COBOL

HELMo

Saint - Laurent

Langage COBOL

Une introduction à

Langages et Compilation

Louis SWINNEN
louis.swinnen@gmail.com

Mars 2008

Introduction (1)

- Les langages
  - Cette partie présente des notions importantes sur les langages de programmation
    - On détaillera les différentes *familles* de langages
    - On classera les langages suivant leurs *paradigmes*
    - Des exemples seront donnés dans ces différentes langages

Introduction (2)

- La compilation
  - Nous présenterons également les principes de base de la compilation
    - Les différentes tâches réalisées par un compilateur
      - Les analyses lexicales, syntaxiques et sémantiques
      - La génération de code
    - Les différences entre la compilation et l'interprétation
  - Un exemple très simple de compilation sera également présenté

Introduction (3)

- Limites
  - Ces deux thèmes que sont les langages et la compilation pourraient faire l'objet de deux cours séparé et conséquent
  - Cette exposé se limitera donc à vous *faire sentir* les différentes étapes importantes
  - J'ai volontairement laissé de coté l'aspect formel sous-jacent aux théories de la compilation. Pour plus d'explication, reportez-vous aux livres de références !

HELMo
Saint - Laurent

Références

- *D. Grune, H. E. Bal, C. J. H. Jacobs, K. G. Langendoen*, Modern Compiler *Design*, John Wiley & Sons, England, 2000
  - Paru en français sous le titre Compilateur aux éditions Dunod, Paris, 2002
- *A. Aho, R. Sethi, M. S. Lam, J. Ullman,* Compilers : principles, techniques and tools, 2nd ed., Addison Wesley, 2006
  - Paru en français sous le titre: Compilateurs : principes, techniques et outils, 2e ed. aux éditions Pearson Education, 2007

HELMo Saint - Laurent

Exercice Introductif

HELMo
Saint - Laurent

Exercice introductif (1)

- Nous allons réaliser un *interpréteur* capable d'analyser une expression mathématique simple et parenthesée.
  - Exemple:
    - (3+(4*5))*2
    - Le système doit donner 46 comme réponse
  - Simplification:
    - Seul des opérateurs binaires seront utilisés
    - Uniquement sur des entiers d'1 chiffre
    - Les parenthèses groupent un seul opérateur
      - pas de (3+4+5) mais ((3+4)+5)
    - Les parenthèses au début et à la fin sont présentes. -> pas (3+4)+5 mais ((3+4)+5)

HELMo Saint - Laurent

Exercice introductif (2)

- Intérêt
  - Exercice relativement simple
    - Peut être fait à la main ! Ce qui n'est pas le cas des compilateurs de langage pour lesquels il faut souvent utiliser des outils spécifiques.
  - Permet de comprendre certaines étapes d'un compilateur / interpréteur
  - Cet exemple sera repris par la suite pour être analysé et commenté
    - Au fur et à mesure que les différents points de théorie seront abordés

HELMo Saint - Laurent

Exercice introductif (3)

- Ecrivez, en C, le programme permettant de réaliser cette analyse.
  - Réfléchissez bien au problème.
    - L'utilisation de piles (opérateur et chiffres) simplifient grandement la résolution !
  - Forme d'une expression entre parenthèse:
    - (val1 operateur val2)
    - val1 et val2 peuvent être d'autres expressions parenthésées
      - ((3+5) * (6+4))
      - ((3+(4*5))*2)

HELMo Saint - Laurent

Les Langages

HELMo
Saint - Laurent

Introduction (1)

- Les paradigmes de programmation
  - *« est un style fondamental de programmation informatique qui traite de **la manière** dont les solutions aux problèmes doivent être formulées dans un langage de programmation »*. (Wikipedia)
  - On distingue différents paradigmes:
    - Programmation impérative
    - Programmation orientée-objet
    - Programmation fonctionnelle
    - Programmation déclarative
    - Programmation logique
    - ...

HELMo Saint - Laurent

Introduction (2)

- Les langages de programmations sont répertoriés en fonction de ces paradigmes

C, Cobol, Fortran, Algol, Assembleur, Basic, Pascal, Perl, PHP, ...

Lisp, OCaml, Scheme

C++, Java, C#, Delphi, SmallTalk, Visual Basic

SQL, ...

HELMo Saint - Laurent

Introduction (3)

- Particularités
  - Tous les langages d'une même famille peuvent résoudre les mêmes types de problème
    - Les langages Cobol, C, BASIC, ... permettent donc de résoudre *les mêmes familles de problèmes*.
  - Ces langages sont alors *équivalents*
    - Pas vraiment de langage « meilleur » par rapport à un autre.
    - Le choix dépendra d'abord du programmeur
    - Certaines particularités propres à un langage peuvent également déterminer le choix.
      - Gestion des fichiers en Cobol
      - Précision scientifique de Fortran
      - Récursivité, pointeur du C

HELMo Saint - Laurent

Langages (1)

- Tour d'horizon des familles de langage
  - Langages impératifs (très courants)
    - Le programmeur développe, étape par étape, l'algorithme que l'ordinateur doit suivre.
      - Ex: Cobol, C, PHP, Fortran, ...

```
printf("Nom ? ");
scanf("%[^\n]%*c", nom);

while(strcmp(nom, "X") != 0)
{
    .
    .
    .
    printf("Nom ? ");
    scanf("%[^\n]%*c", nom);
}
```

Exemple en C

```
DISPLAY "Nom ? " WITH NO
ADVANCING
ACCEPT nom
PERFORM UNTIL nom(1:1) NOT= "X"
    .
    .
    .
    DISPLAY "Nom ? " WITH NO
      ADVANCING
    ACCEPT nom
END-PERFORM
```

Exemple en Cobol

HELMo Saint - Laurent

Langages (2)

- Les langages orientés-objets
  - Cette famille de langage est très proche de la précédente. La raison pour laquelle je l'ai séparée de la classe précédente est qu'on lui adjoint très souvent *la programmation événementielle*
    - La partie *interface vers l'utilisateur* est un aspect important
    - L'application est pilotée par les événements que l'utilisateur réalise.
    - Le fil de l'application n'est plus une structure impérative monolithique classique, mais un ensemble de petites structures impératives déclenchées au grés des événements qui surviennent.
    - Le propre de la programmation orientée-objet est de pouvoir représenter des « objets » du monde réel en objet informatique (un client, une facture, ...)

Langages (3)

- Exemple en Java


```
public class MainForm extends JFrame {
    private JPanel mainPanel;
    private JButton quitterButton;

    public MainForm() {
        add(mainPanel);
        pack();
        quitterButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    dispose();
                    System.exit(0);
                }
        });
        setVisible(true);
    }

    public static void main(String[] args) {
        new MainForm();
    }
}
```

HELMo Saint - Laurent

Langages (4)

- La programmation fonctionnelle
  - En programmation fonctionnelle, il n'y a que des fonctions que l'on peut appeler.
  - Cette programmation, relativement épurée propose des structures assez simple:
    - Pas d'état (pas de variables en dehors des paramètres des fonctions)
    - Pas de structure répétitive (la seule méthode possible est la récursivité)
    - Une sélection simple au moyen de if-then-else
    - Les types particuliers comme les listes, et certaines fonctions de gestion de ces types
  - Cette manière de programmer est proche du mode de pensée utilisé en mathématique
  - Ces langages sont généralement fortement typés

HELMo Saint - Laurent

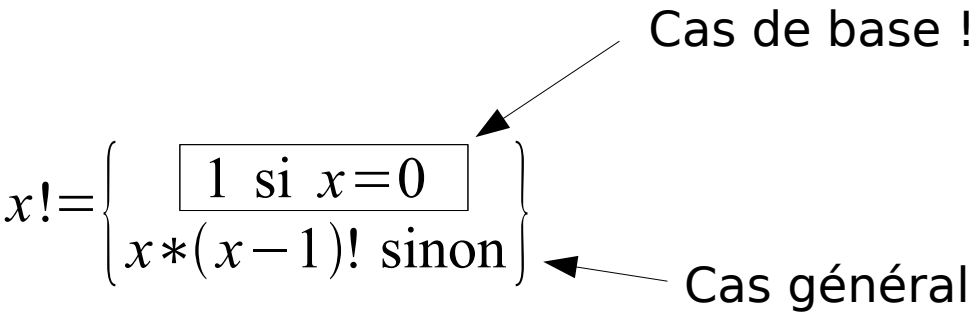
Langages (5)

- Le Pattern Matching
  - Il s'agit d'une correspondance effectuée par le langage. Le pattern Matching est une opération importante car elle détermine le comportement de la fonction.
  - Au départ d'une expression, le système va vérifier s'il elle correspond à une forme déterminée (le pattern). Si c'est le cas, l'action correspondante est réalisée.
  - Si l'expression de correspond pas au pattern proposé, le pattern suivant est alors testé.
  - Exemple: *scanf* est une fonction qui vérifie si une expression correspond au format proposé. C'est une espèce de *pattern matching*.
  - Forme du pattern matching:
    - pattern 1 -> action 1
| pattern 2 -> action 2
| pattern 3 -> action 3
  - La correspondance doit être complète. C'est-à-dire que tous les cas doivent être couverts !

Langages (6)

- Présentation de OCaml
  - Développé par l'INRIA (france)
  - Disponible sur toutes les plateformes
  - Exemples:

1. La factorielle




```
# let rec fact = function 0 -> 1
                        | x -> x*(fact (x-1)) ;;
val fact : int -> int = <fun>

# fact 4;;
- : int = 24
```

HELMo
Saint - Laurent

Langages (7)

- Les types suivants sont reconnus:
  - Les types de base : entier, flottant, booléen, ...
  - Le type string
  - Les listes
- Cas des listes:
  - Une liste est une suite d'élément
  - [1;2;3;4] est une liste d'entier
  - a::b décrit *un patern* pour une liste ou *a* est le premier élément et *b* une sous-liste composée des autres éléments

HELMo Saint - Laurent

Langages (8)

- Autres exemples:

```
exception Error;;
# let first = function [] -> raise Error
                     | a::b -> a;;
val first : 'a list -> 'a = <fun>

# first [1;2;3;4];;
- : int = 1


# let rec last = function [] -> raise Error
                        | a::[] -> a
                        | a::b -> last b;;
val last : 'a list -> 'a = <fun>

# last [1;2;3;4];;
- : int = 4
```

HELMo Saint - Laurent

Langages (9)

- Autres exemples (suite)

Fonction qui calcule le carré d'un nombre :

```
# let carre = function x -> x*x;;
val carre: int -> int = <fun>

# carre 2;;
- : int = 4
```

Fonction qui calcule la TVA d'un montant :

```
# let tva = function x -> x *. 0.21;;
val tva : float -> float = <fun>

# tva 15.0;;
-: float = 3.15
```

HELMo Saint - Laurent

Langages (10)

- Autres exemples

Concaténer deux listes quelconcques

```
# let rec concat l1 l2 = match (l1, l2) with
      ([],l2) -> l2
    | (h::q,l2) -> h::(concat q l2);;
val concat : 'a list -> 'a list -> 'a list = <fun>

# concat [1;2;3] [4;5;6];;
-: int list = [1; 2; 3; 4; 5; 6]
# concat ['a';'b';'c'] ['d';'e';'f'];;
-: char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']
```

Appliquer une fonction quelconque à une liste quelconque

```
# let rec update f l1 = match (f, l1) with
      (f, []) -> []
    | (f, a::b) -> (f a)::update f b;;
val update : ('a -> 'b) -> 'a list -> 'b list = <fun>

# update tva [15.0;30.0;35.0];;
-: float list = [3.15; 6.3; 7.35]
```

HELMo Saint - Laurent

Langages (11)

- Exercices

Fonction qui calcule le carré d'une liste d'entier (2 formes) ?

Fonction qui intercale deux listes ?

HELMo
Saint - Laurent

Langages (9)

- La programmation déclarative
  - Le programmeur décrit le résultat mais jamais comment le système doit lui fournir.
  - Ex: SQL
    - SELECT * FROM CLIENTS WHERE NOM LIKE 'Louis%';
  - Le moteur SQL, en analysant la requête, va déterminer le meilleur moyen de satisfaire la demande.
    - Il va déterminer comment accéder aux données
      - Quelle table d'index utiliser ou s'il est préférable de faire un parcours séquentiel des données
    - Localiser et lire les données voulues
    - Transmettre le résultat
  - Très différent d'un système procédural !

HELMo Saint - Laurent

Langage (10)

- La programmation logique
  - La programmation logique se base sur un ensemble de faits et de règles.
    - Sur base des faits et des règles, un moteur d'inférence permet de résoudre des questions
  - Ce type de programmation est utilisée en Intelligence Artificielle, démonstration de programmes, ...
    - Ex en Prolog (Wikipedia):
      - chat(tom). fait.
      - ?- chat(tom). requête
        oui
      - ?- chat(X). requête
        X=tom ;
        fail

HELMo
Saint - Laurent

Langage (11)

- Autre exemple (wikipedia):

```
frere_ou_soeur(X,Y) :- parent(Z,X), parent(Z,Y), X \= Y.
pere(X, Y) :- parent(X,Y), male(X).
mere(X,Y) :- parent(X,Y), femelle(X).
parent(X,Y) :- pere(X,Y).
parent(X,Y) :- mere(X,Y).                                   règles
```

```
mere(trude, sally).
pere(tom, sally).
pere(tom, erica).
pere(mike, tom).
male(tom).
femelle(trude).
male(mike).                                                 faits
```

```
?- frere_ou_soeur(sally, erica)
      oui.
```

HELMo Saint - Laurent

La Compilation

HELMo
Saint - Laurent

Intérêt (1)

- Pourquoi détailler les compilateurs ?
  - Application très courante
    - Les notions sous-jacentes aux compilateurs sont présentes:
      - Dans le navigateur Web pour afficher une page HTML (analyse d'un texte source et production d'un résultat)
      - Dans les parseurs en tout genre (XML, ...)
      - Dans les documents, les imprimantes, ... pour le format Postscript par exemple
      - Dans les programme de conversion ou de traduction d'un langage vers un autre
      - Le développement de langages particuliers adaptés à des sous-classes de problèmes

Intérêt (2)

- Algorithmiquement intéressant
  - Analyse de texte
    - Nécessite de bien comprendre les fonctions d'analyse de texte, de chaînes, ...
  - Structures de données souvent complexes
    - Utilisation d'arbre pour représenter la structure
    - Utilisation de pile, de liste chaînées, ...
- Intéressant pour les concepts théoriques
  - Syntaxe
    - Syntaxe concrète
    - Syntaxe ou arbre abstrait
  - Génération de code
  - Exécution

HELMo Saint - Laurent

Introduction (1)

- Un compilateur est un programme
  - qui **analyse** un programme
  - pour le **comprendre** (d'une certaine façon)
  - et le **traduire** dans un autre langage
- Le compilateur C
  - analyse un programme source écrit en langage C
  - *comprend* ce programme (le programme est-il valide ?)
  - traduit ce langage en code objet et lie ce code objet avec des librairies -> exécutable

HELMo Saint - Laurent

Introduction (2)

- Le compilateur Java
  - analyse un programme source *java*
  - *comprend* ce programme
  - *traduit* ce programme en bytecode lisible par la machine virtuelle Java
- L'interpréteur Java
  - Analyse le bytecode java
  - comprend ce programme
  - interprète chaque instruction de haut niveau au fur et a mesure

HELMo Saint - Laurent

Introduction (3)

- Schéma d'un compilateur

Analyse lexicale
Analyse syntaxique

Analyse sémantique

Génération de code

- Schéma d'un interpréteur

Analyse lexicale
Analyse syntaxique

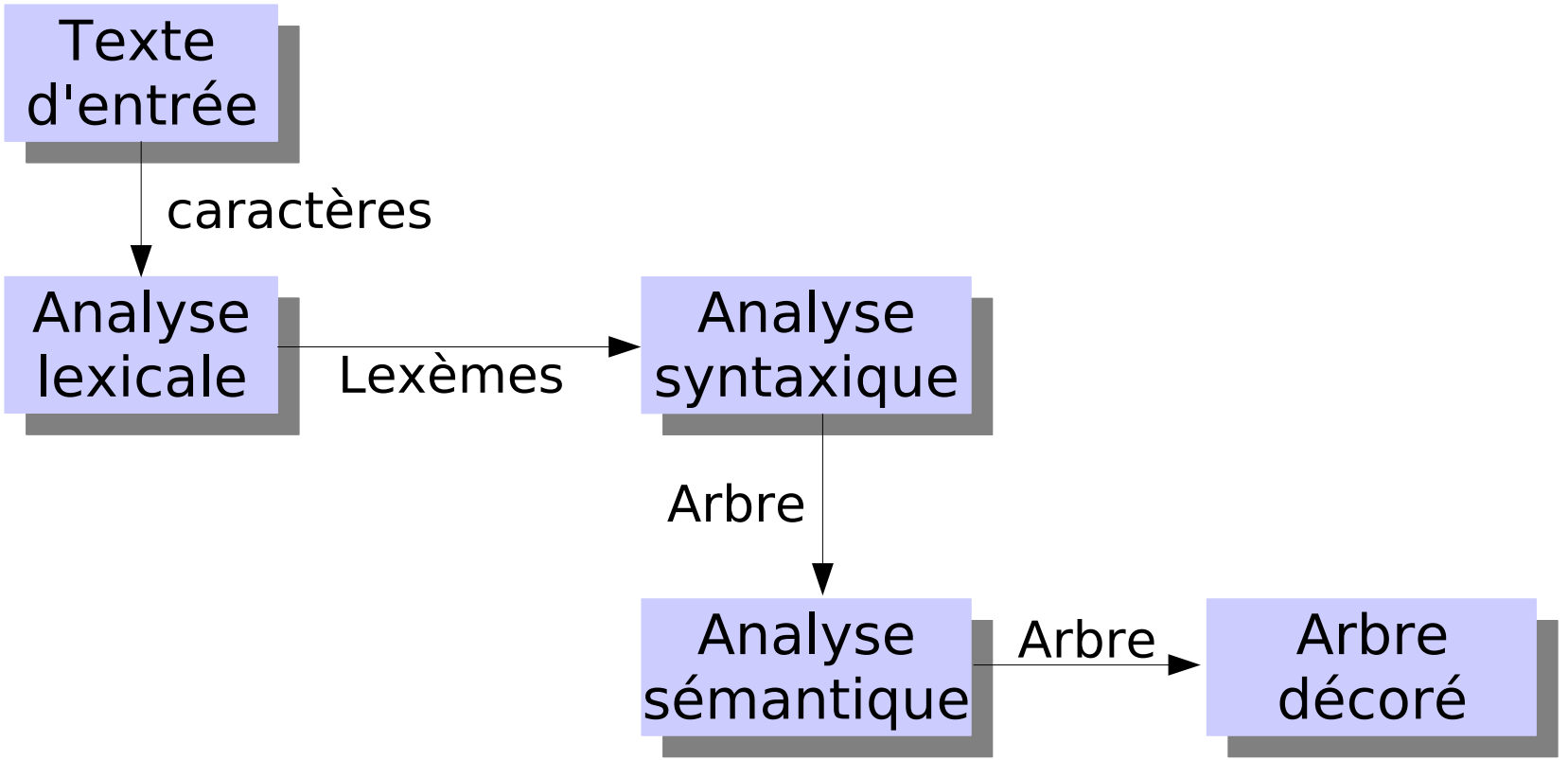
Analyse sémantique

Exécution de code

HELMo Saint - Laurent

Introduction (4)

- Descriptif des analyses



HELMo Saint - Laurent

Figure: copyright (c) John Wiley & Sons, 2000

Analyse lexicale (1)

- But
  - Au départ d'un flux de caractères, l'analyseur lexical **reconnaît les mots réservés du langage**, appelé également Lexème.
    - Il s'agit donc d'un *parser* qui identifie les mots.
- Exemple
  - Dans notre exemple initial, *la parenthèse ouvrante et fermante, les opérateurs, les chiffres* sont les lexèmes de notre langage.
    - (,), +, -, *, /
    - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

HELMo Saint - Laurent

Analyse lexicale (2)

- Spécification
  - Les lexèmes sont souvent décrit par une expression régulière.
    - Exemples:
      - chiffre -> [0-9]
      - operateur -> + | - | * | /
  - Ils sont souvent séparés les uns des autres par des espaces
    - Exemple:
      - 3 + 5 (Nous avons ici 3 lexèmes)
  - Operateurs
    - La répétition: + et * ; Le groupement : ( )
    - L'alternative: |

HELMo Saint - Laurent

Analyse lexicale (3)

- Les grammaires
  - Afin de spécifier complètement les langages, les grammaires *non-contextuelles* sont utilisées.
  - Ainsi, les constructions correctes d'un langage prennent la forme de BNF (*Backus-Naur Form*).
  - Exemples:
    - lettre -> [a-z][A-Z]
    - chiffre -> [0-9]
    - lettre_chiffre -> lettre I chiffre
    - souligne -> _
    - fin_soulignee -> souligne lettre_chiffre+
    - identificateur -> lettre lettre_chiffre* fin_soulignee*
  - Correct ?
    - 2monid monid_ mon_id a_1 bonjour_toi_123 voici_un_chouette_123_identificateur_pour_moi

HELMo Saint - Laurent

Analyse lexicale (4)

- La grammaire reconnaît **la syntaxe dite « concrète » du langage**.
  - Il s'agit exactement de ce que le programmeur doit taper comme texte.
  - Doit faire l'objet d'une description stricte afin que l'utilisateur connaisse la forme à employer
  - En effet, la grammaire permet de vérifier si le programme est rédigé correctement.
    - Vérification des séparateurs (" **;** " en C)
    - Validation du code
  - Il faut lever toute ambiguïté:

```
if(condition)
    if(condition)
        action
else
```

A quel IF se rapporte-t-il ?
En C, au IF le plus proche !

HELMo Saint - Laurent

Analyse lexicale (5)

- Architecture d'un analyseur lexical
  - L'analyseur lexical doit retourner l'information qu'il a identifié
    - LETTRE, CHIFFRE, IDENTIFICATEUR, ...
    - OPERATEUR, TERME, FACTEUR, ...
  - Comment faire ?
    - Les différents mots du langage sont rassemblés dans des clauses DEFINES
      - #define OPERATEUR 1
      - #define CHIFFRE 2
    - Et les fonctions d'analyse retournent *les classes d'information ou lexèmes identifiés.*

HELMo Saint - Laurent

Analyse lexicale (6)

- Exemple (analyse de messages réseaux)

```
/* defined tokens */
#define CONNECT 1
#define LOGON 2
#define LIST 3
#define TALK 4
#define NEWFRIEND 5
#define ACCEPT 6
#define REJECT 7
#define LOGOFF 8
#define SIGNIN 9

/* messages */
#define CONNECT_STR "CONNECT"
#define LOGON_STR "LOGON"
#define LIST_STR "LIST"
#define TALK_STR "TALK"
#define NEWFRIEND_STR "NEWFRIEND"
#define ACCEPT_STR "ACCEPT"
#define REJECT_STR "REJECT"
#define LOGOFF_STR "LOGOFF"
#define SIGNIN_STR "SIGNIN"
```

Les mots du protocole sont définis par des define

HELMo Saint - Laurent

Analyse lexicale (7)

- Exemple (suite)

```
#define CMP(i,j) ( (strlen(i) >= strlen(j)) &&
                   (strncasecmp(i, j, strlen(j)) == 0) )

int parse(char* message, int length)
{
        if(length < 1)
                return PARSE_ERROR;

        if(CMP(message, CONNECT_STR)) {
                return CONNECT;
        } else if(CMP(message, LOGON_STR)) {
                return LOGON;
        } else if(CMP(message, SIGNIN_STR)) {
                return SIGNIN;
        } else if(CMP(message, LIST_STR)) {
                return LIST;
        } else if(CMP(message, TALK_STR)) {
                return TALK;
        .
        .
        .
}
```

HELMo Saint - Laurent

Analyse lexicale (8)

- Génération automatique
  - Plus personne (ou presque) ne construit un analyseur à la main
    - sauf lorsque la grammaire est petite et simple
  - Il existe des programmes qui génèrent l'analyseur lexical
    - Pas d'erreur dans la manipulation des chaînes
    - Génération de code automatique (parfois pas très lisible)
    - Il faut être rigoureux dans la spécification que l'on fait !
    - Outils utilisés: lex, flex, ...
      - code généré en C

HELMo Saint - Laurent

Analyse lexicale (9)

- Exemple de spécification (pour un langage):

Format compatible LEX/FLEX

Définition des règles de base :

```
lettre [a-z]|[A-Z]
chiffre_non_nul [1-9]
chiffre 0|{chiffre_non_nul}
identificateur {lettre}({lettre}|{chiffre})*
mantisse 0|{chiffre_non_nul}({chiffre})*
valeur_exposant {mantisse}
exposant E{valeur_exposant}|e{valeur_exposant}
signe \+|\-
constante_numerique ({signe}{0,1}) {mantisse}
                          ({exposant}{0,1})
constante_booleenne true|false|TRUE|FALSE
enter \n
```

Source:

HELMo Saint - Laurent

Analyse lexicale (10)

- Exemple de spécification (suite):

```
"program" { identificateur } { return PROGRAM; }
"if" { return IF; }
"(" { return LPAR; }
")" { return RPAR; }
"+" { return PLUS; }
"-" { return MOINS; }
"*" { return FOIS; }
"/" { return DIV; }
"and" { return AND; }
"or" { return OR; }
"not" { return NOT; }
"<-" { return AFF; }
```

On définit ainsi tous les mots clés du langage !

Source:

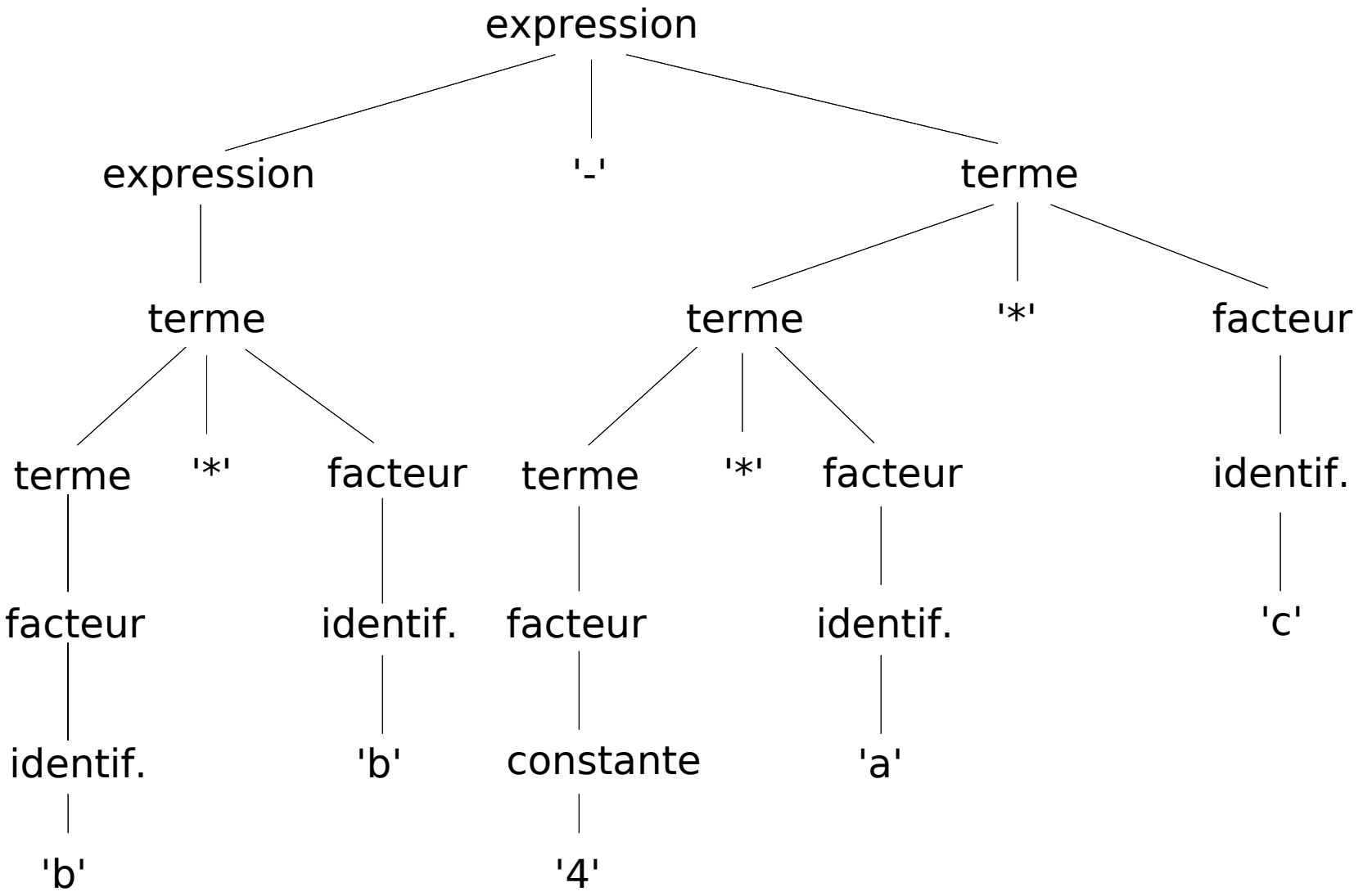
HELMo Saint - Laurent

Analyse syntaxique (1)

- L'analyse syntaxique
  - Grâce à l'analyse lexicale, le compilateur reconnaît des mots
  - L'analyse syntaxique permet, sur base de la grammaire, de reconnaître des phrases
    - Exemple:
      - Si j'ai une parenthèse ouvrante, suivie d'une expression, suivie d'une parenthèse fermante, cela donne une expression parenthésée.
  - En construisant *un arbre*, il est possible de vérifier et reconnaître la grammaire.

HELMo Saint - Laurent

Analyse syntaxique (2)



HELMo Saint - Laurent

Figure: copyright (c) John Wiley & Sons, 2000

Analyse syntaxique (3)

- Analyse descendante et ascendante
  - L'**analyse descendante** tente de déterminer le plus rapidement possible la règle de grammaire en rapport avec ce qui est observé.
    - Construction de l'arbre syntaxique du haut vers le bas
      - expression -> expression -> terme -> terme -> facteur -> identificateur
  - L'**analyse ascendante** va se concentrer sur les éléments observés et ensuite déterminer la règle de grammaire applicable
    - Construction de l'arbre syntaxique du bas vers le haut
  - Ces deux approches sont assez différentes, on utilise principalement l'analyse ascendante dans les compilateurs

HELMo Saint - Laurent

Analyse syntaxique (4)

- Analyse LR(1) et LALR(1)
  - Sans entrer dans des grandes théories, ces méthodes d'analyse ascendantes sont très souvent utilisées:
    - Analyse des lexèmes
    - Avec une **prévision de 1** qui permet de savoir ce qui suit (sans l'avoir déjà traité)
    - Cette prévision permet de déterminer correctement l'élément de la grammaire

```
phrase_simple := mot point
phrase:= mot (SP mot)+ point
point := '.'
mot := (lettre)+
SP := ' '
```

    - Pour différencier une *phrase* d'une *phrase_simple*, l'élément de prévision est important. Suivant ce qui est observé à la source, le choix est alors possible

HELMo Saint - Laurent

Analyse syntaxique (5)

- Construction de l'analyseur
  - A la main
    - Les analyseurs descendant simples peuvent être créés par le programmeur.
    - Difficile si grammaire complexe
  - Générateur automatique
    - Comme pour l'analyse lexicale, il existe des outils de génération d'analyseur syntaxique. Les plus connus sont certainement :
      - Yacc (Yet Another Compiler Compiler) qui génère du C
      - Bison qui génère du C ANSI (conforme !)
    - Ces générateurs s'occupent de la reconnaissance des chaînes, de la sélection de la règle, ... moins d'erreur !
    - Ces générateurs se basent sur des techniques d'analyse LALR(1): analyse ascendante avec 1 élément de prévision.

Analyse syntaxique (6)

- Exemple

```
exp: terme {}
     | terme op_add exp {}

terme: facteur {}
       | facteur op_mult terme {}

facteur: arg {}
         | NOT arg {}
         | op_add arg {}

arg:   NOMBRE {}
       | l_exp {}
       | appel_de_fonction {}
       | LPAR r_exp RPAR {}

op_add: PLUS {}
        | MOINS {}
```

Source:

HELMo Saint - Laurent

Analyse sémantique (1)

- Après avoir
  - Lu le texte source et identifié les lexèmes sur base des mots réservés du langage
  - Assemblé ces lexèmes et reconnus des « phrases », sur base des règles de grammaire
- Il faut
  - Donner **une signification** au texte lu. Car jusqu'à présent, on s'est contenter de reconnaître et valider le texte source.
  - Il faut maintenant **construire l'arbre abstrait** et maintenir les états importants.

HELMo Saint - Laurent

Analyse sémantique (2)

- En effet, il faut **associer des actions** aux éléments identifiés d'un programme.
  - Que faire si l'on rencontre une déclaration de variable ?
    - Il faut mémoriser l'identificateur dans la table des symboles
    - Calculer la valeur qui est affectée et la stocker.
  - Que faire si l'on rencontre une déclaration de fonction ?
    - Mémoriser le nom
    - Analyser la fonction : mémoriser les variables de cette fonction, mémoriser les instructions de cette fonction, mémoriser le type de retour
- Ces actions sont précisées à la suite de l'analyse syntaxique

HELMo Saint - Laurent

Analyse sémantique (3)

- Exemple

```
program:
PROGRAM ENTER bloc procedure_bloc ENDPROGRAM
{   Leaf *l = (create(LEAF, UID)).l;
    l->uleaf.id = pid; /* pid = nom programme */
    $$ = endnode = (create(NODE,TPROGRAM)).n;
    insertNode($$, 0, l, LEAF);
    insertNode($$, 1, $3, NODE);
    insertNode($$, 2, $4, NODE);
}
```

Création de l'arbre

```
bloc:
BLOC ENTER declaration_var suite_instr ENDBLOC ENTER
{   Leaf* l = (create(LEAF,UID)).l
    l->uleaf.id = bid;
    $$ = (create(NODE, TBLOC)).n;
    insertNode($$, 0, l, LEAF);
    insertNode($$, 1, $3, NODE);
    insertNode($$, 2, $4, NODE);
}
```

Création de l'arbre

Source:

HELMo Saint - Laurent

Table des symboles (1)

- Le code source d'un programme contient
  - de nombreux identificateurs
    - noms de variables et constantes
    - noms de fonctions et procédures
  - Ces identificateurs ont **une portée définie**.
    - Un identificateur définit à l'intérieur d'une fonction n'est pas *visible* ailleurs dans le programme.
    - Un identificateur dans un bloc vs un identificateur au début du programme.
  - Ces identificateurs peuvent être associés à des **valeurs**.
    - Il est possible d'affecter des valeurs à des identificateurs, dont il faut se souvenir.

HELMo Saint - Laurent

Table des symboles (2)

- Afin de maintenir ces identificateurs, les compilateurs construisent une table des symboles.
  - Qui prend souvent la forme d'une table de hachage, dont le nom est la clé.
  - Cette table retourne très souvent un pointeur vers la zone mémoire allouée à cet identificateur.
  - Il s'agit également d'une pièce essentielle dans la construction d'un compilateur

HELMo Saint - Laurent

Génération de code (1)

- Une fois l'arbre construit, il faut passer à l'étape suivante:
  - Interpréter le programme
  - Produire du code machine
- La génération de code est une étape relativement difficile dans les compilateurs car il faut **optimiser** le code générés
- Les règles d'optimisation ont parfois des effet non désiré, il faut donc être très prudent.

HELMo
Saint - Laurent

Génération de code (2)

- L'interprétation
  - Idée: parcourir l'arbre abstrait et exécuter les instructions rencontrées en fonction des définitions sémantiques réalisées à l'étape précédente.
  - L'interprétation est très souvent utilisée dans les *nouveaux* langages:
    - Ex: Java et .NET
    - Pourquoi ?
      - Car c'est un mécanisme **portable**. En effet, pour permettre l'exécution sur une autre plateforme, il faut simplement écrire une toute petite partie, la machine virtuelle.

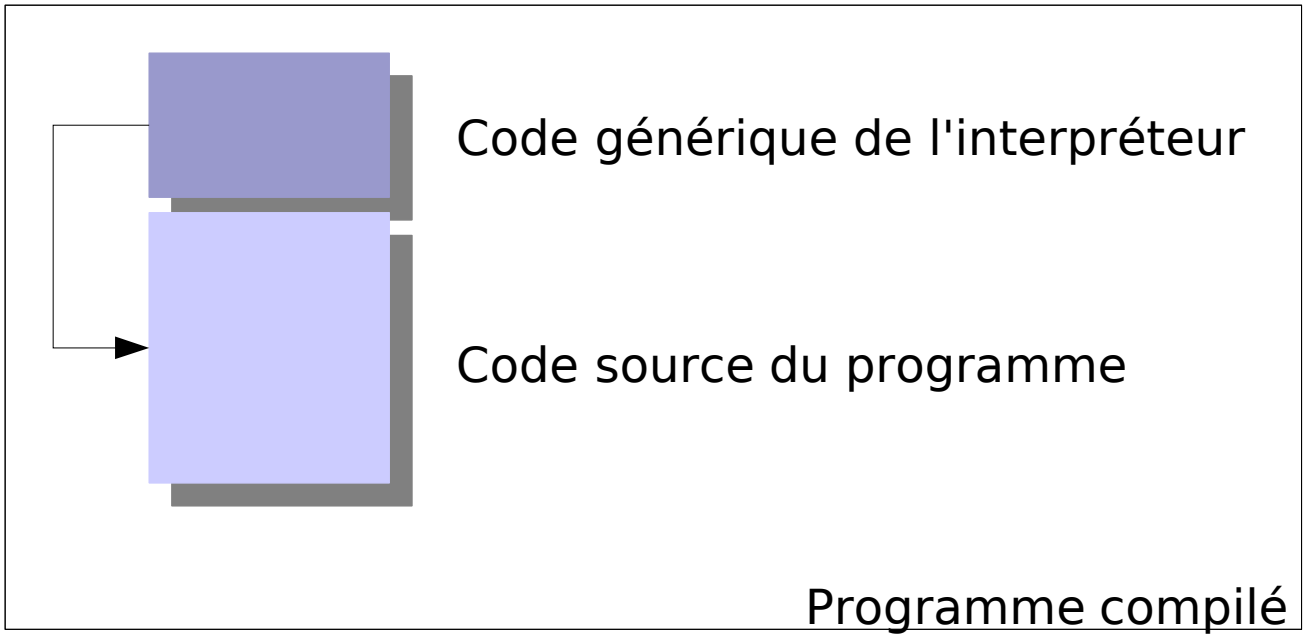
HELMo Saint - Laurent

Génération de code (3)

- Les machines virtuelles
  - En Java et C#, une *machine virtuelle* est implémentée
    - Cette machine virtuelle propose un langage de base de haut niveau
      - La Java Virtual Machine dipose de « son assembleur », qui reprend des instructions de base de haut niveau
    - Le compilateur Java traduit son fichier source en fichier compilé *bytecode* exécutable par la JVM
      - C'est-à-dire qu'un fichier *bytecode* contient du code machine compréhensible par une machine virtuelle Java (peu importe la plateforme sous-jacente)
    - Dès que la machine virtuelle est disponible dans plusieurs environnements, tous les programmes écrits deviennent *multi-plateforme*.
      - Ex: la JVM est disponible sous Windows, Linux, Mac, ... donc les programmes Java sont portables !

Génération de code (4)

- La génération de code
  - Une première solution naïve
    - Au lieu de compiler le programme en code machine, il est possible de créer:
      - un interpréteur
      - qui analyse le code qui lui est donné
      - qui exécute ce programme



HELMo Saint - Laurent

Génération de code (5)

- Génération de code machine
  - En parcourant l'arbre, le système génère le code correspondant
    - En fonction des instructions rencontrées, des segments de code machine sont copiés et assemblés dans le programme objet
    - Les librairies sont ensuite ajoutées.
  - La phase d'optimisation se déroule en même temps
    - Il faut que le code généré soit, sémantiquement, équivalent au code source du programmeur.
    - Il faut trouver des représentations efficaces du code source du programme
  - La génération peut se faire en assembleur ou directement en code objet exécutable par l'ordinateur
    - Le code généré n'est pas portable, il est prévu pour s'exécuter sur une architecture donnée

HELMo Saint - Laurent

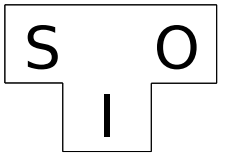
Cross-compiling (1)

- Comment les compilateurs sont-ils développés ?
  - Sur une nouvelle architecture, il n'existe pas encore de compilateur, comment faire ?
    - Exemple: une nouvelle console de jeu, un nouveau GSM, ...
  - La solution: *le cross-compiling*
    - L'idée est assez simple:
      - La machine d'une architecture X dispose d'un compilateur
        - traduisant d'un langage source Y
        - mais à destination d'une autre architecture Z.
      - On dispose donc d'un compilateur X traduisant le langage source Y vers le code objet Z

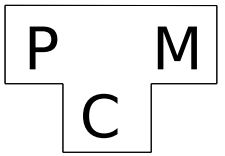
HELMo Saint - Laurent

Cross-compiling (2)

- On le représente souvent dans un *diagramme en T*



  - avec **S** le langage **source**
  - avec **I** le langage **d'implémentation** (du compilateur)
  - avec **O** le langage **cible**
  - Exemple:



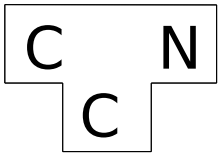
Représenterait un compilateur Pascal pour la machine M écrit en C

- Cette représentation est intéressante lorsqu'il s'agit de combiner des compilateurs entre-eux

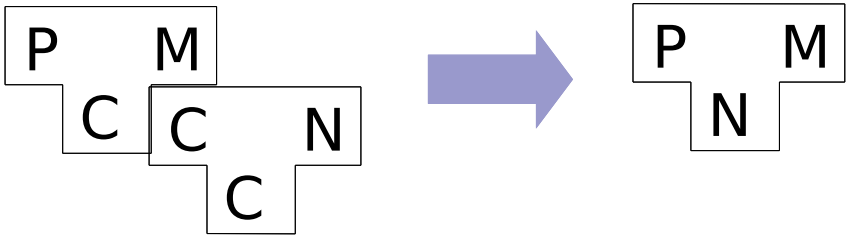
HELMo Saint - Laurent

Cross-compiling (3)

- Supposons que nous disposions, pour la machine N d'un compilateur C, nous avons :



- Si l'on compile le 1ier compilateur avec **ce** compilateur, nous obtenons :



Nous obtenons un compilateur Pascal s'exécutant sur la machine N et produisant du code pour le machine M

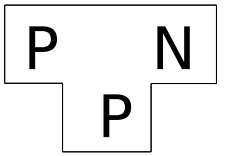
HELMo Saint - Laurent

Cross-compiling (4)

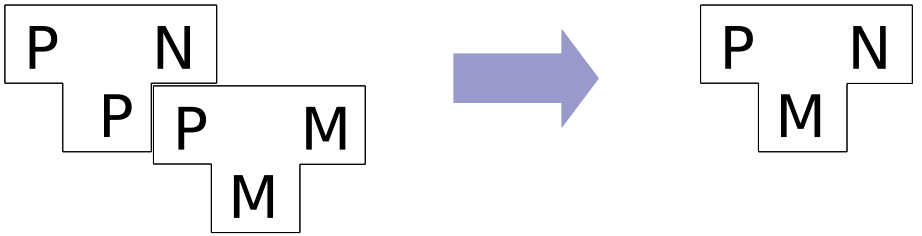
- L'auto-amorçage
  - Comment développer un compilateur C en langage C
    - C'est l'histoire de l'oeuf et de la poule !!
  - Idée: développer des compilateurs successifs permettant de couvrir des sous-ensembles du langage
    - Par exemple, la pascal a été écrit en pascal.
    - Ensuite un sous-ensemble du Pascal a été traduit « à la main » en code machine.
    - Ce compilateur limité a permis de compiler les versions suivantes du compilateur.
    - Cette technique s'appelle l'auto-amorçage.

Cross-compiling (5)

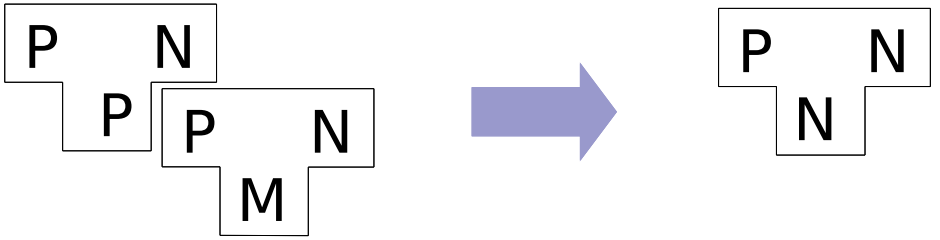
- Exemple:
  - Supposons que nous souhaitons écrire un compilateur Pascal, en Pascal pour la machine N.



  - En utilisant un compilateur Pascal pour la machine M, nous obtenons :



  - Si on compile **une seconde fois** ce compilateur :



Un compilateur Pascal pour la machine N !

HELMo Saint - Laurent

Cross-compiling (6)

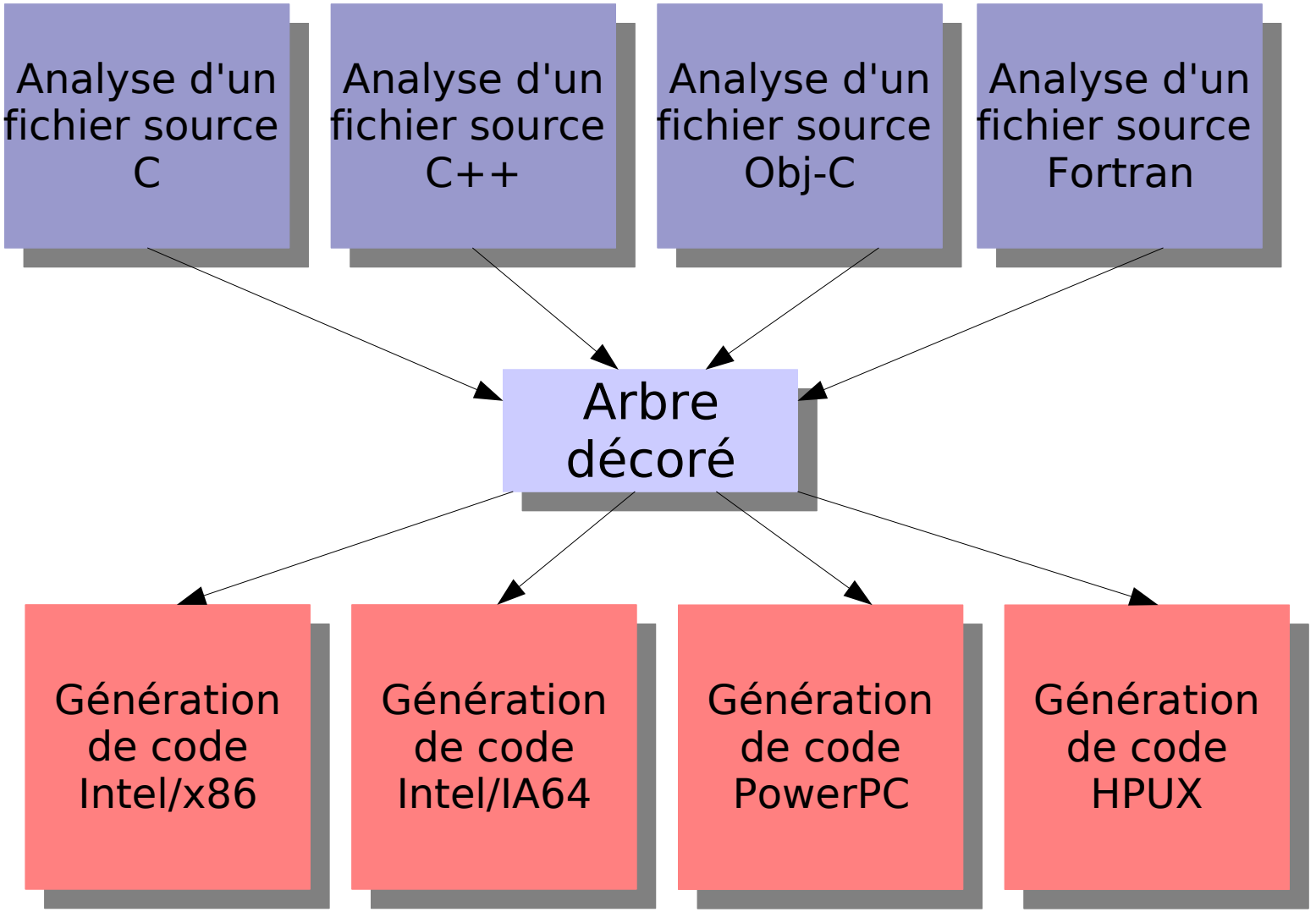
- Exercice

Soit le langage Y, l'ancienne version Y^{--} tourne sur une machine N. La nouvelle version du langage, Y^{+}, développé en Y^{--}, devrait tourner sur une nouvelle machine M. Comment faire ?

HELMo Saint - Laurent

Cross-compiling (7)

- Exemple d'un cross-compiler (GCC)



HELMo Saint - Laurent

Cross-compiling (8)

- Avantages
  - La construction de compilateur multi-langages permet de:
    - diminuer la tâche de développement lors de l'ajout d'un langage (il « suffit » de réaliser les analyseurs qui construiront l'arbre)
    - diminuer la tâche de développement lors de l'ajout d'une nouvelle plateforme. Il « suffit » d'écrire le générateur de code correspondant à cette plateforme.
  - Ces cross-compilateurs permettent de développer sur des plateformes très réduites, ne pouvant supporter un compilateur
    - Ex: un téléphone mobile
    - La mise au point d'un tel programme n'est cependant pas aisée.

Cross-compiling (9)

- Nécessiter de développer des simulateurs
  - En effet, pour mettre au point des applications sur des plateformes réduites, un simulateur devient vite nécessaire.
  - Le simulateur reproduit **l'environnement technique** de la plateforme cible. Par exemple, reprend « le système d'exploitation » de la plateforme cible
  - Grâce à ce simulateur, il est possible:
    - de mettre au point les différents programmes
    - d'exécuter du code prévu pour tourner sur la plateforme cible
- Par rapport à un interpréteur, le simulateur va reproduire/simuler les composants matériels de la plateforme cible.
- L'interpréteur traduit un code cible en instruction machine équivalente sur une autre plateforme
  - Pas de reproduction de l'architecture technique.

HELMo Saint - Laurent

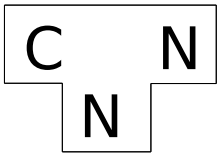
Exercices (1)

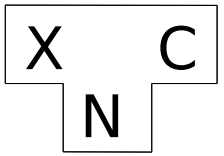
- CAML : créer une fonction qui
  - Retourne tous les éléments d'un tableau
  - Trouve un élément dans un tableau (retourne *true* ou *false* suivant le cas)

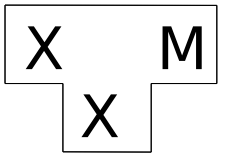
HELMo Saint - Laurent

Exercices (2)

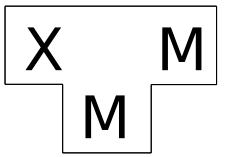
- Compilation
  - En disposant des éléments suivants:







    - Comment écrire un compilateur



HELMo Saint - Laurent