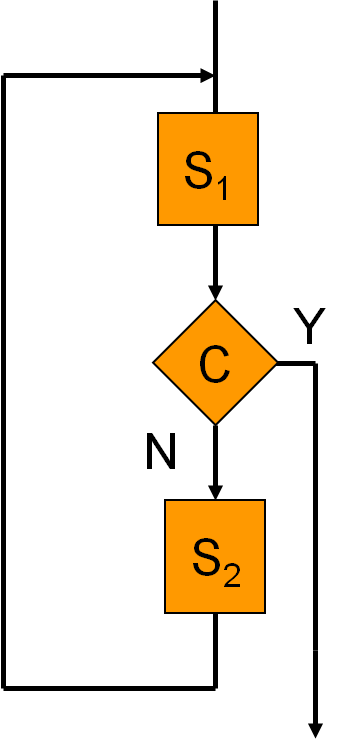
Ada fait en sorte que les étiquettes soient facile à voir dans un programme (ce qui décourage leur usage par des programmeurs voulant les utiliser malgré qu’elles soient interdite par leur patr

**SUM := 0;  
 loop  
 get(X);  
 if X = 0 then goto DONE; end if;  
 SUM := SUM + X;   
 end loop;  
<<DONE>>  
 put(SUM);**

**goto** peut causer des problèmes quand il fait sortir d’un bloc qui n’est ainsi pas termine de façon normale.



**loop - exit - end loop**



**Boucles for**

Les boucles for ("contrôlées par un compteur") sont apparus avant les boucles while ("contrôlées par une condition"), et sont moins générales.

**DO 1000 var = lo, hi** Fortran IV

**...**

**1000 CONTINUE**

**DO label var = lo, hi, incr**

**for var := expr do S** Algol 60

**for var := low step incr until high do S**

**for var := expr while C do S**

Les itérateurs peuvent être combinés:

**for i := 0, i+1 while i ≤ n do S(i)**

**for var in range** Ada  
**loop S end loop;**

**for var in reverse range  
loop S end loop;**

**for (e1; e2; e3) S** C, Java, Perl

Qu’est-ce que ceci? **for (;;) S**

L’itération en Prolog et en Scheme est uniquement obtenu avec la récursivité. La même approche peut aussi être utilisée dans la plupart des autres langages.

**Jump (l’instruction goto)**

Le transfert sans contrainte de l’exécution est le seul mécanisme disponible dans les langages de bas niveau — mais c’est un outil très général. La sélection à une seul branche et le **goto** nous permettent d’exprimer tout les autres structures de contrôle.

Le mécanisme du jump est dangereux, il peut restreindre la lisibilité et devrait être évité — les structures de contrôle plus avancées sont peuvent efficacement accomplir toutes les tâches habituelles, et la plupart des tâches moins fréquentes.

Certain langages restreignent l’usage du **goto** (ne permettent pas son usage à l’intérieur d’une itération ou d’une sélection) et font ainsi en sorte qu’il est difficile de l’utiliser.

Ada fait en sorte que les étiquettes soient facile à voir dans un programme (ce qui décourage leur usage par des programmeurs voulant les utiliser malgré qu’elles soient interdite par leur patron). **SUM := 0;  
 loop  
 get(X);  
 if X = 0 then goto DONE; end if;  
 SUM := SUM + X;   
 end loop;  
<<DONE>>  
 put(SUM);**

**goto** peut causer des problèmes quand il fait sortir d’un bloc qui n’est ainsi pas termine de façon normale.