IFT2030: Concepts des langages de programmation

Marc Feeley (3341)

`http://www.iro.umontreal.ca/~feeley/cours/ift2030`

Manuel: "Programming Languages – Concepts and Constructs" (Ravi SETHI)

```
                Calendrier approximatif

 #1: Historique, syntaxe des langages
 #2: Programmation impérative (C)
 #3: Compilation
 #4: Types, représentation de donnée, gestion mémoire
 #5: Programmation fonctionnelle (Scheme)
 #6: Traitement de liste, fonctions d'ordre supérieur
 #7: Procédures, macros, variables et environnements
 #8: INTRA, Interprétation
 #9: Semaine de lecture
#10: Programmation orientée-objet (Scheme/C++)
#11: Programmation concurrente (Scheme/Java)
#12: Synchronisation, client-serveur
#13: Programmation logique (Scheme/Prolog)
#14: Unification, retour-arrière
```


Aperçu du cours

- Il existe un très grand nombre de langages de programmation (> 2000)

- Les buts du cours sont

  1. de vous permettre d'utiliser la majorité de ces langages efficacement

  2. d'étudier l'implantation des langages de programmation (par des compilateurs, etc), ce qui permettra de mieux vous en servir

- C'est possible car les langages partagent beaucoup de concepts

  1. nombre limité de **syntaxes**

  2. nombre limité de **styles de programmation** (i.e. approche d'organisation d'un programme pour effectuer un certain calcul)

- Des langages représentatifs de ces styles seront utilisés pour étudier les concepts (C, Scheme, Prolog, ...)

Historique des langages de progr. (1)

=> Débuts (< 1955): **langage machine**

- Seul langage compris par la machine (directement)

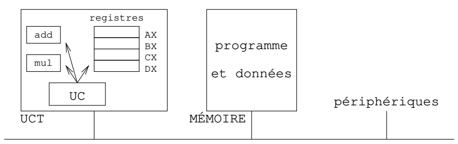
- L'unité centrale de traitement (UCT) et la mémoire sont séparées; la mémoire contient le programme et les données à traiter ("architecture de Von Neumann" = même mémoire, "architecture Harvard" = mémoires séparées)

  – Type d'instructions machine:

    * transfert de donnée entre registres, mémoire et périphériques

    * arithmétique (addition, multiplication, etc)

    * contrôle (poursuivre l'exécution à un autre point du programme, conditionnellement ou inconditionnellement)



Historique des langages de progr. (2)

- L'UC lit, décode puis exécute les instructions une à une

- Les instructions sont encodées par des chaînes de bits (typiquement des multiples de 8, 16 ou 32 bits)

- Exemple sur famille x86: additionner 2 entiers

```
binaire

10001011   \
01000101    | lire 1er entier dans registre AX
00001010   /
00000011   \
01000101    | lire 2ème entier et ajouter à AX
00010100   /
```

- Avantage: contrôle total sur la machine

- Désavantages: inintelligible pour les humains, différent pour chaque type de machine

Historique des langages de progr. (3)

=> **langage d'assemblage** ("assembler")

- Essentiellement une représentation textuelle symbolique du langage machine

- Même exemple:

```
mov 10(%ebp),%ax  # lire 1er entier dans registre AX
add 20(%ebp),%ax  # lire 2ème entier et ajouter à AX
```

- Un peu plus lisible mais:
  - beaucoup de code à écrire pour faire peu de choses
  - le programmeur doit être constamment conscient des caractéristiques de la machine (nb. registres, taille des registres et mémoire, irrégularités du jeu d'instruction, organisation des pipelines, etc)

- Ce sont des **langages de bas niveau**: où il faut s'occuper de détails qui ont plus rapport avec la machine qu'avec le calcul à faire

Historique des langages de progr. (4)

=> **langage de haut niveau** ("high-level language")

- Langage qui facilite la tâche de programmation en cachant les détails se rapportant à la machine

- Même exemple en C: `X+Y`

- Avantages:
  1. notation plus familière (facilite écriture, analyse, modification et entretien du programme)
  2. fiabilité (Dijkstra: *"Il s'agit d'organiser les calculs de telle manière à ce que nos moyens limités nous permettent d'en garantir l'effet."*, Hoare: *"Il y a deux façons de construire un logiciel: avec un design si simple qu'il n'y a évidemment pas de fautes et avec un design si compliqué qu'il n'y a pas de fautes évidentes. La première méthode est de loin la plus difficile."*)
  3. portabilité (habileté d'un programme d'être utilisé sur d'autres machines avec peu de changements)
  4. réutilisation de librairies de code
  5. détection automatique de certaines erreurs

Historique des langages de progr. (5)

- Les langages de haut niveau permettent au programmeur d'être **plus productif** (temps requis pour réaliser/entretenir un programme)

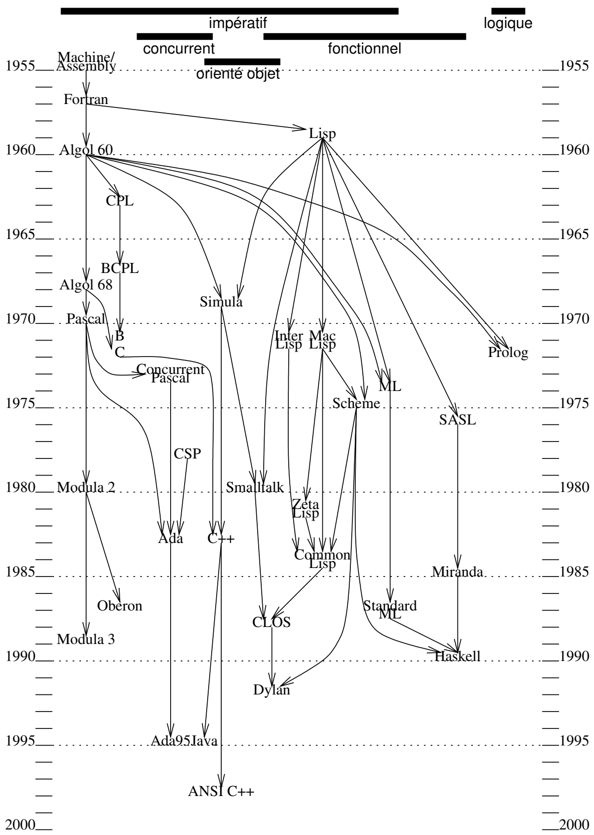
- Tout est relatif: un langage qui en 1960 était de "haut niveau" peut bien paraitre de "bas niveau" aujourd'hui

  machine $<$ assembleur $<$ FORTRAN $<$ C $<$ Java

- Le **niveau d'abstraction** est un continuum à plusieurs dimensions

- Un langage peut être de plus haut niveau qu'un autre sur un certain point mais de plus bas niveau sur un autre point
  1. Java (gestion mémoire automatique, point flottants à précision fixe)
  2. Ada (gestion mémoire manuelle, point flottants à précision variable)

Généalogie des langages de progr.



Styles de programmation (1)

=> **Programmation impérative**

- Calcul = séquence d'actions
- Exemple: lire X, lire Y, calculer X*Y, mettre résultat dans Z
- 1957: **FORTRAN** (FORmula TRANslation), populaire rapidement car plus facile que l'assembleur et code presqu'aussi rapide
- 1960: **Algol 60** (ALGOrithmic Language), surtout utilisé pour exprimer des algorithmes pour publication, a donné lieu à la "famille ALGOL"
- 1970: **Pascal**, sous-ensemble de Algol, très facile à compiler, Wirth créa ensuite Modula-2 et Oberon
- 1972: **B** et **C**, assez bas niveau pour écrire un système d'exploitation (créés pour implanter UNIX)
- 1983: **Ada**, créé pour remplacer tous les langages utilisés par le DoD, fusion de plusieurs langages

Styles de programmation (2)

=> **Programmation fonctionnelle**

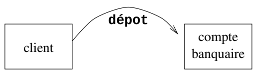
- Calcul = fonction (au sens mathématique)

- Un programme peut être construit en composant des fonctions plus primitives
- 1959: **Lisp** (LISt Processor), conçu pour le traitement de listes, surtout utilisé en IA, premier langage avec fonctions récursives, gestion automatique de la mémoire, etc.
- 1975: **Scheme**, variante épurée de Lisp, simple et rapide
- 1984: **Common Lisp**, fusion de plusieurs Lisps, énorme
- 1974: **ML**, syntaxe à la Algol + typage statique
- 1990: **Haskell**, fonctionnel "pur"

Styles de programmation (3)

=> **Programmation orientée objet**

- Calcul = objets qui interagissent en s'envoyant des messages



- 1968: **SIMULA**, conçu pour la simulation, introduit le concept de "classe", coroutines
- 1980: **Smalltalk**, typage dynamique, tout est un objet, messages entre objets
- 1983: **C++**, extension stricte de C avec aspects OO de SIMULA
- 1995: **Java**, variante épurée de C++, sécuritaire et portable

Styles de programmation (4)

=> **Programmation logique**

- Calcul = preuve logique

```
  César est humain
  Tous les humains sont mortels
---------------------------------
  César est mortel
```

- 1972: **Prolog** (PROgrammation LOGique), créé pour le traitement de la langue naturelle, applications en IA

=> **Programmation concurrente**

- Calcul est divisé en plusieurs tâches qui s'exécutent simultanément en coordonnant leurs actions
- Cela permet le calcul distribué (p.e. WWW) et parallèle (p.e. prévision météo)
- Note: un langage peut permettre plusieurs styles de programmation (ils ne sont pas mutuellement exclusifs)

Choix d'un langage de progr. (1)

Facteurs qui influencent le choix d'un langage pour une certaine tâche:

- **Niveau d'abstraction**
  - doit être adapté à la tâche (p.e. C vs Prolog)
- **Simplicité conceptuelle**
  - nombre et régularité des concepts
  - plus le langage est simple, plus c'est facile de le maîtriser (p.e. C vs C++, Common Lisp vs Scheme)
- **Puissance expressive**
  - supporte plusieurs styles de programmation (p.e. C vs Scheme)

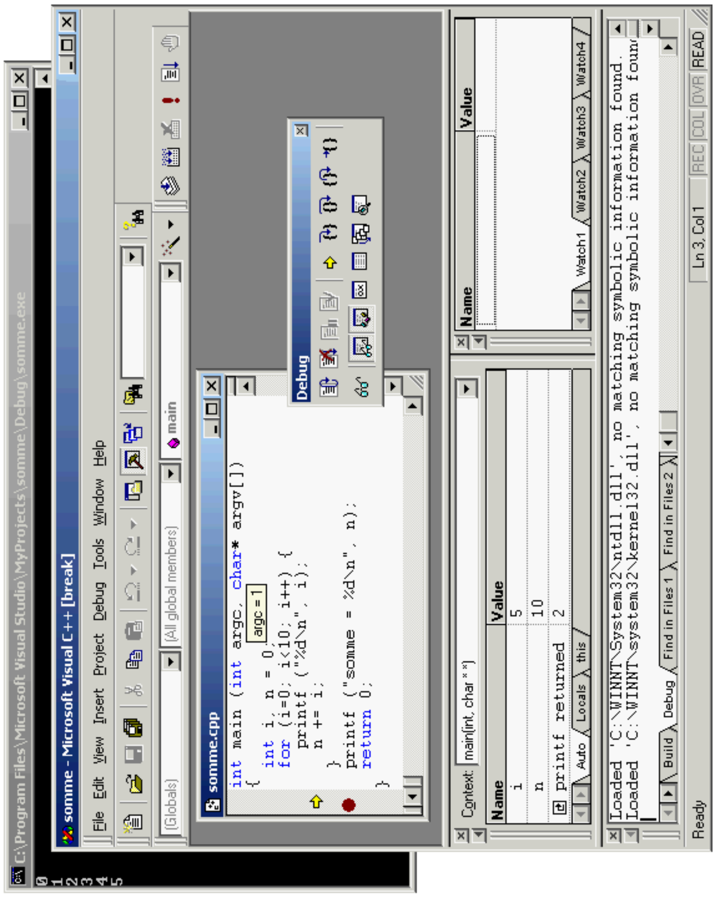
Choix d'un langage de progr. (2)

- **Degré de spécialisation**
  - un langage spécialisé est conçu spécialement pour certaines applications (p.e. SPSS=stat, TCL/Tk=GUI, Postscript=dessin, Icon=texte, Perl=script, COBOL=administration)
  - un langage général (p.e. C, Ada, ML) peut faire n'importe quoi mais avec un peu plus d'effort de programmation
- **Sécurité**
  - le langage aide-t-il à prévenir les erreurs? (p.e. programmation structurée)
  - détection des erreurs? à la compilation? à l'exécution? (p.e. C vs Lisp)
- **Support de gros projets**
  - le langage offre-t-il des mécanismes pour aider la réalisation de gros projets (> 10,000 lignes)?
  - modules? compilation séparée? (p.e. Pascal vs Ada)

Choix d'un langage de progr. (3)

- **Environnement de développement**
  - existe-t-il des outils de développement adaptés au langage?
  - éditeur avec indentation automatique et coloration syntaxique
  - compilateur avec messages précis
  - débogueur symbolique avec exécution pas-à-pas
  - exemple: "Integrated Development Environnement" (IDE) de Microsoft Visual C/C++

Choix d'un langage de progr. (4)



Choix d'un langage de progr. (5)

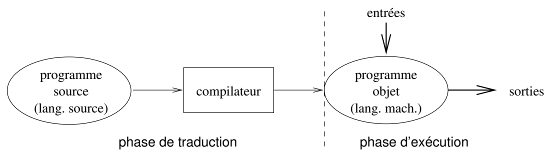
- **Performance**
  - compilateur rapide? exécutable rapide? choix d'optimisations (p.e. C++ vs Java)
- **Coûts de développement et maintenance**
  - les programmeurs disponibles maîtrisent-ils le langage? sont-ils productifs? le langage permet-t-il de réutiliser et étendre des librairies et programmes existants? (p.e. C vs Lisp) le langage incite-t'il une bonne programmation? (p.e. APL et FORTH sont "write-only")
- **Portabilité**
  - le langage est-il répandu? existe-t-il un standard bien établi? (p.e. C vs C++)

Choix d'un langage de progr. (6)

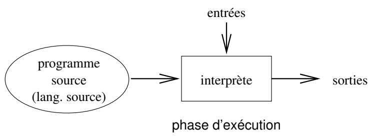
- Note 1: on fait souvent face à des compromis car ces facteurs sont souvent contradictoires (p.e. Java est de haut niveau, de haute puissance expressive, sécuritaire et portable, mais pas très performant)
- Note 2: parfois il est possible d'écrire un programme en utilisant plus qu'un langage, ce qui permet d'utiliser le langage le plus approprié pour chaque partie (p.e. avec la venue des architectures orienté-objet COM et CORBA il est maintenant facile d'écrire des "composantes" réutilisables et d'y faire appel à partir de n'importe quel langage)

Implantation des langages de programmation (1)

- **Compilation**: 2 phases
  1. **traduction** du programme source en un programme objet équivalent en langage machine
  2. **exécution** du programme objet

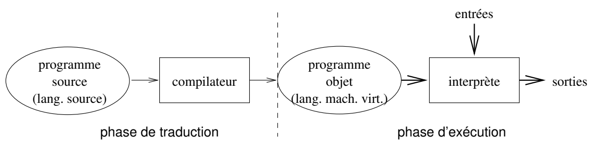


- **Interprétation**: 1 seule phase
  1. **exécution** du programme source, par décodage progressif des instructions



Implantation des langages de programmation (2)

- **Exécution plus rapide** par compilation si programme exécuté plusieurs fois ou contient des boucles (l'interprète répètera le décodage à chaque itération)
- **Plus prompt, compact, flexible et portable** par interprétation (les shells de commande, p.e. `csh`, sont souvent implantés par des interprètes)
- Un interprète implante en quelquesorte une "**machine virtuelle**" (son langage machine = langage source de l'interprète)
- **Approche hybride**: compilation vers machine virtuelle est très portable et moyennement rapide (p.e. JVM)



Syntaxe et sémantique

- Un langage de programmation est défini en 2 parties
  1. **Syntaxe**: apparence textuelle des programmes
  2. **Sémantique**: sens attaché aux programmes (ce qu'ils calculent)
- Exemple 1: en C
  1. `X = 4+1;` est un énoncé valide
  2. `X+4 = 1;` n'est pas un énoncé valide
- Exemple 2: ces énoncés C et Pascal ont le même sens mais pas la même syntaxe
  1. `if (X == 0) Y = 5; /* C */`
  2. `if X = 0 then Y := 5; (* Pascal *)`
- Exemple 3: l'expression `6+1/3` n'a pas le même sens en C (résultat = 6), Pascal (résultat = 6.3333) et Smalltalk (résultat = 2.3333)

Spécification et implantation (1)

- Il est important de distinguer la **spécification** d'un langage de programmation (p.e. description informelle, manuel de programmation, norme) et une **implantation** du langage (p.e. un compilateur ou interprète particulier)
- Un compilateur est **conforme** à une spécification s'il satisfait toutes les exigences de la spécification
- Exemple: spécification **norme ANSI C X3.159-1989**, compilateurs **gcc 2.91.66** et **Microsoft Visual C/C++ 6.0**
- Typiquement une spécification permet plusieurs variantes pour des raisons d'efficacité et de portabilité du compilateur (l'implanteur du compilateur a une plus grande liberté dans ses choix d'implantation)
- Un compilateur peut offrir des **extensions** au langage tout en restant conforme à la spécification

Spécification et implantation (2)

- Exemple 1:
  - en ANSI C, `char` $\subseteq$ `short` $\subseteq$ `int` $\subseteq$ `long`
  - en théorie `char` = `short` = `int` = `long` est possible
  - en général les compilateurs C implantent ceci: `char` = $\{-128..127\}$, `short` = $\{-32768..32767\}$, `int` = $\{-2147483648..2147483647\}$, `long` = `int`
  - exceptions: `char` = $\{0..255\}$ (certains vieux compilateurs C), `short` = `int` = `long` (cc sur Cray T90), `long` = $\{-2^{63}..2^{63}-1\}$ (gcc sur Compaq Alpha)
  - extensions: `long long` = $\{-2^{63}..2^{63}-1\}$ (gcc sur toutes les plateformes)
- Exemple 2:
  - en ANSI C et dans plusieurs langages, l'ordre d'évaluation des sous-expressions n'est pas spécifié
  - `t[f(1)] = g(2)+h(3);` peut faire les appels à `f(1)`, `g(2)`, et `h(3)` dans n'importe quel ordre
  - `i = 0; t[i] = i++;` peut affecter `t[0]` ou `t[1]`

Syntaxes des expressions (1)

- Il existe 3 notations pour les expressions, qui se distinguent par la position de l'opérateur (p.e. +) par rapport aux opérandes (p.e. E1 et E2)
- **Infixe**: E1 + E2
  - opérateur entre les opérandes (notation la plus répandue)
- **Préfixe**: + E1 E2
  - opérateur avant les opérandes (pour Lisp et appel de fonction on ajoute des parenthèses)
  - `+ * x 2 / 1 x` (préfixe) = `x*2+1/x` (infixe)
  - `modulo x y` (préfixe) = `x modulo y` (infixe)
- **Postfixe**: E1 E2 +
  - opérateur après les opérandes (Postscript, calculatrices HP, FORTH et langage machine)
  - `x 2 * 1 x / +` (postfixe) = `x*2+1/x` (infixe)
  - `x y lineto` (postfixe) = `lineto x y` (préfixe)

Syntaxes des expressions (2)

- Pour les opérateurs unaires (i.e. une seule opérande) il faut utiliser une notation préfixe ou postfixe

- Exemple, "`not`" = négation booléene
  - `not < x 0` (préfixe) = `not x<0` (infixe)
  - `x 0 < not` (postfixe) = `not x<0` (infixe)

- Notation infixe est familière mais le sens n'est pas clair dans certains contextes
  - `not x<0` = `(not x)<0` ou `not (x<0)`
  - `a+b*c` = `(a+b)*c` ou `a+(b*c)`
  - `a-b-c` = `(a-b)-c` ou `a-(b-c)`

Syntaxes des expressions (3)

- Solution: employer des règles supplémentaires

  1. **niveau de précéance** d'un opérateur indique dans quel ordre regrouper les sous expressions (plus haut niveau en premier)
     - Typique:

```
^       plus haut niveau
* /
+ -     plus bas niveau
```

  2. **ordre d'associativité** d'un opérateur indique dans quel ordre regrouper les sous expressions lorsque le niveau est le même
     - Typique:

```
^       droite à gauche
* / + - gauche à droite
```

     - `1+2^3^4*5-6` = `1+2^(3^4)*5-6` = `1+(2^(3^4))*5-6` = `1+((2^(3^4))*5)-6` = `(1+((2^(3^4))*5))-6`

  3. **parenthèses** pour forcer un ordre précis
     - Note: inutile en **préfixe** et **postfixe** si le nombre d'opérandes de chaque opérateur est connu
     - `x*(y-2)` = `* x - y 2` (préfixe) = `x y 2 - *` (postfixe)

Syntaxes des expressions (4)

- C a 15 niveaux!

```
15:  x(y)  x[y]              appel de fonction et indexation
14:  x++   ++x   x--   --x   post/pré- incré/décré- mentation de x
     ~x                      négation logique bit-à-bit
     !x                      négation booléene, équivalent à x==0
     &x                      retourne pointeur vers la variable x
     *x                      indirection de pointeur
     -x    +x                inverse additif et identité
13:  x*y   x/y   x%y         multiplication, division, modulo
12:  x+y   x-y               addition, soustraction
11:  x<<y  x>>y              décalage à gauche/droite
10:  x<y   x<=y  x>y   x>=y  1 si la relation est vraie, 0 sinon
 9:  x==y  x!=y              1 si la relation est vraie, 0 sinon
 8:  x&y                     et logique bit-à-bit
 7:  x^y                     ou exclusif logique bit-à-bit
 6:  x|y                     ou logique bit-à-bit
 5:  x&&y                    et booléen, équivalent à x?(y!=0):0
 4:  x||y                    ou booléen, équivalent à x?1:(y!=0)
 3:  x?y:z                   si x!=0 retourne y sinon retourne z
 2:  x=y                     affecte y à x et retourne y
     x+=y  x-=y  x*=y  x/=y  affectations composées: x=x+y, etc
     x%=y  x&=y  x^=y  x|=y
     x>>=y  x<<=y
 1:  x,y                     évalue x puis retourne la valeur de y
```

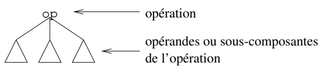
- Les opérateurs de C s'associent de gauche à droite sauf les opérateurs des niveaux 14, 3, et 2, par exemple `*x++` = `*(x++)`, `a?b:c?d:e` = `a?b:(c?d:e)`, et `a=b=c` = `a=(b=c)`

- Il est difficile de se rappeler de tous les niveaux et certains sont contre intuitifs, par exemple l'expression `(x&1 == 0)` est équivalente à `(x & (1==0))` (morale: utiliser des parenthèses sauf pour les opérateurs les plus communs)

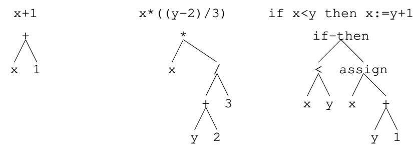
Arbres de syntaxe abstraits (ASA)

- Déf: représentation d'un (fragment de) programme sous forme d'arbre qui souligne sa structure et élimine les détails syntaxiques

- Forme générale:



- Exemples:



- L'ASA ne contient pas les parenthèses, point-virgules, etc car l'ASA exprime directement les regroupements qu'elles produisent

- Si deux fragments dans 2 langages différents ont le même ASA c'est qu'ils sont "équivalents"
  - Par exemple en C: `if (x<y) x=y+1;`

Définition formelle de la syntaxe (1)

- Basé sur des concepts d'informatique théorique
- Méthode classique = définir une **grammaire** pour le langage
- La grammaire spécifie exactement les programmes (ou "phrases") qui sont acceptables syntaxiquement
- Analogie avec langue naturelle:

```
ce chat est noir         OK
est noir chat ce         pas OK

x := 0 ;                 OK
0 ; := x                 pas OK
```

- Note: un programme est composé d'une séquence de symboles (identificateurs, mot réservés, ponctuation, etc)

Définition formelle de la syntaxe (2)

- Déf: **Vocabulaire** (Σ) = ensemble de symboles qu'on peut retrouver dans une phrase
  Ex: Σ={0,1} Σ={+,-,a,...,z} Σ={if,and,:=,<,...}
- Déf: **Phrase** = séquence possiblement vide de symboles tirés de Σ
  Ex: Σ={0,1} phrase1=00101 phrase2= (vide)
- Déf: **Langage** = ensemble de phrases, possiblement infini
  Ex1: Σ={0,1} L1={00,01,10,11}
  Ex2: Σ={0,1} L2={1,10,100,1000,...}

Définition formelle de la syntaxe (3)

- Déf: **Grammaire hors-contexte** en format BNF (Backus-Naur Form) = ensemble de **catégories** et **productions**
  - **Catégorie** = nom qui désigne un type de fragment de phrase, par convention entouré de < ... >
    Ex: <*expression*> <*entier*> <*vide*>
  - **Production** = règle de la forme
    <*cat*> ::= x1 x2 ... xn
    où <*cat*> est une catégorie et x*i* est une catégorie ou symbole pour tout *i*
    Ex: <*X*> ::= 0 <*Y*> 1
- Par convention la première production indique la **catégorie de départ**
- Exemple: grammaire G1 =
  <*bin*> ::= 0
  <*bin*> ::= 1
  <*bin*> ::= <*bin*> <*bin*>

Définition formelle de la syntaxe (4)

- Déf: **Dérivation directe** (notation X=>Y)
  Soit x1 ... xn une chaîne et
  - x*i* est une catégorie ou symbole pour tout *i*
  - x*j* = <*C*>
  - il existe une production <*C*> ::= y1 ... ym

  alors
  x1...x*j-1* <*C*> x*j+1*...xn => x1...x*j-1* y1...ym x*j+1*...xn
- Déf: **Dérivation** (d'une phrase) = une séquence de dérivations directes à partir de la catégorie de départ dont le résultat est la phrase en question
  Ex: <*bin*> => <*bin*> <*bin*> => <*bin*> 0 => 1 0
- Déf: **L(G)** (le langage défini par la grammaire G) = ensemble de toutes les phrases dérivables par G
- L(G) = { p | <*départ*> => ... => p }

Définition formelle de la syntaxe (5)

- Exemple: grammaire G2 =
  <flottant> ::= *<entier>* . *<entier>*
  <entier> ::= *<chiffre>*
  <entier> ::= *<chiffre>* *<entier>*
  <chiffre> ::= 0
  <chiffre> ::= 1
  <chiffre> ::= 2
  <chiffre> ::= 3
  <chiffre> ::= 4
  <chiffre> ::= 5
  <chiffre> ::= 6
  <chiffre> ::= 7
  <chiffre> ::= 8
  <chiffre> ::= 9

- `1.5` dans L(G2)? oui:
  <flottant> => *<entier>* . *<entier>*
  => *<chiffre>* . *<entier>* => 1 . *<entier>*
  => 1 . *<chiffre>* => 1 . 5

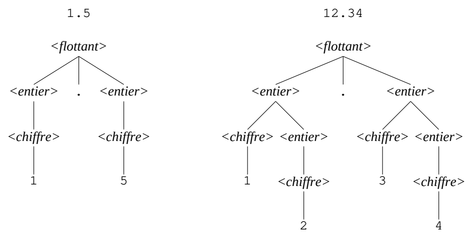
- `.5` dans L(G2)? non

Arbre de dérivation

- Utile pour représenter la structure syntaxique d'une phrase

- Étant donné une certaine dérivation d'une phrase P, l'arbre de dérivation correspondant aura la catégorie de départ à sa racine, des symboles aux feuilles, et chaque noeud interne est une catégorie qui a comme enfants la partie droite de la production qui l'a remplacée dans la dérivation

- Exemples avec grammaire G2



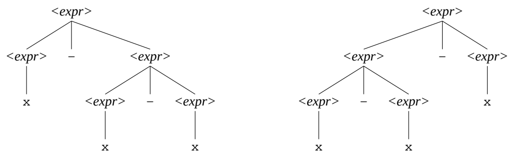
- Les feuilles de l'arbre de dérivation = la phrase

Grammaires ambiguës (1)

- Déf: Une grammaire G est **ambiguë** ssi il existe une phrase P dans L(G) tel que P a plus qu'un arbre de dérivation

- Exemple de grammaire ambiguë: G3 =
  <expr> ::= x
  <expr> ::= *<expr>* - *<expr>*

- 2 arbres de dérivation pour: `x - x - x`



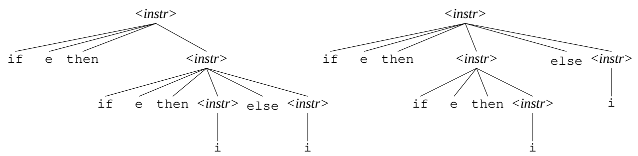
- Les grammaires ambiguës sont à éviter car normalement le compilateur utilise l'arbre de dérivation pour attacher un sens au programme

- Si la grammaire est ambiguë, le compilateur doit utiliser d'autres règles pour retirer les ambiguïtés

Grammaires ambiguës (2)

- Cas classique, le "else" pendant: gram. G4 =
  <instr> ::= i
  <instr> ::= **if e then** *<instr>*
  <instr> ::= **if e then** *<instr>* **else** *<instr>*

  if e then if e then i else i



- Solution typique (Algol60/Pascal/C): règle "**else** va avec le **if** le plus proche"

- Algol68/Modula-2/Ada/Fortran77 évitent ce problème en demandant un **fi**/**END**/**end if**/**endif** à la fin d'un if: gram. G5 =
  <instr> ::= i
  <instr> ::= **if e then** *<instr>* **fi**
  <instr> ::= **if e then** *<instr>* **else** *<instr>* **fi**

  if e then if e then i else i fi fi
  if e then if e then i fi else i fi

Autres formalismes syntaxiques (1)

- Déf: **BNF étendu (EBNF)** = BNF + expressions régulières

- Notation permise dans les productions:
  - **[** X **]** −> X est optionnel
  - { X } −> répétition de X (0 ou + fois)
  - X | Y −> choix entre X et Y
  - **(** X Y Z **)** −> groupement

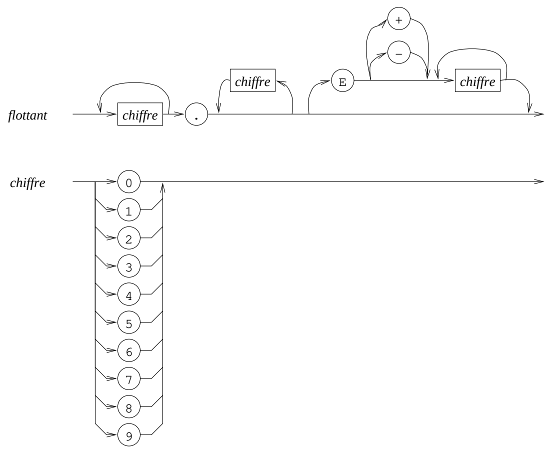
- Exemple: gram. G6 =
  <flottant> ::= *<entier>* . { *<chiffre>* } [*<exp>*]
  <entier> ::= *<chiffre>* { *<chiffre>* }
  <chiffre> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  <exp> ::= E [+ | -] *<entier>*

Autres formalismes syntaxiques (2)

- Déf: **Diagramme syntaxique** = représentation graphique d'une grammaire avec 1 graphe orienté par catégorie (les noeuds des graphes sont soit des catégories ou symboles du vocabulaire)

- Langage = ensemble de tous les chemins possible dans le graphe à partir de la catégorie de départ

- Exemple: équivalent de la grammaire G6



Programmation impérative

- Style de programmation offert par la plupart des langages de programmation (machine, assembleur, FORTRAN, Algol, SIMULA, C/C++, Pascal, Ada, Java, Lisp, Scheme, ML, ...)

- Basé sur le modèle d'une **machine**; le programme indique la **séquence d'opérations** que la machine doit effectuer pour faire le calcul

- Déf: un **point de contrôle** c'est un emplacement du programme entre deux opérations

```
begin  ! programme SIMULA ;
   integer x, y;
1
   x := inint;
2
   y := inint;
3
   x := x+y*y;
4
   outint (x, 0);
5
end;
```

- Note: dépendant du point de vue certains énoncés du programme peuvent donner lieu à plus d'un point de contrôle si ils font plus qu'une opération (p.e. `x:=inint;`)

Exécution d'un programme

- Déf: l'**état** de l'exécution d'un programme (si on suppose une seule entrée et sortie et pas d'appel de procédure) =

  1. **point de contrôle** où se trouve l'exécution
  2. **contenu** de toutes les variables et structures de données
  3. partie de l'**entrée** qui reste à lire et **sortie** produite jusqu'à date

```
begin  ! programme SIMULA ;
   integer x, y;
1
   x := inint;
2
   y := inint;
3
   x := x+y*y;
4
   outint (x, 0);
5
end;
```

EXÉCUTION

pt.	variables	entrée	sortie
1	x=? y=?	1 2	
2	x=1 y=?	2	
3	x=1 y=2		
4	x=5 y=2		
5	x=5 y=2		5

- L'état change après l'exécution de chaque opération

- L'**ordre des opérations** est important

```
y := inint; ! ce programme n'est pas équivalent ;
x := inint;
x := x+y*y;
outint (x, 0);
```